WCF 自習書

第 2 部 サービス開発 WCF 編

制作者: エディフィストラーニング株式会社

目次

はじめに

この第 2 部の WCF 編では、WCF の具体的な実装方法や関連する Visual Studio 2010 の使用方法について取り上げます。ここでは、実際のアプリケーションの中で、WCF 関連のテクノロジをどう活用するのかを確認していただくために、原則として予め完成してあるサンプルプログラムを題材にします。サンプルプログラムの中から、重要なコードの実装例や、そのコードに至る Visual Studio の操作のポイントなどを確認します。また、バリエーションとして、一部のサンプルについては、簡単なものを新規に作成します。

ここでは、WCF テクノロジが提供するすべての機能を網羅的に解説するわけではありませんが、ここで取り上げたことは、今後、開発者のみなさんが WCF を使用して実装する際に、それらを使いこなす上での「着眼点」や「指針」、「コツ」など得ることができるでしょう。

ここでのサンプルプログラムは、フェリーやバスなどの交通機関を運営する架空の企業である Adventure Works Transport Services において、フェリーの予約管理や乗船手続き、関連するマイレージの処理を行うシステムです。顧客向けのインターネットに公開される ASP.NET Web サイトと、WPF アプリケーションをクライアントとするイントラネット向けの多層アプリケーションから構成されています。以降は、「AWTS サンプルプログラム」と表記します。ポイントとなるサンプルコードは、ドキュメントに記載しますので、サンプルの実行環境がなくとも、感触をある程度つかむことができると思いますが、理解をより深めるためにも、サンプルプログラムをセットアップして動作確認をしながら、読み進めることをお奨めします。なお、AWTS サンプルプログラムのセットアップ方法は、別途、「AWTS サンプルセットアップガイド」をご参照ください。

前提知識

サンプルの動作を確認するにあたり、Visual Studio の基本的な操作手順等については割愛しており、当ドキュメントに沿って操作する上での前提知識として必要になります。また、ソースコード例の解説では、紙面の都合もあり、そのコードの特徴的な部分に焦点を当てて説明しています。そのため、基本的な言語文法の知識も前提として必要になります。サンプルは主に C#で記載されていますが、WCF のポイントとなる部分については、Visual Basic で実装した場合の参考コードが、「_note.txt」というファイルの中に記述されています。また、C++、Java などのいずれかのオブジェクト指向プログラミング言語の知識をお持ちであれば、サンプルコードの内容は想像がつくでしょう。

具体的な前提知識としては、主に以下の事項に関して必要となります。

- Visual Studio のソリューションやプロジェクトを開き、プログラムコードを編集することができる。また、ソリューションとプロジェクトの違いを理解している。
- Visual Studio の各編集可能のウィンドウ、たとえば、フォームデザイナーやコードエディターなど、特定の機能のウィンドウを開き、使用することができる。
- ビルドやデバッグの実行方法や、Web アプリケーションの実行方法が分かる。
- ソリューションに含まれる複数のプロジェクトの中から、特定のプロジェクトを実行することができる。

- 1 つのプロジェクトから、別のプロジェクトを参照するための「参照の追加」(参照設定) を行うことができる。
- SQL Server Management Studio、または、Visual Studio などを使用して、SQL Server データベースに格納されたデータを確認できる。
- Visual Basic、または C#、C++ などのオブジェクト指向言語の基本的な文法が理解できる。特に、クラスライブラリを使用する上での、インスタンスの作成やメソッド呼び出し、プロパティの参照や設定など。

なお、第 1 部では、WCF の基本的な特徴について既に確認しています。特に、次に挙げる用語の意味などは、第 1 部を参照してください。

- サービス コントラクト、サービス クラス、およびホスト アプリケーション
- エンドポイント、バインディング
- プロキシ、サービス参照の追加

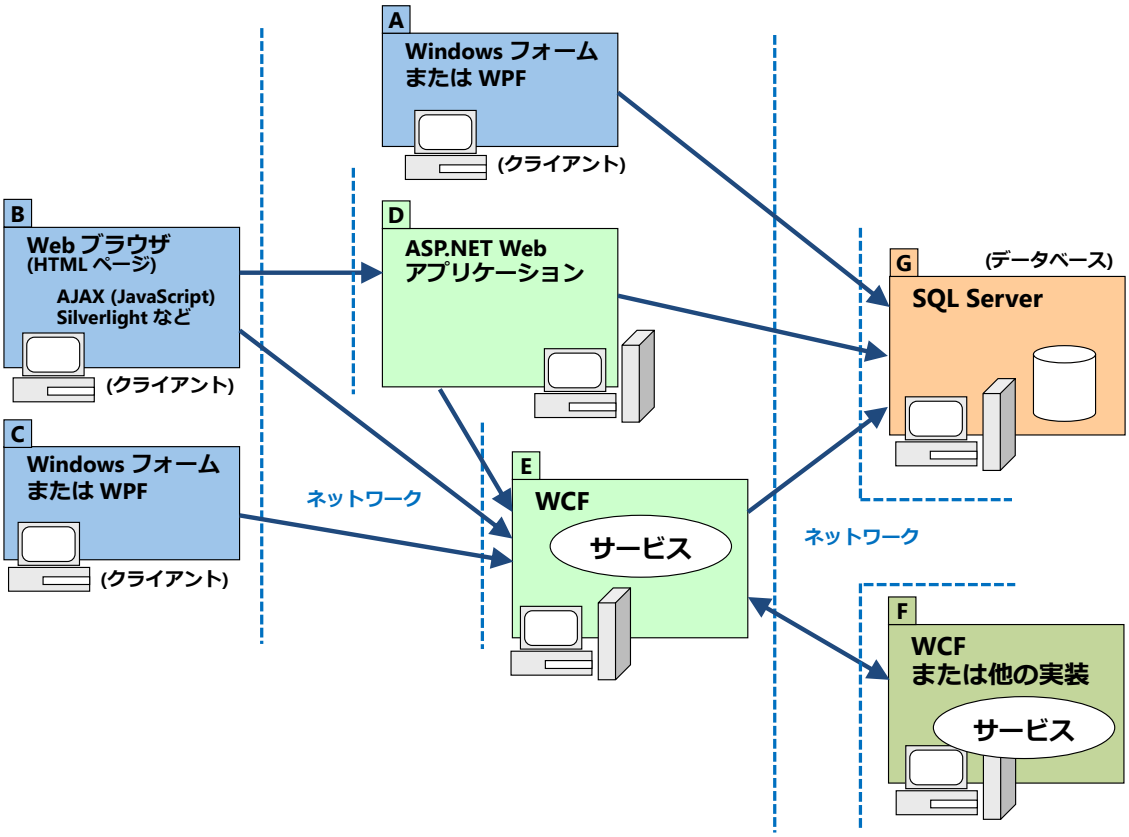
第 1 章 サンプルのシステム構成と特徴

まずは、AWTS サンプルプログラムの全体構成を把握し、システムのどの部分で、WCF のどの機能をどう活用しているか確認しましょう。なお、サンプルの基本的な操作方法は、セットアップガイドの後半部分である「サンプル操作方法ウォークスルー」を参照ください。

1.1 サンプルのシステム全体像と WCF の各機能

次図は、第 1 部で触れた WCF サービスを含む、.NET における典型的なシステム構成例の再掲です。この図には、WCF サービスが使用されている個所が示されているほか、様々な .NET テクノロジが含まれています。

図 1. .NET Framework を用いたシステムの全体像（再掲）

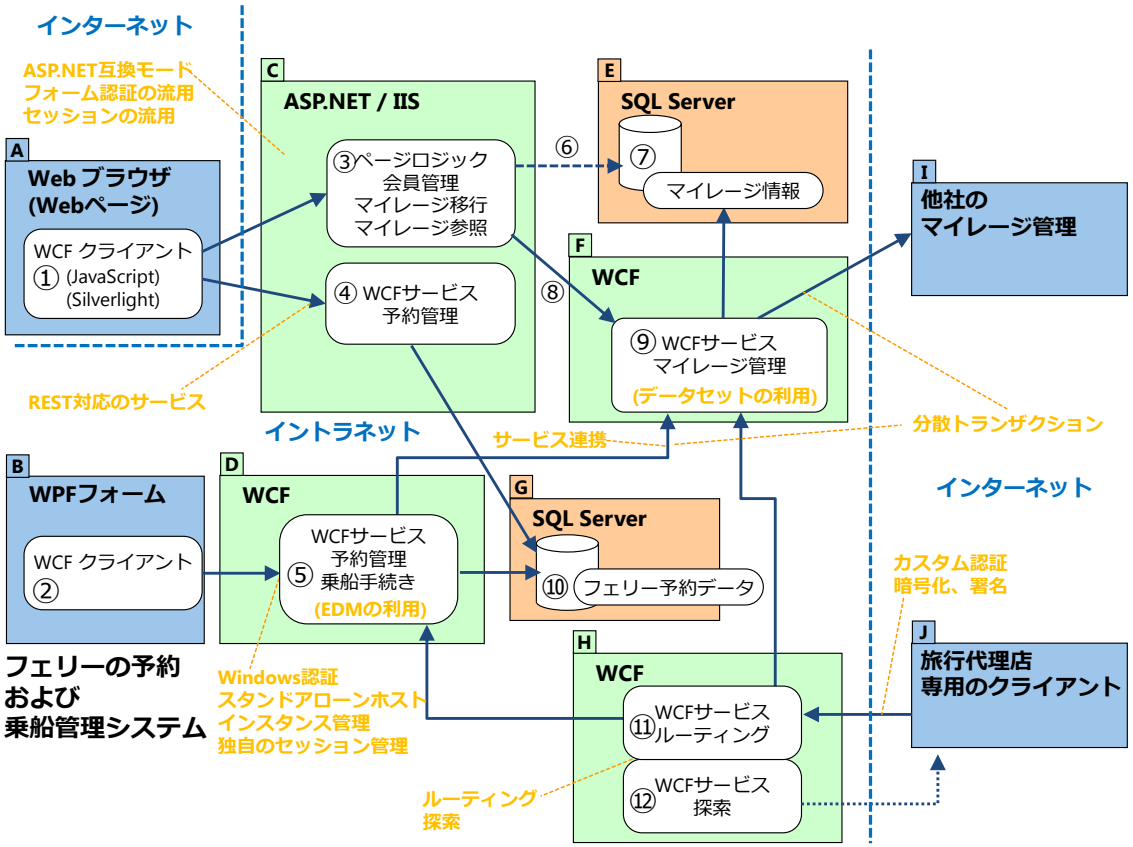


このシステム全体図では、1 つの分散システムの中で、E の四角形ブロックのところで、WCF サービスが利用されています。C、E、および G からなる 3 層システム、また、B、D、E、および G からなる多層システムの中で、中間層として使用されています。このほか、F の他のシステムの WCF サービスと連携しています。

これらの点を反映した AWTS サンプルプログラムのシステム全体図を次に示します。次図では、特に WCF 関連のテクノロジのうち、どの機能が使用されているかという点も示しているほか、このサンプルプログラムを実装する際に使用された、そのほかの主なテクノロジや機能が示されています。この図 2 も前述の図 1 と同様に、A や B などのブロックは論理的なソフトウェアの構成単位であって、必ずしも、物理マシンに個別に配置されるとは限りませんが、おおよそこのブロックを単位として、

マシンも分けることになるでしょう。ただし、今回のサンプルプログラムは、1 台のマシンで動作するように構成されています。（図 1 と図 2 のそれぞれのブロックに付いた英字は、お互いに対応しているわけではなく、それぞれの図の中で、順番に割り振っているだけです。）

図 2. AWTS サンプルプログラムのシステム全体像



この図 2 では、この架空企業のバスやフェリーの総合案内などを行う Web サイトが、顧客向けにインターネットに公開されているほか、企業内のイントラネット上には、社員が操作するフェリーの予約管理システムがあります。インターネット向けの Web サイトに関わる構成要素は、A、C、E、および F です。一方、イントラネット向けのシステムに関わる要素は、B、D、および G です。

そのほか社外のシステムとも連携しており、この企業で管理する顧客のマイレージを、他社のマイレージシステムに移行できるよう、F と I との間で連携しています。また、外部の旅行代理店からもフェリー予約の管理ができるよう H と J で連携しています。

ここで、この図 2 のシステム全体像に関して、図 1 のシステム全体像と照らし合わせ、どこで WCF サービスが利用されているか確認してみましょう。

上記の 2 つの図のうち、前者（図 1）における E の WCF サービス（中間層）は、後者（図 2）のサンプルプログラムのシステム全体図では、D や F、H に当たります。また、前者の図 1 における E と F とのサービス連携に相当するものは、図 2 では F と I との間、また、H と J との間になります。このほか図 2 では、C の ASP.NET Web サイトは WCF サービスのホスティング環境として利用できるので、この ASP.NET Web サイト環境でも WCF サービスが実行されています（④の部分）。

また、このサンプルでは、これらの WCF サービスの実装の中に、WCF サービスが提供する様々な機能を使用しています。ここで、どのような機能が使用されているか確認してみましょう（主な機能は、オレンジ色の注釈として、図 2 の中に記載されています）。

先述のとおり、図 2 の C の ASP.NET Web アプリケーションでは、WCF サービス（④）のホスティングも行っています。さらに、ここでは「ASP.NET 互換モード」を使用しており、ASP.NET のフォーム認証や ASP.NET のセッションを、そのまま④の WCF サービスにアクセスする際の「認証」や「セッション管理」に利用できます。

図 2 の C の ASP.NET Web サイトにおいて、データベースへのアクセスを伴う一般的な多層システム構成を構成する場合は、A、C、および E の構成です。このうち、マイレージ情報を持つ⑦のデータベースにアクセスする際には、点線で示すように、③の Web ページ上のマイレージ管理を行うロジックが記述されたコードの実装から、何らかのデータアクセステクノロジーを利用して、データベースを参照することになります。ただし今回、このマイレージ管理機能は、他の様々なシステムの構成要素からアクセスされるので、F のように、中間層の WCF サービスとして独立させました。よって、A の Web クライアントから C の Web サイトを介して、マイレージ管理を行う基本的な流れは、順に A（①）、C（③）、F（⑨）、E（⑦）となります。今回は、点線⑥の流れを使用しません。

一方、C の Web サイトの中でフェリー予約に関する実装は、④の WCF サービスです。④の WCF サービスにアクセスするために、①の Web ページ上の JavaScript や Silverlight を使用しています。このサンプルでは、スクリプトのような軽量な実装のクライアントからもアクセス可能にするため、④の WCF サービスを「RESTful」なサービスとして構築しています。

なお、この部分は別の選択肢として、WCF RIA Services や WCF Data Services も利用できます。既に使い分けの指針については第 1 部で説明しましたが、このサンプルでは単純な予約データのやり取り行っており、基本的な REST 対応サービスを実装しています。また、クライアントからは簡単なスクリプトを用いてアクセスしており、参考として、Silverlight 版も用意しています。この REST 対応サービスの部分については、第 3 部で取り上げます。

註: WCF RIA Services については、次のアドレスから学習を目的としたサンプルアプリケーションを入手することができます。
http://msdn.microsoft.com/ja-jp/silverlight/hh219352
Visual Studio 2010/Silverlight 4 ソリューション サンプル (MVVM) 編 ～ Silverlight 4 + WCF RIA Services + MVVM ～

註: WCF Data Services については、以下のアドレスから自習書を入手することができます。
http://msdn.microsoft.com/ja-jp/data/gg615417 データ アクセス自習書
「データ アクセス自習書 (Visual Studio 2010/.NET4 版)」の中の
「第 3 部 SQL Server データ アクセス手法 - (4) WCF Data Services 編」

このほか、図 2 の F の WCF サービスには、企業内のマイレージデータを他社のマイレージシステムに移行する機能があり、I の他社の WCF サービスと連携しています。ここでは、マイレージデータの一貫性を維持するため、複数の WCF サービスをまたぐ、「分散トランザクション」として実装されています。この部分では、場合によっては他社システムのプラットフォームの違いや、常時オンライン

にできないという理由から、分散トランザクションが実現できない場合もあります。その場合には別の選択肢として、日次バッチ処理に対応するために、WCF サービスを用いて社内のキュー（MSMQ）などにマイレージデータを投函して一時的に退避し、定期的に移行処理を行う方法も考えられます。また、企業内のマイレージデータの取出しから、他社システムへのデータ移行の一連の流れをワークフローとして構築して、この WCF サービスを「ワークフロー サービス」として実装する方法も選択肢として考えられます。

註: WCF におけるキューの使用については、以下のアドレスを参照してください。
http://msdn.microsoft.com/ja-jp/library/ms732355.aspx キューと信頼できるセッション

註: Windows Workflow Foundations、および WCF におけるワークフロー サービスについては、以下のアドレスを参照してください。
http://msdn.microsoft.com/ja-jp/library/dd489441.aspx Windows Workflow Foundation
http://msdn.microsoft.com/ja-jp/library/dd456788.aspx ワークフロー サービス

図 2 において、イントラネットのフェリー予約や乗船の管理を行うシステムとして、B（②）、D（⑤）、および、G（⑩）からなる分散システムを構成しています。ここでも、⑤の部分で中間層として WCF サービスを用いてします。

⑤の WCF サービスでは、スタントアローンのホストアプリケーションとして実装しています。また、クライアント（②）からのアクセスには、社員が使用する Windows ユーザーアカウントを用いた「Windows 認証」に対応するようにしています。この⑤のサービスでは、アクセスする社員ごとに異なるステートを管理できるようにするため、WCF の「インスタンス管理」や「セッション管理」の機能を用いて、社員のユーザーアカウントごとに、サービスインスタンスを用意しています。

さらに、⑤の WCF サービスでは乗船手続きの際に、マイル加算をするため、⑤から⑨の WCF サービスを利用しており、2 つの WCF サービスが連携しています。マイル加算の際には、⑤のマイル情報と、⑨のマイル情報との一貫性を維持するため、ここでも分散トランザクションを使用しています。

また、外部の J は架空の旅行代理店がフェリーの予約手続きを行うための WCF クライアントです。これは、専用のアプリケーションであり、H のサービスにアクセスする際にセキュリティを強化するため、「カスタム認証」やメッセージの「暗号化」、ならびに「署名」を利用しています。

本来ならば、この J の WCF クライアントは、この図 2 のシステム内に点在する WCF サービスの各エンドポイントにアクセスする必要があるのですが、わざわざ外部の旅行代理店向けに、それぞれの WCF サービスで個別にエンドポイトンを用意するのは保守の手間が増えます。そこで、旅行代理店に公開しているエンドポイントは、⑪の WCF サービスが持つエンドポイントだけに集約しています。⑪では、WCF のルーティングサービスが実装されており、J のクライアントからの要求に応じて、⑤と⑨の各サービスへ要求を転送しています。

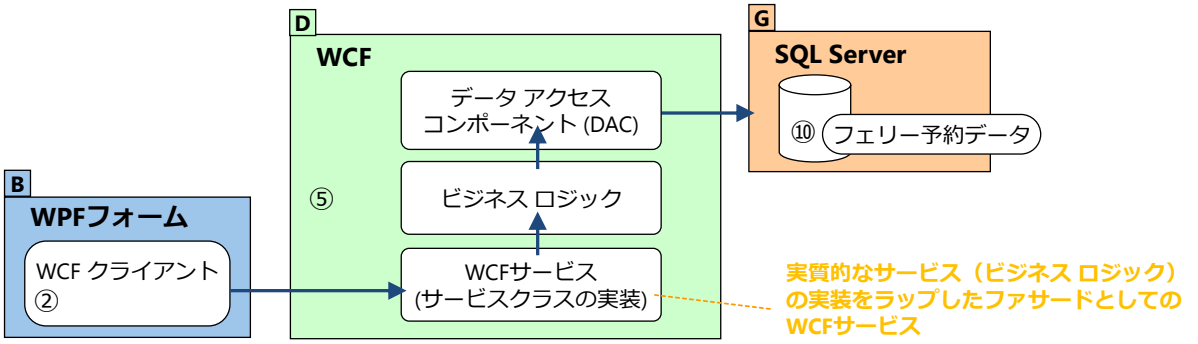
また、この⑪のルーティングサービスは、サービス環境の保守の際に、代替の別のサーバーで実行され、異なるエンドポイントで公開されます。このとき、クライアントは動的にアクセス先を認識できるようにするため、クライアントは⑫の探索用のサービスを利用して、エンドポイトンの変更を認識できるようにしています。

なお、⑤と⑨のサービスでは、クライアントとデータのやり取りを行う際のデータコントラクトとして、異なるデータ構造を使用しました。それぞれのサービスでは、カスタムデータ型を用いるほか、⑤の予約管理サービスでは、Entity Data Model のエンティティを利用しており、一方、⑨のマイレージ管理サービスでは、ADO.NET データセットも使用しています。それぞれのデータ構造に関してデータコントラクトとしての使用方法や特徴、留意点についても、この第 2 部の後半で取り上げます。

1.2 WCF サービスの実装における基本的な構成

なお、図 2 に記載された WCF サービス（⑤や⑨など）は、論理的に 1 つのサービスを提供する単位とみなせるものを表していますが、実際の実装では、一般的に複数のソフトウェア コンポーネントに分割されて構成されています。たとえば、図 2 の D にある⑤の WCF サービスは、実装の役割に応じてソフトウェア コンポーネントを分割すると、次のようになります。

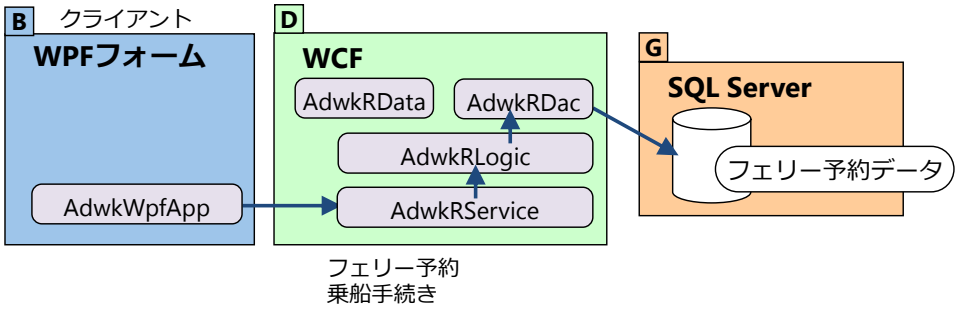
図 3. サービスを構成する各役割のソフトウェア コンポーネント



たしかに、WCF の 1 つのサービスクラスの中に、ビジネスロジックやデータアクセスのコードを記述することも可能です。しかし、一般的には保守性を高めるために、この図のように、データ アクセスを専用に行うデータアクセス コンポーネント（DAC）や、DAC の操作を伴うビジネス ロジックを実装したコンポーネントなど、役割に分けて実装します。そして、このビジネスロジックにネットワーク経由でアクセスするための入口（ファサード）として、WCF サービスを実装します。この 3 つのコンポーネントを実装する最も簡単な方法は、それぞれを異なるクラスとして実装する方法です。

また、この 3 つのコンポーネントから共有されるデータ構造などは、別のコンポーネントとして分割する方法もあるでしょう。今回のサンプルでも、そのように分割しており、この WCF サービスに関しては、次のように 4 つのプロジェクトから構成されています。

図 4. 予約管理サービス（および乗船手続きサービス）のプロジェクト構成



この場合、WCF サービス、ビジネスロジック、および、データアクセスに関しては、それぞれ順に、AdwkRService プロジェクト、AdwkRLogic プロジェクト、AdwkRDac プロジェクトに実装されています。また、各プロジェクトで使用する共通のデータ構造を AdwkRData プロジェクトに実装しています。

ただし、さらに分割する余地が残っています。本格的に作成するのであれば、WCF サービスに関しては、「サービス ホスト」、「サービス クラス」、および「コントラクト」をそれぞれ別のプロジェクトに分割するのが適切でしょう。そのようにすれば、サービスホストの環境が変わっても、サービス本体（サービス クラスとコントラクト）は再利用しやすくなります。また、「コントラクト」（サービス コントラクトとデータ コントラクト）は、他の環境でも使い回しができるようになります。特に、.NET Framework のクライアントで「チャネル ファクトリ」を用いて実装する場合は、予め「コントラクト」のアセンブリが用意されていると、クライアントのプロジェクトからそのアセンブリを参照して、直ぐにクライアントの作り込みが行えるので便利です。

最後に、今回のサンプル プログラムにおけるプロジェクト構成の全体像と、それぞれのサービスを実行する際の最低限必要なプロジェクト（ソリューション）の構成を掲載しておきます。以降の章でサンプルを実行し検証する際に、これらの図を必要に応じて参照してみてください。

図 5. AWTS サンプル プログラムにおけるプロジェクト構成の全体図

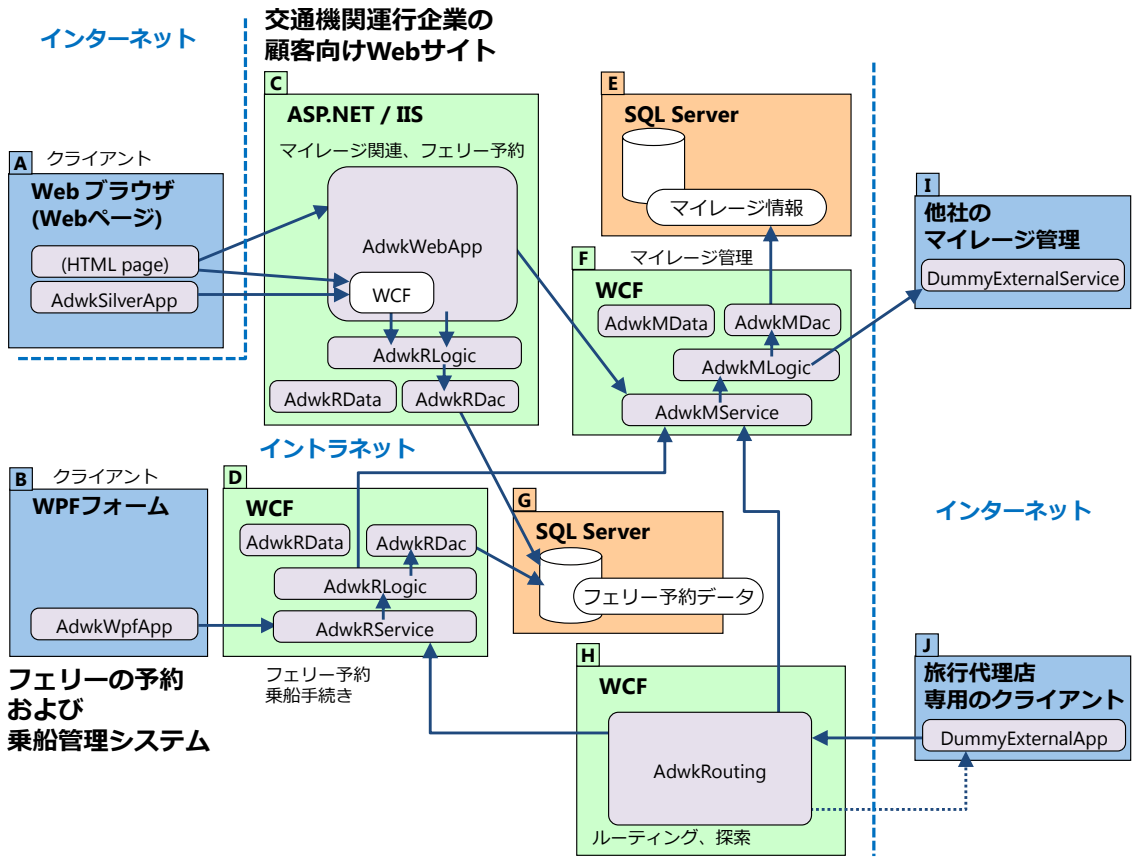


図 6. 顧客向け Web サイト実行時の構成

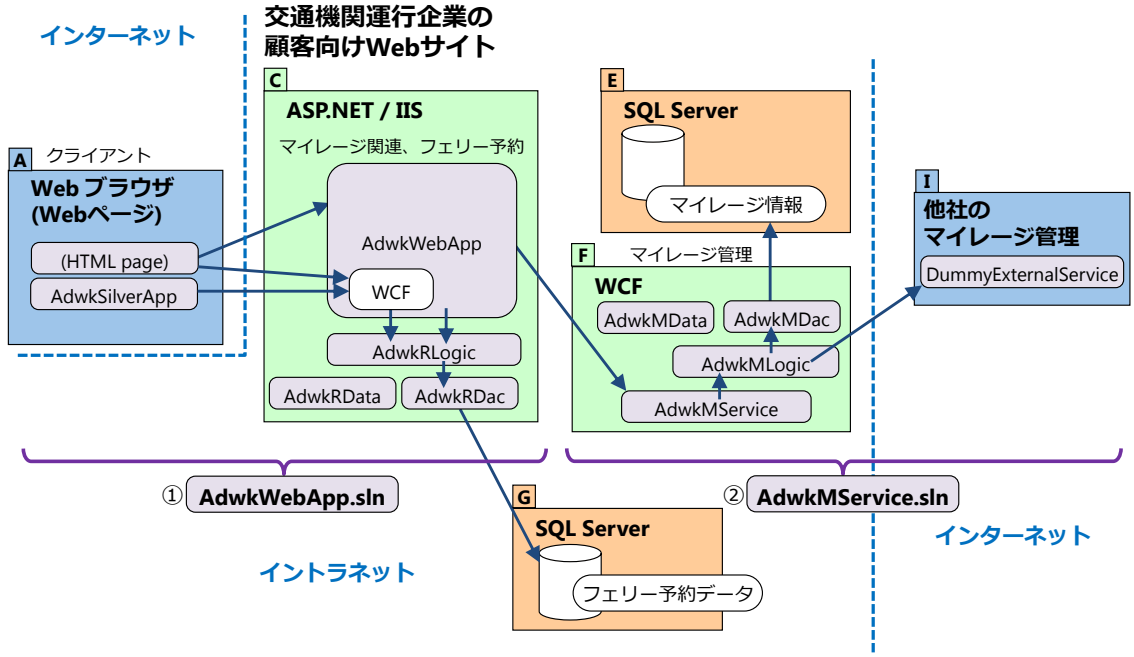


図 7. 社内向けシステム実行時の構成

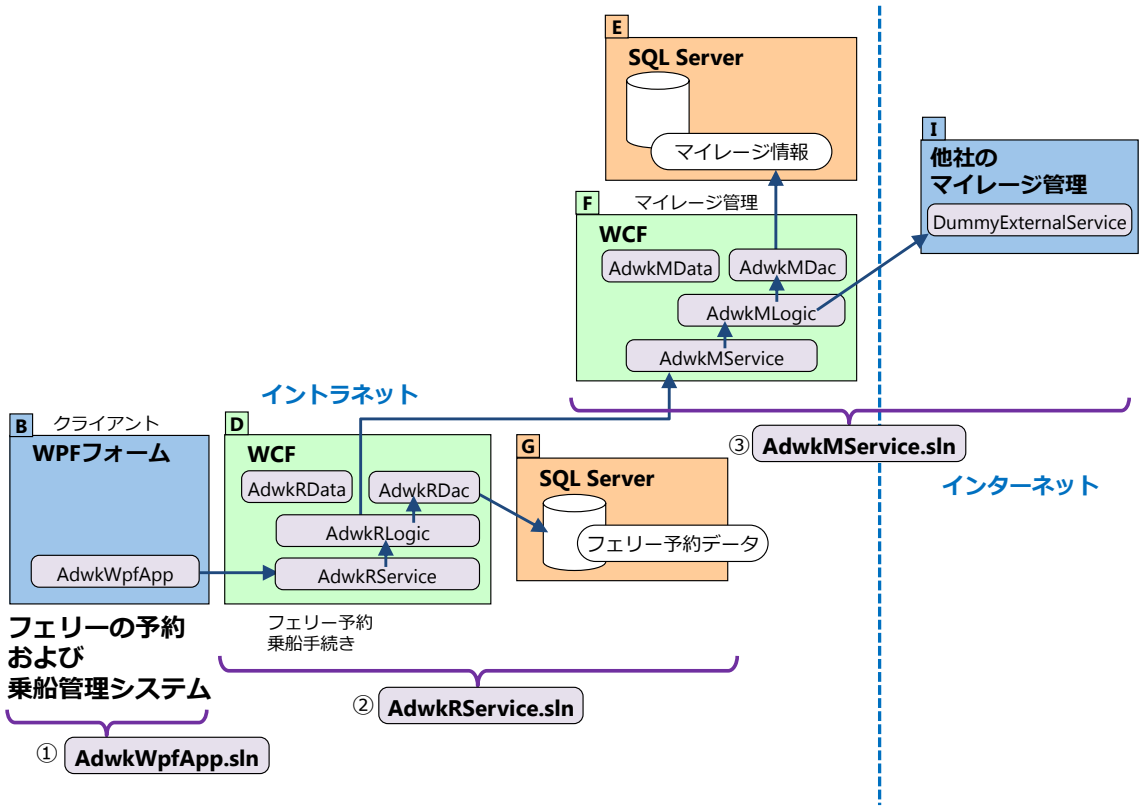
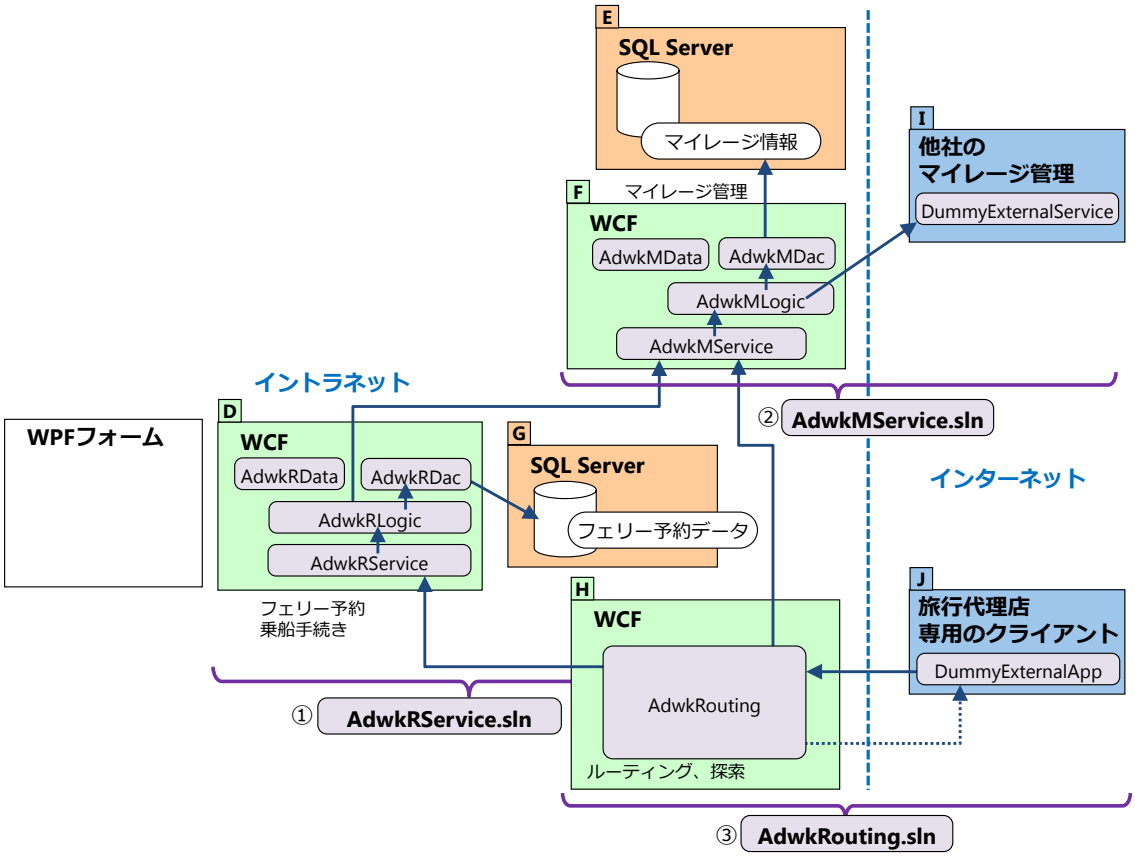


図 8. 社外の旅行代理店からの利用



註: このサンプルプログラムでは、WCF を用いた実装方法に焦点を当てているため、WCF に直接関係しないデータ構造やビジネスロジック、および、ユーザーインターフェイスに関しては、いくつか簡略化しています。
たとえば、マイレージ会員情報を扱うテーブルでは、一般的には生年月日や性別、住所などの列を含みますが、このサンプルでは、会員を識別する会員コードのほか、姓と名のみを情報として使用しています。また、会員の検索やフェリーの運航予定の検索画面でも、その検索方法は限定的なものになっています。

第 2 章 基本的な実装方法

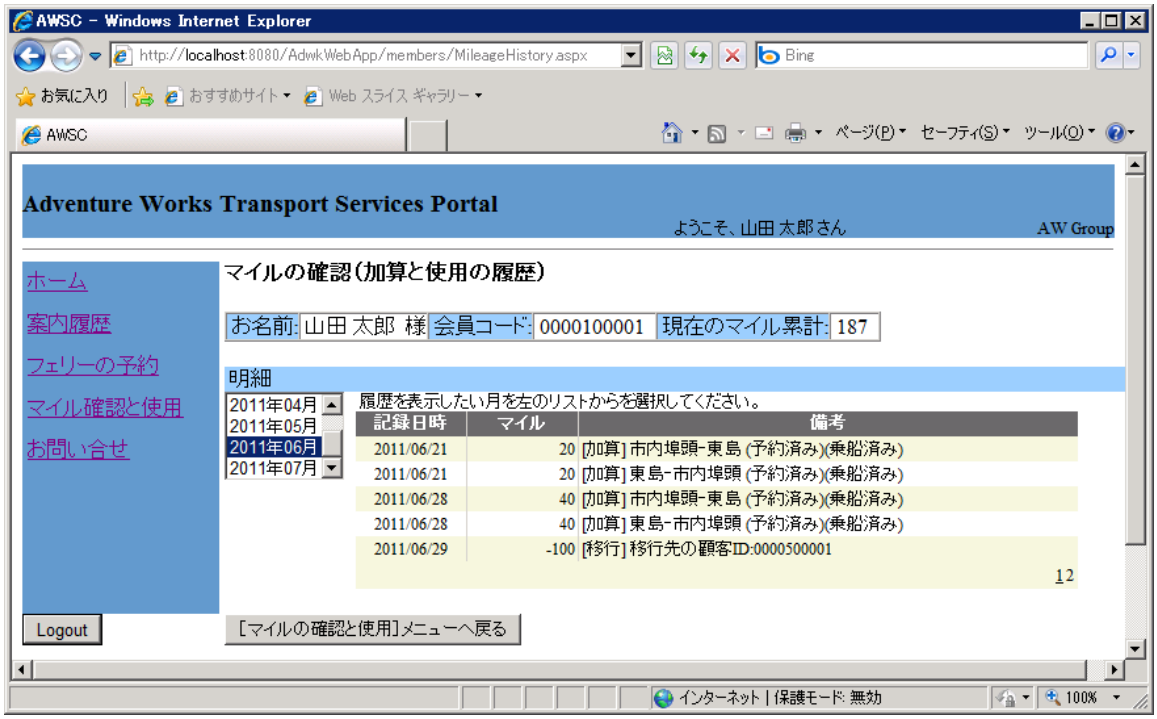
この章では、WCF サービスとクライアントの基本的な実装手順を改めて確認します。まず、既存の AWTS サンプル プログラムの一部を流用し、基本的な WCF サービスを作成した後、そのサービスにアクセスする簡単なクライアントを新規に作成します。また、AWTS サンプル プログラムで使用されている、いくつかの基本的な実装のバリエーションについても確認します。

2.1 基本的な WCF サービスの実装

このサンプル プログラムの ASP.NET Web アプリケーション（AdwkWebApp）では、次図のように、会員のページには名前やマイル残高の増減に関する履歴一覧が表示されます。これらの情報は、Web アプリケーションからマイレージ管理サービス（AdwkMService）にアクセスして取得したものです。このアクセス経路を図 5 のプロジェクト構成図に当てはめると、C の AdwkWebApp から F の AdwkMService にアクセスする部分に当たります。

この節 2.1 では、サンプルそのものを吟味する前に、WCF サービスの基本的な作成手順を確認するため、このマイレージ管理サービスに簡単なサービスを追加してみます。第 1 部で触れた WCF サービスの作成手順に基づき、このあとは順に作り込んでいきます。

図 9. マイレージ管理サービスから取得した会員情報を表示



> **註:** これ以降は、元の状態の AWTS サンプルプログラムに対して修正する場合が何度かあるので、元の状態のサンプルをバックアップとして残しておいてください。

・サービス コントラクト（インターフェイス）の定義

ここでは、マイレージの会員コードを引数として渡すと、会員の姓と名を返す簡単なサービスをAdwkMService プロジェクトの中に実装します。まず、Visual Studio 2010 を使用して AdwkMService.sln ソリューションを開き、AdwkMService プロジェクトの中に、次のサービス コントラクト（インターフェイス）を追加してください。以下に続く説明も参考にしてください。なお、保存する際のファイル名とプロジェクト名は各ソースコードの冒頭に記載してあります。

例 1. 会員の姓名を取得するサービスのコントラクト

IMBasicInfo.cs（AdwkMService プロジェクト）

```
using System.ServiceModel;  //←[1]

namespace Adwk.Mileage
{
    [ServiceContract(Namespace="http://Adwk.Mileage")]  //←[2]
    public interface IMBasicInfo                        //←[3]
    {
        [OperationContract]  //←[4]
        string QueryNameByMember(string memberCode);     //←[5]
    }
}
```

ここでは、[3]の IMBasicInfo インターフェイスをサービス コントラクトとして使用します。このインターフェイスでは、[5]の QueryNameByMember メソッドが定義されており、引数として会員コードを渡すと、戻り値として会員の姓名を返すことにします。なお、このインターフェイスはサービス側だけでなく、クライアント側でも利用することがあるので、名前空間はこのプロジェクトの既定である「Adwk.Mileage.Service」ではなく、「Adwk.Mileage」としました。

このインターフェイスをサービスコントラクトとして使用するには、[2]のようにインターフェイス自体に ServiceContract 属性を付け、また、[4]のようにメソッドには OperationContract 属性を付ける必要があります。この 2 つの属性は、クラスライブラリの名前空間「System.ServiceModel」に定義されており、属性の表記に完全修飾名を書かずに済むようにするため、[1]の using ディレクティブを使用しました。

また、[2]の ServiceContract 属性には、Namespace プロパティを指定することができます。このプロパティはサービスとやり取りすべき SOAP メッセージ（XML データ）の名前空間などに反映され、他のサービスのメッセージと重複することを避けるために使用します。

なお、IMBasicInfo インターフェイス自体をパブリックにすることは、WCF サービスとしては必須ではありません。しかし、本来であれば他のプロジェクトからこのインターフェイスを再利用することもあるので、ここでもパブリックとします。

・サービス クラスの実装

次に、前述の IMBasicInfo インターフェイスを実装するサービスクラスを記述します。AdwkMService プロジェクトに次のクラスを追加してください。

例 2. サービスコントラクト（IMBasicInfo インターフェイス）を実装したクラス

MileageBasicInfoService.cs（AdwkMService プロジェクト）

```
using Adwk.Mileage.Data;  //←[1]
```

```
using Adwk.Mileage.Logic; //←[2]

namespace Adwk.Mileage.Service
{
    public class MileageBasicInfoService  //←[3]
        : IMBasicInfo                     //←[4]
    {
        string IMBasicInfo.QueryNameByMember(string memberCode) //←[5]
        {
            bool result;
            AdwkMStatus stat;
            string msg;
            var table = new AdwkMileagesDataSet.会員マイル累計DataTable(); //←[6]

            result = BLComp1.QueryInfo1ByMember(   //←[7]
                        memberCode, table, out stat, out msg);
            if (result && table.Rows.Count >= 1)
            {
                return table[0].姓 + " " + table[0].名;  //←[8]
            }
            else
            {
                return "";
            }
        }
    }
}
```

[3]がサービス クラスであり、[4]の IMBasicInfo インターフェイスを実装しています。ここでも[3]をパブリック クラスにすることは必須ではないのですが、このサービス クラスを使用するホスト アプリケーションが別のプロジェクトの場合もあるので、パブリックとしました。

IMBasicInfo インターフェイスのメソッドは、[5]に実装してあります。ここでは、C#の構文の「明示的なインターフェイスの実装」を使用しましたが、明示的ではない実装も使用できます。

このサービスはファサードとして使用しているので、[5]のメソッドの内部では、会員情報を取得するデータアクセスなどの詳細なコードを直接記述せず、その代わりに[7]で、ビジネスロジックが実装されたコンポーネント（ビジネス コンポーネント）である BLComp1 を利用し、そのメソッドを呼び出しています。このコンポーネントは[2]の名前空間に属しており、AdwkMLogic プロジェクトの中に記述されています。

また、このコンポーネントのメソッドを呼び出す際には、メモリ上の会員情報の受け皿として、[6]にある ADO.NET の「型指定されたデータテーブル（DataTable 派生クラス）」が必要です。このデータテーブルは、[1]の名前空間に属しており、AdwkMData プロジェクトの中で定義されています。

ここでは相互運用性を重視して、型指定されたテータテーブル構造自体を公開しないようにするため、サービスの戻り値としては [8]のように string 型として、会員の姓名を返しています。（型指定されたデータテーブルをサービスの外部に返す例は、別途取り上げます。）

これで、サービス クラスが出来上がりました。単純なサービス クラスの実装の場合、サービス クラス自体には、WCF の属性など、WCF 固有の記述は登場しません。

・サービスの構成（構成ファイルでの定義）

次にサービスを公開するために、エンドポイントなどの構成を行います。今回のサンプルでは、すでに AdwkMService プロジェクトの中に、アプリケーション構成ファイル（App.config）があるので、この中に構成を追記します（黄色の太字部分）。

例 3. サービスのための構成

App.config（AdwkMService プロジェクト）

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>    ←[1]
    <services>             ←[2]
      <!-- 基本的な会員情報サービス -->
      <service name="Adwk.Mileage.Service.MileageBasicInfoService"  ←[3]
               behaviorConfiguration="MyBehaviors">  ←[4]
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8091/AdwkMService/Sample" />  ←[5]
          </baseAddresses>
        </host>
        <endpoint    ←[6]
          contract="Adwk.Mileage.IMBasicInfo"  ←[7]
          binding="basicHttpBinding"           ←[8]
          address="BasicInfo" />               ←[9]
        <endpoint    ←[10]
          contract="IMetadataExchange"
          binding="mexHttpBinding"
          address="mex" />
      </service>

     (略)

    <behaviors>
      <serviceBehaviors>
        <behavior name="MyBehaviors" >  ←[11]
          <serviceMetadata httpGetEnabled="true" />  ←[12]
        </behavior>
      </serviceBehaviors>
    </behaviors>

     (略)

  </system.serviceModel>
</configuration>
```

WCF 関連の構成は、[1]の<system.serviceModel>要素ブロックに記述します。特に、サービス クラスに関する記述は、[2]の<services>要素ブロックに記述します。

そして、1 つ分のサービスクラスに関する記述は、[3]の<service>ブロックの中に記述します。その際、name 属性には該当するサービス クラスの型の完全修飾名を指定します。なお、サービス クラスの定義には、次の例 4 のように ServiceBehavior 属性に ConfigurationName プロパティを用いて、構成ファイルからクラスを参照できる一種のエイリアスを付けることもできます。

例 4. 参考:構成ファイルから参照できるエイリアスをサービス クラスに付ける

```
    [ServiceBehavior(ConfigurationName="MyService2")]
```

```
    public class MileageBasicInfoService
        : IMBasicInfo
    {
```

この例 4 のサービスクラスの場合であれば、構成ファイル側では、例 3 の[3]の name 属性に「name="MyService"」と記述して、このエイリアス介してサービスクラスを特定できます。

また、例 3 の[4]に記述された behaviorConfiguration 属性は必須でありませんが、サービスに様々な構成を付加する際に用います。この[4]では「MyBehaviors」という名前の構成を参照しており、この構成が[11]に定義されています。ここでは、[12]の構成情報が適用されます。[12]のように、<serviceMetadata>要素の httpGetEnable 属性が true に設定されると、サービスのメタデータである WSDL が、HTTP GET を介して提供できるようになります。

例 3 の[5]の baseAddress 属性は、このサービスのルートアドレスです。

そして、エンドポイントの定義が[6]にあります。ここでは[7]から[9]に、エンドポイントの「ABC」が定義されています。[7]では、コントラクトとして今回定義した IMBasicInfo インターフェイスの型名が定義されています。[8]では、バインディングとして、基本的な HTTP 要求/応答を使用して、SOAP メッセージのやり取りをする「basicHttpBinding」が指定されています。また、[9]のアドレスは、[5]のルートアドレスに対する相対アドレスです。よって、実際のサービスのアドレスは、次のようになります。

```
http://localhost:8091/AdwkMService/Sample/BasicInfo
```

このほか[10]には、もう 1 つエンドポイントが定義されています。このように、1 つのサービスクラスに対して、複数のエンドポイントが公開できます。[10]のエンドポイトンは必須でありませんが、特別なエンドポイントであり、WS-MetaDataExchange を介して、サービスのメタデータを公開するための構成です。この[10]のエンドポイントの構成では、コントラクトとバインディングの設定値は、この例に記載された名前（それぞれ、IMetadataExchange、mexHttpBinding）にする必要があります。

・サービス ホストの実装

今回使用している AdwkMService プロジェクトは、コンソール アプリケーション形式のホスト アプリケーションとして、既に Main メソッドの中に実装されています。ここで新規に作成したサービスもホスティングできるように、Program クラスの Main メソッドを次のように修正してください。

例 5. サービスのホスティング

Program.cs（AdwkMService プロジェクト）

```
    // マイレージ管理サービス ホスト
    class Program
    {
        static void Main(string[] args)
        {

            var srv0 = new ServiceHost(typeof(MileageBasicInfoService));  //←[1]
            var srv1 = new ServiceHost(typeof(MileageQueryService));
            var srv2 = new ServiceHost(typeof(MileageUpdateService));
            var srv3 = new ServiceHost(typeof(MileageTransferService));

            srv0.Open();  //←[2]
            srv1.Open();
```

```
            srv2.Open();
            srv3.Open();

            ShowService(srv0);  //←[3]
            ShowService(srv1);
            ShowService(srv2);
            ShowService(srv3);

            Console.WriteLine("終了には、何かキーを押してください。");
            Console.ReadKey();  //←[4]

            srv0.Close();  //←[5]
            srv1.Close();
            srv2.Close();
            srv3.Close();
        }

        static void ShowService(ServiceHost srv)  //←[6]
        {
            foreach (var endpoint in srv.Description.Endpoints)  //←[7]
            {
                Console.WriteLine("    Endpoint:{0}", endpoint.Address);
            }
            Console.WriteLine();
        }
```

サービスのホスティング環境を構築するには、[1]のように、ServiceHost クラスのインスタンスを使用します。コンストラクターの引数には、サービス クラス（ここでは MileageBasicInfoService）の型情報を渡します。

そして、[2]の Open メソッドを呼び出すと、サービスが起動した状態になり、このサービスがクライアントからの要求を受け付けることができるようになります。トランスポート プロトコルの観点で言うと、通信チャネルを使用して、リッスン（Listen）するようになります。そして、このホスト アプリケーションが終了しない限り、[5]の Close メソッドが呼び出されるまで、サービスは要求を受け付けることができます。

逆に言えば、Close メソッドを明示的に呼び出さなくとも、アプリケーションが終了するとサービスも終了してしまうので、アプリケーションは実行を続ける必要があります。ここでは簡単にするため、[4]のようにキー入力待ち状態にして、アプリケーションの実行状態を維持しています。

また、[4]のようにすることは、アプリケーションの Main メソッドのスレッドは待機した状態であるので、WCF サービスは別のスレッドで並行実行されるという点も読み取れるでしょう（サービスにおける並行処理の制御については、第 3 章で取り上げます）。

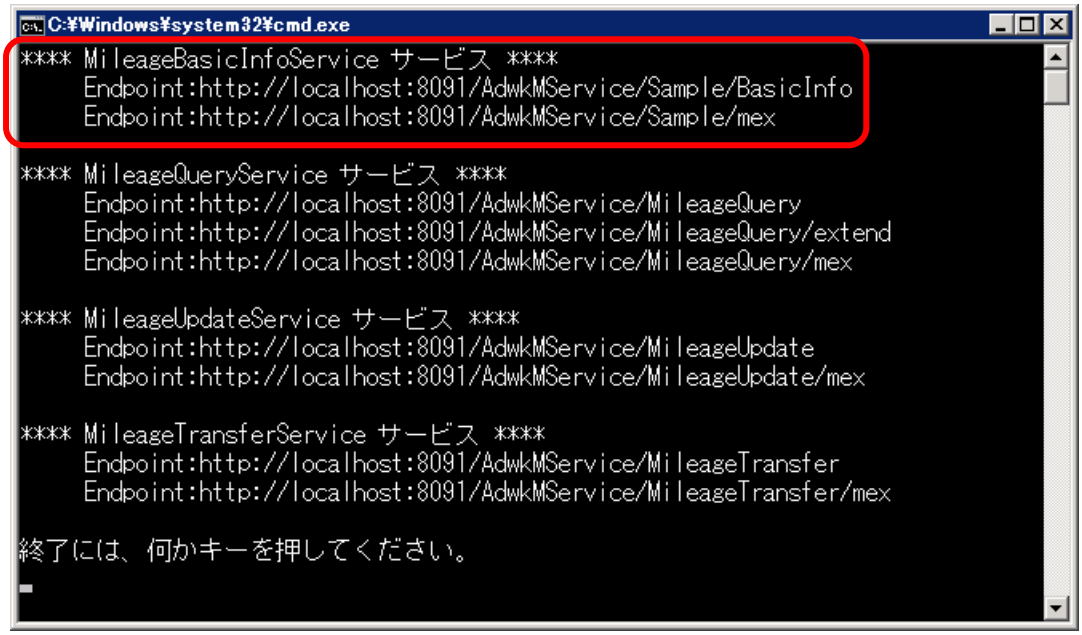
なお、1 つのサービス クラスのインスタンスのライフタイムは、リッスンの状態にあることとは直接関係はなく、既定では、セッション単位で 1 つのインスタンスが作成されます。このほか、インスタンスのライフタイムを 1 回の呼び出しごとに設定したり、セッション数に関わりなくインスタンスを 1 つにしたりすることもできます（この設定方法についても第 3 章で取り上げます）。

[3]から呼び出している[6]の ShowService メソッドは、参考までに提示したものであり必須ではありません。この ShowService メソッドでは、引数として ServiceHost インスタンスを用いて、[7]のよう

にサービスのエンドポイントの情報（srv.Description.Endpoints コレクション）をコンソールに表示しています。

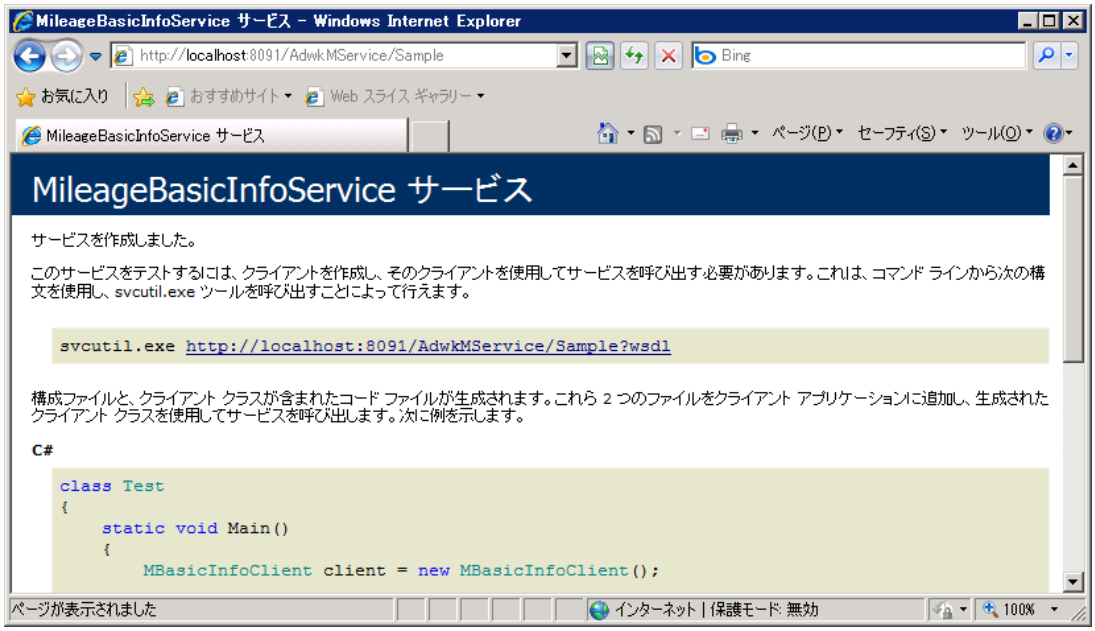
これで基本機能を備えたサービスが出来上がりました。この時点で AdwkMService プロジェクトを実行すると、ホスト アプリケーションが起動したのち、例 5 の[3]にある ShowService メソッドの呼び出しによって、今回作成したサービスのエンドポイントが表示されます。

図 10. ホスト アプリケーションの実行



また、既定の構成では、ブラウザーなどからサービスのベースアドレス（例 3 の[5]）に HTTP GET を使用してアクセスするすと、次図のように、そのサービスについて説明する HTML 形式のページ（HTML ヘルプ ページ）が開きます。（このページの表示有無の制御は、節 2.3 で取り上げます。）

図 11. サービスの HTML ヘルプ ページ



この HTML ヘルプ ページには、メタデータ（WSDL）を取得するアドレスや、クライアントのコーディング方法など、サービス関する情報が表示されます。

2.2 基本的な WCF クライアントの実装

次に、前節で作成した WCF サービスにアクセスする WCF クライアントを作成します。第 1 部の「2.2 WCF（基本部分）の実装概要と開発環境」でも触れたように、クライアントの実装方法には複数の選択肢がありますが（第 1 部の表 2 参照）、ここでは Visual Studio を用いた典型的な実装方法である「サービス参照の追加」を行います。

まず実験用として、新規に Windows フォーム アプリケーションを作成することにします。まずは、以下のプロジェクト（C#版）を新規作成してください（[ファイル]メニューの [新規作成]、[プロジェクト]）。

・使用するプロジェクト テンプレート: Windows フォーム アプリケーション

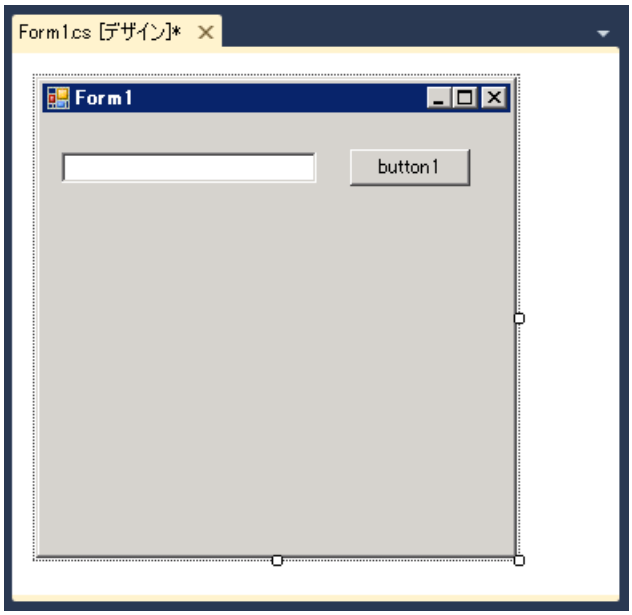
・プロジェクト名: TestClient1

・場所: （任意）

・ソリューション名: TestClient1Sol

また、フォーム Form1.cs には、テキストボックスとボタンを 1 つずつ貼り付けておいてください。それぞれのコントロールの名前は既定のままで構いません。

図 12. クライアントで使用するフォーム



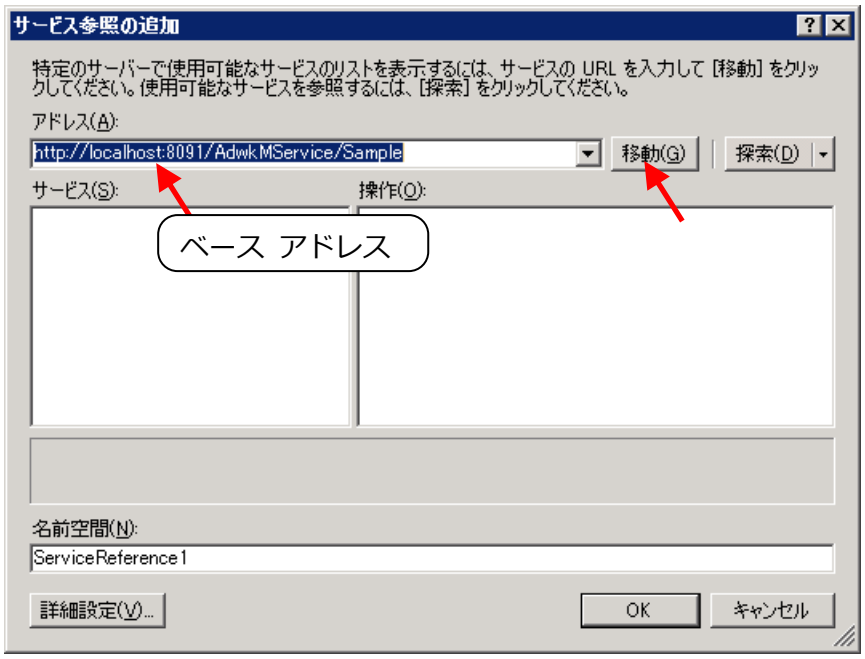
・サービス参照の追加

ここで、サービスにアクセスするためのプロキシを作成するために、「サービス参照の追加」を行います。

1. 予め、AdwkMService プロジェクトを実行して、前節で作成したサービスにアクセスできる状態にしておきます。

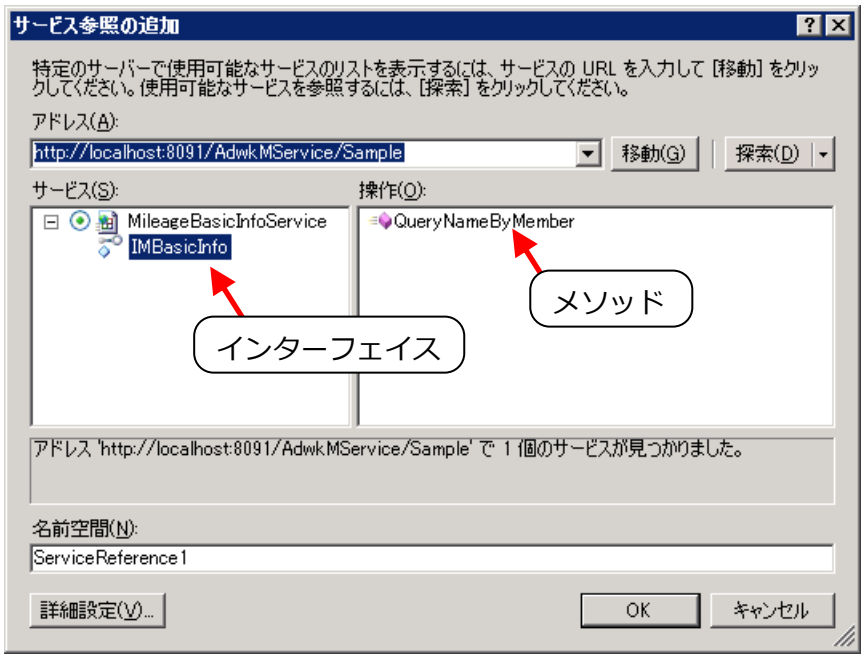
2. ソリューション エクスプローラーのツリー上で、TestClient プロジェクト ノードを右クリックして、ショートカット メニューから [サービス参照の追加] をクリックします。
3. [サービス参照の追加] ダイアログボックスが表示されたら、図 13 のように、サービスのベース アドレス（例 3 の[5]）を入力して、[検索] ボタンをクリックします。

図 13. [サービス参照の追加] ダイアログボックスでベースアドレスを指定



4. すると、メタデータがサービスから取得され、図 14 のように、左のツリーのルートノードにはサービス名が現れ、それを展開するとインターフェイス名が表示され、さらに、インターフェイスをクリックして選択すると、右の一覧には、メソッド名が表示されます。

図 14. メタデータを取得

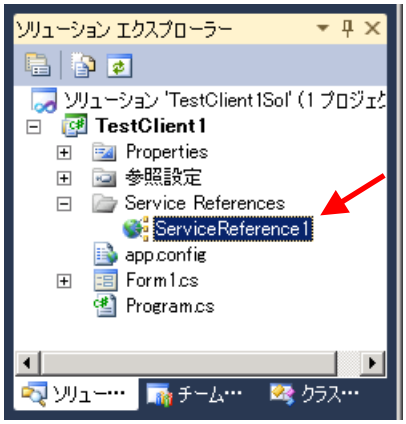


今回の構成の場合、実際のメタデータ（WSDL）を取得するアドレスは、図 11 の HTML ヘルプページにも表示されたように、ベースアドレスにクエリ文字列「?wsdl」を付けたものですが、前述のダイアログボックスでは、適切にメタデータのアドレスを見つけて取得することができます。

なお、図 14 のダイアログボックスの下部にある「名前空間」の欄は、プロキシ クラスの名前空間に反映されますが、ここでは既定のまま（ServiceReference1）にしておきます。

5. [サービス参照の追加] ダイアログボックスで、[OK] ボタンをクリックして確定し、プロキシを生成させます。
6. ソリューション エクスプローラーには、図 15 のように、サービス参照が追加されたことを表すノード（ServiceReference1）が表示されることを確認します。

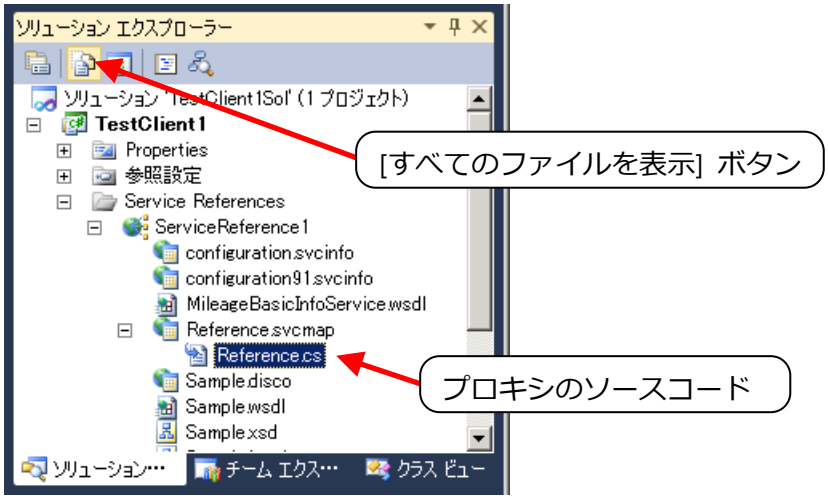
図 15. サービス参照のノードが追加される



・「サービス参照の追加」の成果物

ソリューション エクスプローラーで、 [すべてのファイルを表示] ボタン（図 16）をクリックすると、[ServiceReference1] ノードに配下に、さらに詳細な情報を表示することができます。その中のノードを展開すると、ファイル「Reference.cs」が生成されていることが分かります。このファイルには、プロキシのソースコードが含まれています。

図 16. 「サービス参照の追加」によって生成されたプロキシ



このソリューション エクスプローラー上の「Reference.cs」ダブルクリックして、ソースファイルを開くと、次のように、サービス コントラクトにあたる IMBasicInfo インターフェイスと、プロキシクラスにあたる MBasicInfoClient クラスがあることが分かります。これらが属する名前空間には、図 14 で指定したものが反映され、「プロジェクト名 + ServiceReference1」となっています。（分かりやすくするため、ポイントのみ抜粋しています。また、紙面の都合で一部の長い行は改行しています。）

例 6. 生成されたプロキシ クラス

```
namespace TestClient1.ServiceReference1 {

   public interface IMBasicInfo {  //←[1]

        [System.ServiceModel.OperationContractAttribute(
            Action="http://Adwk.Mileage/IMBasicInfo/QueryNameByMember",
            ReplyAction=
                "http://Adwk.Mileage/IMBasicInfo/QueryNameByMemberResponse")]
        string QueryNameByMember(string memberCode);
    }

    public partial class MBasicInfoClient  //←[2]
        : System.ServiceModel.ClientBase<  //←[3]
          TestClient1.ServiceReference1.IMBasicInfo>,
          TestClient1.ServiceReference1.IMBasicInfo  //←[4]
    {

        (略)

        public string QueryNameByMember(string memberCode) {  //←[5]
            return base.Channel.QueryNameByMember(memberCode);
        }
    }
}
```

[1]の IMBasicInfo インターフェイスは、例 1 のサービス側で定義したコントラクトと同じ構成になっています。また、[2]のプロキシクラスは、[3]にあるように ClientBase<T>クラスから派生して WCF クライアントとしての基本的な機能を備えています。同時に、[4]のようにサービスコントラクトである IMBasicInfo インターフェイスを実装しており、クライアント アプリケーション側では[5]の QueryNameByMember メソッドを呼び出せば、サービス側の QueryNameByMember メソッドを呼び出すことができます。

> **註:** サービス参照の追加を行った後、サービス側が変更されたなどの都合でプロキシを変更する必要がある場合は、図 15 の [ServiceReference1] ノードを右クリックし、ショートカット メニューから [サービス参照の更新] をクリックすれば、サービス側からメタデータを再度取得して、プロキシを更新することができます。もしくは、同じショートカット メニューの [サービス参照の構成] をクリックすると、 [サービス参照設定] ダイアログボックスが表示されるので、アドレスの変更ほか、プロキシに関するより細かい調整を行うことができます。

また、クライアント側にも必要な構成が自動生成されます。TestClient1 プロジェクトの app.config ファイルには、次のように、クライアント側からアクセスする際のエンドポイントが構成されます。

例 7. クライアント側のエンドポイントの構成

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<configuration>
    <system.serviceModel>

    (略)

        <client>  ←[1]
            <endpoint
                address="http://localhost:8091/AdwkMService/Sample/BasicInfo"
                binding="basicHttpBinding"
                bindingConfiguration="BasicHttpBinding_IMBasicInfo"
                contract="ServiceReference1.IMBasicInfo"
                name="BasicHttpBinding_IMBasicInfo" />
        </client>
    </system.serviceModel>
</configuration>
```

サービス側のエンドポイントの構成とは異なり、エンドポイントの記述は<service>要素ブロック内に記述するのではなく、[1]のように<client>要素ブロックの中に記述します。しかし、エンドポイントの基本的な構成は同様です。この例でも、アドレス（address）、バインディング（binding）、および、コントラクト（contract）がエンドポイント（<endpoint>要素ブロック）に記述されています。

・プロキシの利用

次に、前述のプロキシを使用して、WCF サービスにアクセスしてみましょう。フォーム Form1 のボタン（button1）の Click イベント ハンドラーを生成し、次のようにコードを入力してください。

例 8. プロキシを使用した WCF サービスへのアクセス

Form1.cs（TestClient1 プロジェクト）

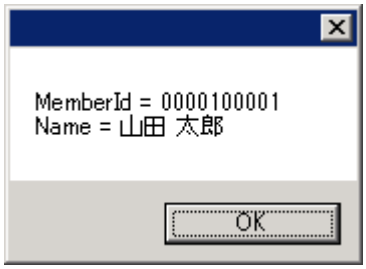
```
  (略)

using TestClient1.ServiceReference1;  //←[1]

  (略)

        private void button1_Click(object sender, EventArgs e)
        {
            var proxy = new MBasicInfoClient();  //←[2]
            string result = proxy.QueryNameByMember(textBox1.Text);  //←[3]
            MessageBox.Show(
                "MemberId = " + textBox1.Text + "¥n" +
                "Name = " + result);
            proxy.Close();  //←[4]
        }
```

既に触れたように、プロキシのソメッドを呼び出せはよいので、まず[2]のように、プロキシ クラス MBasicInfoClient のインスタンスを作成します。このプロキシ クラスは、[1]の名前空間に属しています。そして、[3]で QueryNameByMember メソッドを呼び出します。ここでは、テキストボックス（textBox1）に入力した会員コードを引数として受け取り、戻り値として、該当する会員の姓名を返します。その次の行では、メッセージボックスに実行結果を表示しています。

最後にプロキシが使用したリソースを解放するために、[4]のように、Close メソッドを呼び出します。セッションが有効なバインディングを使用していた場合、Close メソッドの呼び出しによって、セッションも終了します。

ここで、実際に動作を確認してみましょう。この TestClient1 プロジェクトをビルドして実行し、表示されたフォームのテキストボックスに、会員コードとして「0000100001」と入力します。[button1] ボタンをクリックすると、例 8 の Click イベント ハンドラーが実行され、図 17 のように、メッセージボックスには、会員コードとその会員の姓名が表示されます。

図 17. サービスの実行結果



2.3 基本的な構成要素の実装バリエーション

この節では、前節までに作成した基本的なサービスやクライアント、また、AWTS サンプル プログラムを使用して、WCF サービスの基本的な各構成要素について、そのバリエーションを紹介します（手順が明示されていない限り、特に操作やコード入力を行わなくとも構いません）。主な項目としては、HTML ヘルプ ページの表示制御、コントラクトにおけるメソッドの出力引数の扱い方やメソッドのオーバーロード、また、派生インターフェイスを用いた際のサービスの実装方法について取り上げます。

・HTML ヘルプ ページの表示制御

場合によっては、WCF サービスの HTML ヘルプページを閲覧されたくないこともあります。WCF サービスの構成を次のように変更することで、HTML ヘルプページを表示しないように設定できます。

例 9. HTML ヘルプ ページを表示しないようにする

app.config（AdwkMService プロジェクト）

```
    <behaviors>
      <serviceBehaviors>
        <behavior name="MyBehaviors" >
          <serviceDebug httpHelpPageEnabled="false" />  ←[1]
          <serviceMetadata  httpGetEnabled="false" />   ←[2]
        </behavior>
      </serviceBehaviors>
    </behaviors>
```

例 3 の[11]の<behavior>要素ブロックの中に、この例 9 の[1]のように<serviceDebug>要素を追加して、httpHelpPageEnabled 属性を false に設定します。これで、HTML ヘルプ ページ自体は表示されなくなります。ただし、この[1]の httpHelpPageEnabled 属性が false であっても、[2]の httpGetEnabled 属性が true であると、ベースアドレスに対して HTTP GET を使用すれば、WSDL 自体が提供されます。ベースアドレスから WSDL も提供したくない場合、[2]も false に設定します。

また、これ以外にも例 3 の[10]のエンドポイントの構成によって、WS-MetaDataExchange によるメタデータの提供はできるので、これも無効にする場合は、例 3 の[10]のエンドポイントも削除します。

・メソッドの出力引数

サービス コントラクトのメソッドでは、出力引数（out パラメーター）も使用することができます。たとえば、AdwkMService プロジェクトの IMileageQuery インターフェイスでは、次のように、2 番目の引数が出力引数になっています。（ソースコードに記載されたコメントの一部は省略しています。）

例 10. 出力引数を伴うサービス コントラクト

IMileageQuery.cs（AdwkMService プロジェクト）

```
[ServiceContract(Namespace="http://Adwk.Mileage")]
public interface IMileageQuery
{
    [OperationContract]
    bool QueryInfo1ByMember(
        string memberCode,       // [in]  会員コード
        out MemberInfo minfo);   // [out] 会員情報
}
```

ただし、この出力引数を伴うサービスについて、サービス参照の追加を行うと、クライアント側のプロキシでは、サービスと引数の順番が異なる点に注意してください。たとえば、AdwkWebApp ソリューションの AdwkWebApp プロジェクトに含まれるサービス参照「ServiceReference1」では、Reference.cs に記述されたプロキシクラスにおいて、上記の例 10 のメソッドの引数リストが、例 11 の[1]のメソッドのようになっています。

例 11. プロキシでの引数の並び

Reference.cs（AdwkWebApp プロジェクトの ServiceReference1）

```
public partial class MileageQueryClient
    : System.ServiceModel.ClientBase<
        Adwk.WebApp.ServiceReference1.IMileageQuery>,
      Adwk.WebApp.ServiceReference1.IMileageQuery
{

    (略)

    public bool QueryInfo1ByMember(  //←[1]
        out Adwk.WebApp.ServiceReference1.MemberInfo minfo,
        string memberCode)
    {
        return base.Channel.QueryInfo1ByMember(out minfo, memberCode);
    }
}
```

この例 11 の[1]では、本来は 2 番目であった出力引数 minfo が 1 番目の引数となっています。

こうなるのは理由があります。そもそも WSDL では、やり取りするデータを、入力データと出力データのグループに分けてそれぞれ記述しています。入力データ（入力引数）や出力データ（出力引数）が、それぞれ複数ある場合、それぞれのグループの中での順番を表現できますが、入力データと出力データとの間での相互の順番を記述できません。そのため、サービス参照の追加では、必ずしも、引数の順番が元のコントラクトと同じになるとは限らないのです。このように、入力引数と出力引数がある場合は、出力引数が先に並びます。（ref パラメーターも伴う場合は、若干異なります。）

いずれにしてもクライアント側では、実際に生成されたプロキシのメソッド定義に合わせて、引数を順番に渡す必要があります。

> **註:** そもそも、クライアントがサービスに関して知るべき部分は、コントラクトだけなので、クライアントはコントラクトに従っていればよく、実際の引数の順番がサービス側と一致していなくとも問題ありません。しかし、引数の順番をサービス側と合わせたいのならば、チャネル ファクトリを利用する方法もあります。第 4 章で取り上げるチャネル ファクトリを使用する方法では、サービス側のサービス コントラクトやデータ コントラクトのアセンブリをクライアントが参照して、そのまま利用できるので、もともとの引数の順番でデータを渡すことができます。

・メソッドのオーバーロード

一般に、機能が同じ複数のメソッドを実装し、それらの渡すべき引数のパターンが異なる状況では、メソッドのオーバーロードと称して、引数のパターンだけを変えて、名前が同じ複数のメソッドを定義します。.NET Framework のインターフェイスも、メソッドのオーバーロードを行うことができます。

しかし、WSDL ではメソッドのオーバーロードをサポートしていないので、WCF サービスもオーバーロードされたメソッドをそのまま公開することはできません。オーバーロードされたメソッドを持つインターフェイスを、WCF サービスのコントラクトとして使用するには、公開されるメソッドの名前を明示的に変更する必要があります。

たとえば、AdwkRService プロジェクトの IRBasicInfo インターフェイスでは、次のように定義されています（一部、コメントを省略しています）。

例 12. インターフェイスのメソッドのオーバーロード

IRBasicInfo.cs（AdwkRService プロジェクト）

```
// 予約管理サービスの基本情報を返すサービス
[ServiceContract(Namespace="http://Adwk.Reservation")]
public interface IRBasicInfo
{
    [OperationContract(Name="GetBasicRouteInfo1")]    //←[1]
    bool GetBasicRouteInfo(                           //←[2]
        out IEnumerable<場所> places, out IEnumerable<クラス> levels);

    [OperationContract(Name="GetBasicRouteInfo2")]    //←[3]
    bool GetBasicRouteInfo(                           //←[4]
        out 場所 [] places, out クラス [] levels);

    [OperationContract(Name="GetBasicRouteInfo3")]    //←[5]
    bool GetBasicRouteInfo(                           //←[6]
        out PlaceInfo[] places, out LevelInfo[] levels);
}
```

[2]、[4]、および [6]には、同じ名前の GetBasicRouteInfo メソッドが定義されています。このままでは、サービス コントラクトとして使用すると実行時エラーになります。これを回避するには、メソッドの名前自体を変更する方法もありますが、このインターフェイスを他で流用するなどの理由から、オーバーロードをそのまま維持したい場合もあります。その場合は、[1]のように、OperationContract 属性の Name プロパティを利用して、重複しない名前を明示的に付けて公開すれば、このインターフェイスをそのまま利用することができます。この例では、WCF サービスとして公開されるメソッドの名前は、順に、GetBasicRouteInfo1、GetBasicRouteInfo2、GetBasicRouteInfo3 となります。

この状態でサービスを起動し、サービス参照の追加を行って、クライアント側のコントラクトとプロキシを生成すると、そのメソッド名は公開された名前になります。たとえば、AdwkWpfApp ソリューションの AdwkWpfApp プロジェクトに含まれるサービス参照「ServiceReference1」では、Reference.cs の中のコントラクトにおいて、例 12 で指定した名前が、次のように反映されています。（分かりやすくするため、属性を省略しています。）

例 13. クライアント側では異なる名前のメソッドとして生成される

Reference.cs（AdwkWebApp プロジェクトの ServiceReference1）

```
public interface IRBasicInfo {
    bool GetBasicRouteInfo1(
        out Adwk.WpfApp.ServiceReference1.場所[] places,
        out Adwk.WpfApp.ServiceReference1.クラス[] levels);

    bool GetBasicRouteInfo2(
        out Adwk.WpfApp.ServiceReference1.場所[] places,
        out Adwk.WpfApp.ServiceReference1.クラス[] levels);

    bool GetBasicRouteInfo3(
        out Adwk.WpfApp.ServiceReference1.PlaceInfo[] places,
        out Adwk.WpfApp.ServiceReference1.LevelInfo[] levels);
}
```

・派生インターフェイスの利用

インターフェイスは、基本インターフェイスから継承して、派生インターフェイスを定義することができます。この方法を用いれば、よく利用される共通のメソッドを基本インターフェイスに定義しておき、それを再利用する形で派生インターフェイスを定義することで開発効率が上がります。また、複数のインターフェイスを共通の基本インターフェイスから継承することで、一貫性ある操作を提供できるようになります。この仕組みは、WCF サービスでも利用できます。

AdwkMService プロジェクトでは、次のように基本インターフェイス IMileageQuery と、その派生インターフェイス IMileageQuery2 が定義されています。（一部のコメントを割愛しています。）

例 14. 基本インターフェイスと派生インターフェイス

IMileageQuery.cs（AdwkMService プロジェクト）

```
[ServiceContract(Namespace="http://Adwk.Mileage")]
public interface IMileageQuery
{
    [OperationContract]
    bool QueryInfo1ByMember(
        string memberCode,       // [in]  会員コード
        out MemberInfo minfo);   // [out] 会員情報
}
```

IMileageQuery2.cs（AdwkMService プロジェクト）

```
[ServiceContract(Namespace="http://Adwk.Mileage")]
public interface IMileageQuery2
    : IMileageQuery
{
    [OperationContract]
    AdwkMileagesDataSet.会員マイル履歴DataTable QueryHistoryByMemberAndPeriod(
        string memberCode,        // [in] 会員コード
        DateTime fromDate,        // [in] 期間開始日
```

```
            DateTime untilDate,        // [in] 期間終了日(の翌日0時)
            out AdwkMStatus stat,      // [out] ステータス
            out string systemMessage); // [out] システムメッセージ
    }
}
```

前者の IMileageQuery インターフェイスは、基本的な会員情報を返す QueryInfo1ByMember メソッドがあり、このサンプルでは、このインターフェイスが社内システムと社外システムの両方で利用されています。

一方、後者の IMileageQuery2 インターフェイスは、IMileageQuery から継承しており、社内専用の型指定されたデータテーブル「会員マイル履歴 DataTable」を返すメソッドとして、QueryHistoryByMemberAndPeriod が追加されています。このサンプルでは、IMileageQuery2 インターフェイスは社内向けに作られたものですが、IMileageQuery インターフェイスを継承しているので、社内と社外に共通する QueryInfo1ByMember メソッドも利用することができます。

> **註:** WCF サービスにおける型指定されたデータテーブルの扱いについては、第 4 章で取り上げます。

実際にサービス クラスを実装するときは、派生インターフェイスのほうを実装します。そうすれば、基本インターフェイスの実装も提供できるようになります。AdwkMService プロジェクトでは、前述の派生インターフェイスを用いて、サービス クラスが次のように定義されています。

例 15. 派生インターフェイスを実装したサービス クラス

MileageQueryService.cs（AdwkMService プロジェクト）

```
    // マイル照会関連のサービス
    public class MileageQueryService
        : IMileageQuery2                //←[1]
    {
        // 特定の会員に関して、会員情報を返す
        bool IMileageQuery.QueryInfo1ByMember(    //←[2]
            string memberCode, out MemberInfo minfo)
        {

            (略)

            return result;
        }

        // 特定の会員に関して、会員のマイル履歴を返す
        AdwkMileagesDataSet.会員マイル履歴DataTable
            IMileageQuery2.QueryHistoryByMemberAndPeriod(    //←[3]
            string memberCode, DateTime fromDate, DateTime untilDate,
            out AdwkMStatus stat, out string systemMessage)
        {

            (略)

            return table;  //エラーの場合はデータ0件のDataTable
        }
    }
```

ここでは、[1]のように IMileageQuery2 インターフェイスを実装するように定義し、このインターフェイスが持つ 2 つのメソッド（[2]と[3]）が記述されています。このうち、[2]のメソッドは基本インターフェイスである IMileageQuery のメソッドでもあるので、このサービスクラスは IMileageQuery インターフェイスを使用してアクセスすることもできます。

そして、次のように構成ファイルには、基本インターフェイスと派生インターフェイスのそれぞれに、異なるエンドポイントを公開することができます。このサンプルでは、基本インターフェイスのエンドポイントは、社外向けと社内向けに利用されることを意図しており、派生インターフェイスのエンドポイントは、社内向けに利用されます。

例 16. 基本インターフェイスと派生インターフェイスのエンドポイント

App.config（AdwkMService プロジェクト）

```
<!-- マイル照会サービス -->
<service name="Adwk.Mileage.Service.MileageQueryService"   ←[1]
        behaviorConfiguration="MyBehaviors">
  <host>
    <baseAddresses>
      <add baseAddress="http://localhost:8091/AdwkMService/MileageQuery" />
    </baseAddresses>
  </host>
  <endpoint
    contract="Adwk.Mileage.IMileageQuery" ←[2]
    binding="wsHttpBinding"
    address="" />
  <endpoint
    contract="Adwk.Mileage.IMileageQuery2" ←[3]
    binding="basicHttpBinding"
    address="extend" />
  <endpoint
    contract="IMetadataExchange"
    binding="mexHttpBinding"
    address="mex" />
</service>
```

この例では、[1]の 1 つのサービス クラス MileageQueryService の構成に関して、[2]および[3]において、異なるコントラクトを使用して、2 つのエンドポイントが公開されています。

2.4 メッセージ ログの基本操作

このあとの第 3 章では、様々なサービスの実装方法について取り上げますが、その前に、サービスとクライアントとの間でやり取りされるメッセージのログ（メッセージ ログ）を出力する方法を確認しておきましょう。

この方法を知っておけば、この後のサンプルを検証する中で、どのようなやり取りがサービスとクライアントとの間で行われるか確認できます。また、サービスとクライアントとの間で、期待した通りに動作しない場合、トラブルシューティングの 1 つの方法として利用できます。

既に第 1 章で触れたように、WCF には様々な管理と診断の機能があり、やり取りされるメッセージをトレースし、ログとして出力することができます。実際にログを出力させるには、特別なコーディングを追加する必要はなく、構成ファイルを使って指定できます。また、XML 形式の構成ファイルを

テキスト エディターで編集しなくとも、「WCF サービス構成エディター」を使用すると、対話形式のダイアログボックスを介して、これらの必要な構成を行うことができます。

さらに、出力されたログを様々な形式で分析表示する「サービス トレース ビューアー」も用意されています。

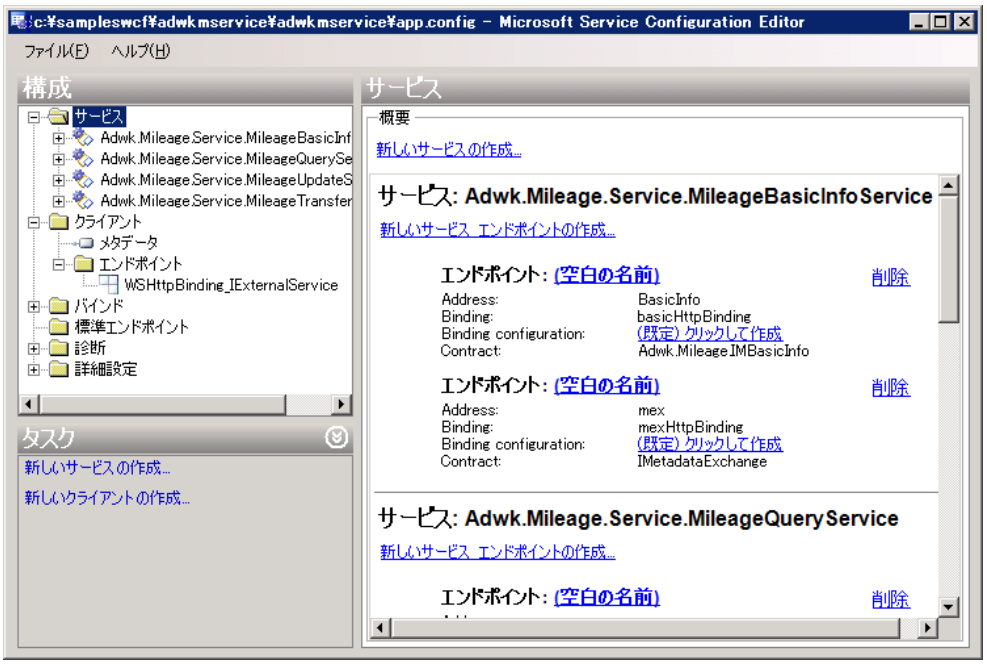
ここでは、節 2.1 で作成した簡単な WCF サービスに対して、これらのツールを使用して、メッセージ ログを出力するように設定し、そのログを確認してみます。

1. AdwkMService ソリューションの AdwkMService プロジェクトを開きます。
2. AdwkMService プロジェクト内の App.config ファイルを右クリックして、ショートカット メニューの [WCF 構成の編集] をクリックします。

註: このメニュー項目が表示されない場合は、Visual Studio の [ツール] メニューにある [WCF サービス構成エディター] をクリックして、WCF サービス構成エディターを一旦起動して終了してみてください。すると、手順 2 のショートカット メニューが表示されるようになります。

3. 次のように、WCF サービス構成エディターが起動することを確認します。

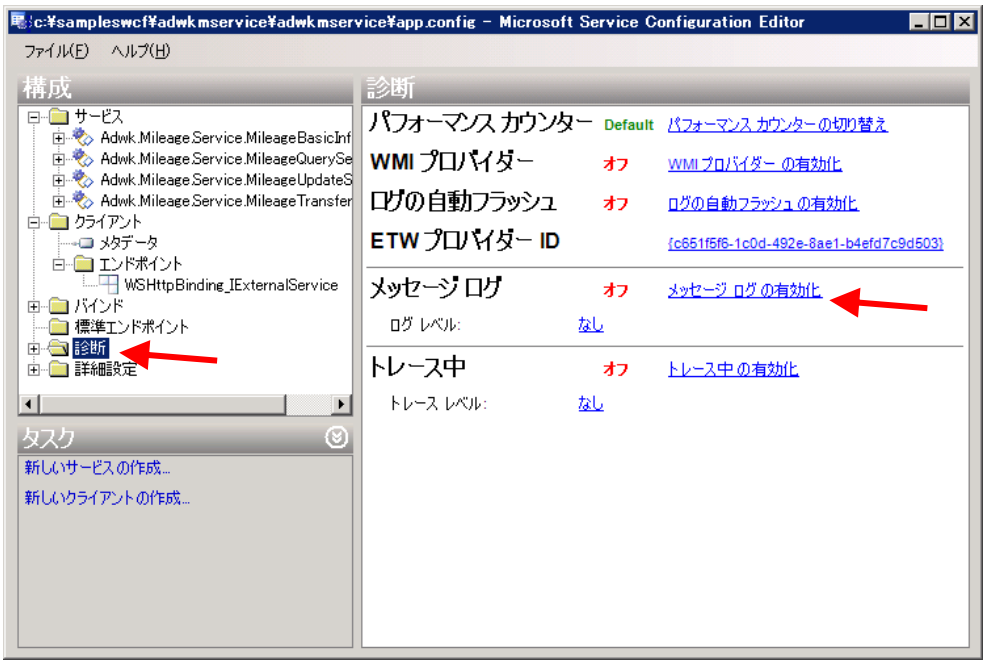
図 18. WCF サービス構成エディター



この状態は、すでに App.config ファイルが読み込まれており、この画面を通じて、App.config ファイルに対して、必要な設定を行うことができます。

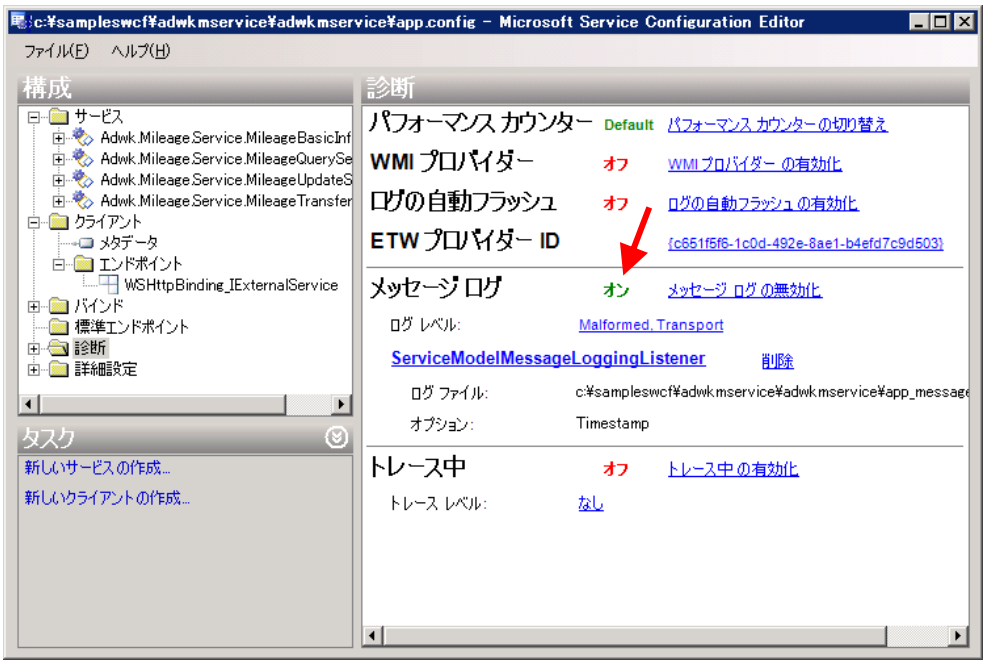
4. 左側の [構成] ペインのツリー上で、[診断] ノードをクリックします。このとき、右ペインに表示された項目中で、「メッセージ ログの有効化」をクリックして、メッセージ ログが出力できるように設定します。

図 19. メッセージ ログの有効化



5. すると、次のようにメッセージ ログが「オン」の状態になったことを確認します。

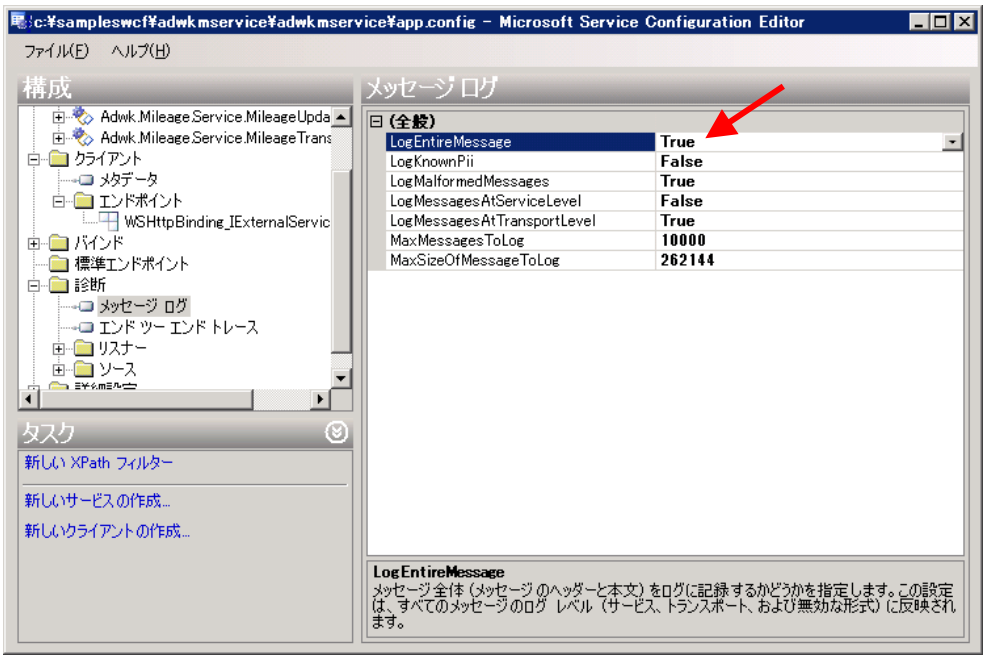
図 20. メッセージ ログがオンの状態



この時点で、メッセージ ログが出力されるようになります。ただし、既定の構成では、トランスポート レベルでの情報や不正なメッセージをログとして出力できますが、正常なメッセージの本体は出力されません。ここでは、メッセージ全体が出力できるように構成を変更することにします。

6. [構成] ペインのツリーにある [診断] ノードを展開し、[メッセージ ログ] ノードをクリックします。
7. すると、図 21 のように右側には [メッセージ ログ] ペインが表示されるので、メッセージ全体をログとして出力させるため、「LogEntireMessage」プロパティを True に設定します。

図 21. メッセージ全体をログとして出力するように設定



8. 設定が済んだら、[ファイル] メニューの [保存] をクリックして、App.config ファイルなに対して上書きしておきます。
9. WCF サービス構成エディターを終了します。

これでメッセージ全体がログとして出力できるようになったので、次にサービスを実行して、ログを確認してみましょう。

10. AdwkMService プロジェクトを実行し、ホスト アプリケーションを起動します。
11. 節 2.2 で作成した TestClient1 プロジェクトを実行します。
12. TestClient1 プロジェクトのフォームが表示されたら、テキストボックスに「0000100001」と入力して、 [button1] ボタンをクリックします。
13. 会員情報を取得する WCF サービスを呼び出し、その結果がメッセージボックスに表示されるので、閉じておきます。（この呼び出しによってメッセージ ログが作成されます。）
14. TestClient1 プロジェクトのフォームを閉じておきます。
15. AdwkMService プロジェクトのホスト アプリケーションを終了します。

註: ホスト アプリケーションを終了しないと、メッセージ ログが完全にファイルに書き込まれず、バッファーに残っている場合があります。ログ ファイルを確認する前に、ホスト アプリケーションを終了してください。

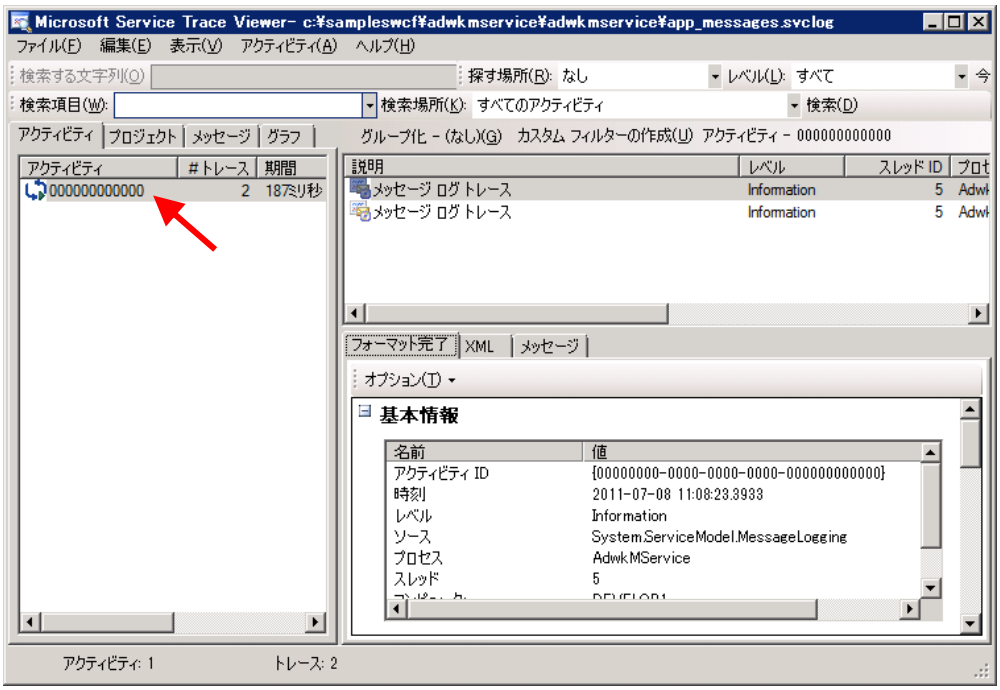
16. AdwkMService プロジェクトのフォルダーの中に、次のログ ファイルが作成されていることを確認します。（このファイルが作成される場所は、ソリューション フォルダーではなく、プロジェクト フォルダーです。）

    app_messages.svclog

17. このファイルをダブルクリックして、サービス トレース ビューアー（SvcTraceViewer.exe）で開きます。

18. サービス トレース ビューアーが起動したら、左側の [アクティビティ] タブにある 1 つ目のアクティビティをクリックして選択します。すると、右ペインには 2 つの「メッセージ ログ トレース」の行が表示されることを確認します。

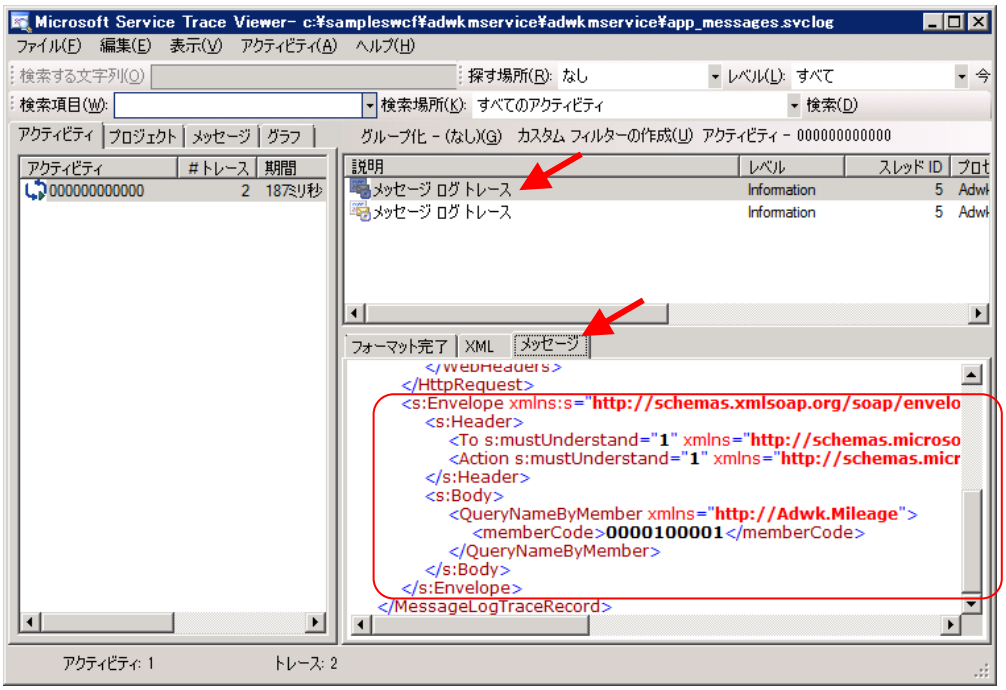
図 22. アクティビティを選択



今回の例では、WCF サービスの呼び出しを 1 回行ったので、ここに表示された 2 行は、それぞれ、要求メッセージと応答メッセージについてのメッセージ ログです。

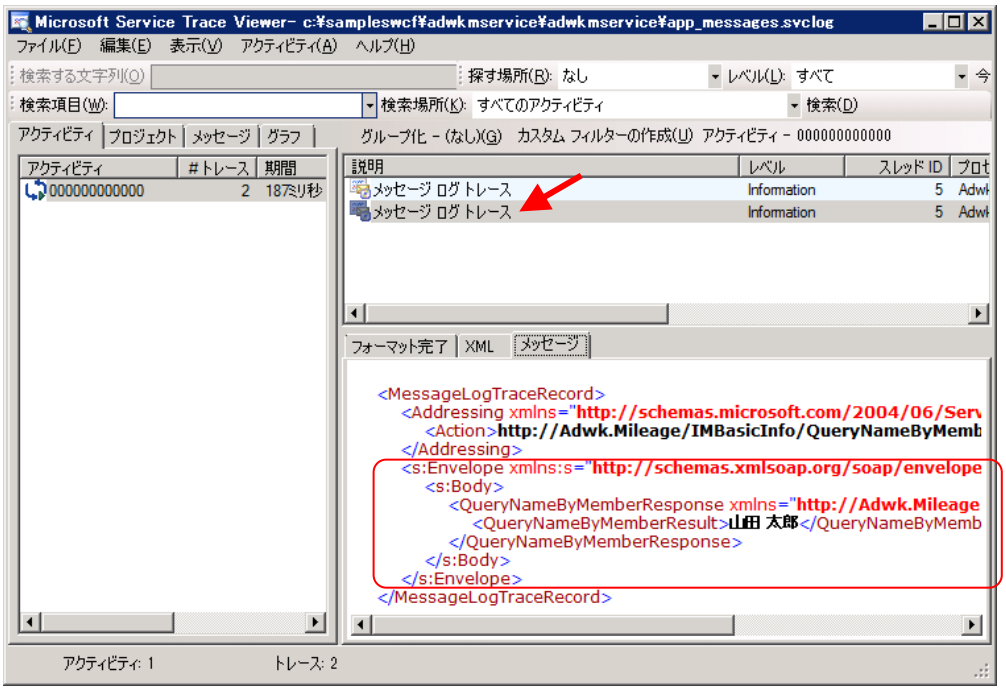
19. 右ペインで 1 行目の「メッセージ ログ トレース」をクリックし、右下半分のペインにある [メッセージ] タブをクリックします。すると次図のように、このタブには、専用の形式のメッセージ ログが表示されるので、このメッセージを下部までスクロールして、要求メッセージに当たる SOAP エンベロープが含まれていることを確認します。

図 23. メッセージ ログに含まれる要求メッセージの SOAP エンベロープ



20. 今度は 2 行目の「メッセージ ログ トレース」をクリックして選択します。すると、下部の [メッセージ] タブには、応答メッセージの SOAP エンベロープが表示されます。

図 24. メッセージ ログに含まれる応答メッセージの SOAP エンベロープ



21. 確認が済んだら、サービス トレース ビューアーを終了しておきます。

なお、これ以降においてサンプル プログラムを実行する際に、特にメッセージ ログを確認しないのであれば、図 20 の右ペイン中央にある「メッセージ ログの無効化」をクリックし、メッセージ ログを出力しないように設定しておいてください（設定後は、忘れずに保存しておいてください）。

第 3 章 サービス側の様々な実装

この章では、WCF が提供する機能を用いて、サービスを強化する様々な実装方法を取り上げます。AWTS サンプルプログラムを題材にして、サンプルで利用されている各種実装方法について確認していきます。必要に応じて、サンプルのソースコードも参照してみてください。

なお、サービスで扱うデータの種類やその操作方法については、クライアントとのやり取りが関係するので、第 4 章「クライアント側の様々な実装とデータ操作」で取り上げます。

3.1 インスタンス、同時実行、セッションの管理

WCF では、サービスを提供する言わば本体の実装はサービス クラスであり、WCF サービスの実行時には、このサービス クラスのインスタンスが作成されます。このインスタンスは、「サービス オブジェクト」、または、「サービス インスタンス」とも呼ばれています。

サービスのステートを管理する必要があるとき、このようなサービス オブジェクトのインスタンスに、ステートを保持させることもできます。その際、ステートをどう管理するかによって、インスタンスのライフタイムやセッションの有無などの制御が必要になってきます。また、一般に複数のインスタンスや複数のセッションを使用する環境では、同時に複数のクライアントから呼び出されることも考慮し、同時実行（Concurrency）に関しても制御する必要があります。

ここでは、このようなインスタンス、同時実行、セッションの制御方法について改めて取り上げます。まずは、基本的な制御方法について確認します。

・インスタンスのライフタイムの制御

サービス クラスに属性を付けることで、そのインスタンスのライフタイムを制御できます。

たとえば、AdwkRService ソリューションの AdwkRService プロジェクトのサービス クラスの 1 つである ReservationService クラスには、例 17 のように ServiceBehavior 属性が付いています。

例 17. サービス クラス インスタンスのライフタイムの制御

ReservationService.cs（AdwkRService プロジェクト）

```
[ServiceBehavior(                                        //←[1]
    InstanceContextMode=InstanceContextMode.PerSession,  //←[2]
    ConcurrencyMode=ConcurrencyMode.Single)]             //←[3]
public class ReservationService
    : IReservation
{

    (略)

}
```

[1]に記述された ServiceBehavior 属性は、サービス クラスに対して付けることができる属性の 1 つであり、[2]のように InstanceContextMode プロパティの値によって、インスタンスのライフタイムが異なります。このプロパティには InstanceContextMode 列挙型を用いて、次の値を設定することができます。

表 1. InstanceContextMode プロパティによるインスタンスのライフタイム制御

InstanceContextMode 列挙型の値	説明
InstanceContextMode.PerCall	メソッド呼び出しの都度、新しいインスタンスを作成する。
InstanceContextMode.PerSession	セッションごとに、インスタンスを作成する。
InstanceContextMode.Single	すべての呼び出しに対して、1 つのインスタンスを利用する。

前述の例 17 では、この表 1 のうち「PerSession」が指定されているので、セッションごとにインスタンスが確保されます。つまり、セッションの開始とともにインスタンスが確保され、セッションの終了とともにインスタンスは破棄されます。このインスタンスのメンバー変数にステートを格納すれば、セッション単位でステートを管理することができます（より具体的なサンプルは後述）。

このほか WCF では、この表 1 のとおり、呼び出しごとにインスタンスを作成する構成（PerCall）や、構成された 1 つのサービスに 1 つのインスタンス、つまり、ホスト アプリケーションのプロセスに 1 つだけの構成（Single）を指定することができます。

・同時実行の制御

サービス インスタンスの環境を構成する際には、前述のように、セッションや呼び出しとライフタイムとの関係を決める必要があるほか、このインスタンスに対してクライアントからの複数の要求を、どのように同時実行させるかという点も考慮する必要があります。

前述の例 17 の[3]の ConcurrencyMode プロパティは、そのような同時実行の制御を行うプロパティです。このプロパティには、次の値を指定できます。

表 2. ConcurrencyMode プロパティ

ConcurrencyMode 列挙型の値	説明
ConcurrencyMode.Single	1 つのインスタンスにつき、1 つのスレッドで実行される。1 つの要求がサービスで処理されている間は、次の要求は待機する。サービスの再帰呼び出しもできない。
ConcurrencyMode.Reentrant	1 つのインスタンスにつき、1 つのスレッドで実行される。このモードのサービスから、別のサービス、または、コールバックを介して再帰呼び出しをすることができる。
ConcurrencyMode.Multiple	1 つのインスタンスにつき、複数のスレッドを実行できる。複数の要求を同時に実行できる。

この表のとおり、ConcurrencyMode プロパティの値によって、サービスのインスタンスの実行環境が、シングルスレッドか、マルチスレッドであるかが決まります。この表 2 の 1 番目と 2 番目の選択肢は、シングルスレッドですが、2 番目の Reentrant の場合は、自身のサービスから別のサービスを介して、再帰呼び出することができます。

一般に、シングルスレッドの場合は、同一データに対する複数スレッドからの同時アクセスを配慮する必要がないので、実装が簡単になります。しかし、クライアントから一度に複数の要求が来た場合、それらの要求は単一のスレッドで順次処理されるため、応答に時間がかかる場合があります。一方、マルチスレッドの場合は、同時に複数の要求を処理できますが、複数のスレッドからのデータアクセスの同期など、スレッド間の実行のタイミングの調整など、実装が複雑になる場合もあります。

今回の例 17 の予約システムでは、1 人のクライアントが、同時に複数の要求を送ることは、まずありません。仮に 1 人のユーザーから複数の要求が来たとしても、順次処理で十分でしょう。よって、1 つのセッションにつきインスタンスが 1 つになるように構成した上で、[3]のように 1 つのインスタンスはシングルスレッドになるよう設定してあります。

・セッションの制御

前述の例 17 では、セッションごとにインスタンスを生成するように設定しましたが、セッションの有無の制御など、セッション自身の制御は、サービス コントラクトに付ける別の属性で行います。例 17 のサービス クラスのサービス コントラクトには、セッションを使用するために、次の例 18 のように属性が記述されています。

例 18. サービス コントラクトにおけるセッションの制御

IReservation.cs（AdwkRService プロジェクト）

```
    [ServiceContract(                              //←[1]
        Namespace="http://Adwk.Reservation",
        SessionMode=SessionMode.Required)]         //←[2]
    public interface IReservation
    {

        (略)

    }
```

[1]の ServiceContract 属性では、[2]の SessionMode プロパティを介して、このサービスにおけるセッションの確立を制御できます。このプロパティには、次の表 3 のように、SessionMode 列挙型の値を指定できます。

表 3. SessionMode プロパティ

SessionMode 列挙型の値	説明
SessionMode.Allowed	セッションをサポートしているバインディングを使用している場合は、セッションを確立する。
SessionMode.Required	セッションをサポートしているバインディングを使用している場合は、セッションを確立する。サポートしていないバインディングを使用すると、実行時エラーが発生する。
SessionMode.NotAllowed	セッションをサポートしているバインディングを使用している場合でも、セッションを確立しない。

1 つ注意すべき点として、この表からも読み取れるように、SessionMode プロパティだけで、セッションを確立できるわけではない点です。エンドポイントで使用するバインディングが、セッションをサポートするものでなければ、セッションを利用できません。

たとえば、バインディングの 1 つである「wsHttpBinding」は、セッションをサポートしています。このバインディングを使用するよう構成した環境で、「Allowed」または「Required」が指定されているサービス コントラクトを使用すると、セッションが確立されます。「wsHttpBinding」を用いても、「NotAllowed」が指定されたサービス コントラクトの場合には、セッションは利用されません。

一方、「basicHttpBinding」ではセッションをサポートしていません。よって、「Required」が指定されたサービス コントラクトを使用するサービスに対して、このバインディングを用いると、実行時エラーになります。

まとめると、「Required」を指定したサービス コントラクトでは、セッションの確立が強制され、「NotAllowed」を指定したサービス コントラクトでは、セッションを使用しいなことが強制されます。そして、「Allowed」の場合は、使用するバインディングがセッションをサポートできるか否かによって、セッションが確立するかどうかが決まります。

> **註:** 各種バインディングのセッションのサポート状況に関しては、以下のアドレスのページに表が掲載されています。
> http://msdn.microsoft.com/ja-jp/library/ms731092.aspx
> システムが提供するバインディングの構成

上記のとおり、サービス インスタンスや同時実行、セッションなどの制御には、複数の選択肢があり、様々な実装のバリエーションが考えられます。この節では、まとめとして後ほど、サービス インスタンスや同時実行の制御もふまえた、典型的なセッションの利用例を 1 つ確認します。

> **註:** サービス インスタンス、同時実行、および、セッションに関する情報は、以下のアドレスにも記載されています。
> http://msdn.microsoft.com/ja-jp/library/ms731193.aspx
> セッション、インスタンス化、および同時実行

・セッション単位でのステート管理の必要性と特徴

セッションやステート管理の利用例を確認する前に、まず、その必要性や特徴を改めて確認してみましょう。

クライアントが WCF サービスを利用する場合、1 回の呼び出しで処理が完結するのではなく、複数回の呼び出しで、論理的な 1 つの処理として意味をなす場合があります。たとえば、サーバー側に登録済みの予約データを更新する場合、1 回目のサービスの呼び出しで、予約データを参照して確認し、2 回目のサービスの呼び出しで、その予約データに対して更新（修正や削除）を行うような場合です。この場合、1 回目のサービス呼び出しの際に、現在照会されている予約データが何であるかという情報（ステート）を維持しておかないと、次のサービス呼び出しに処理を続けることができません。サービスでステートを管理する必要があるのは、このような連続的な呼び出しが、論理的に 1 つの処理として形成されるような場合です。一般に、このような連続的な操作は、1 つのユーザー（クライアント）が 1 つのセッションの中で行うことになるので、セッション単位でステートを管理することになります。

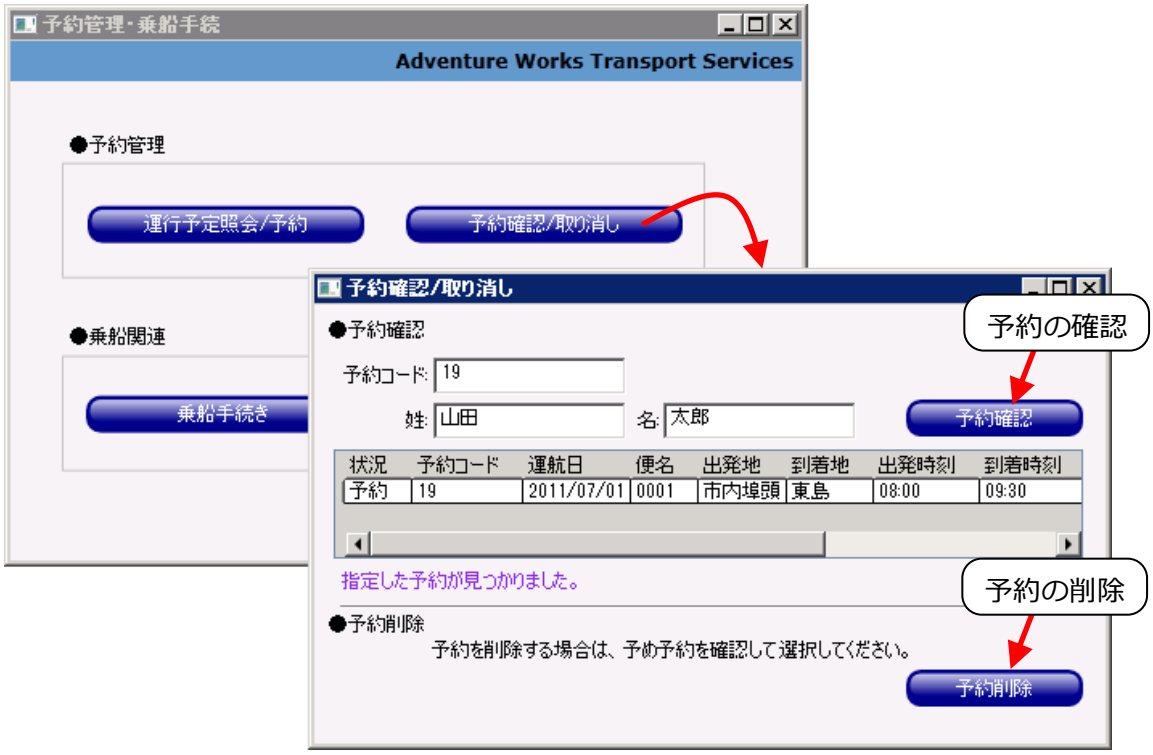
そもそも WCF 自身のアーキテクチャには、セッション単位のステートを自動的に保持する仕組みが用意されていないため、プログラマーが明示的にコーディングを行います。その典型的な実装方法としては、セッションごとにサービス インスタンスを用意し、そのインスタンスのメンバー変数などを利用して、セッションを管理する方法があります。

註: 一連のサービス呼び出しのステートを管理する方法としては、クライアント側でステートを維持する方法もあります。クライアント側でステートを管理するメリットとしては、サービス側の実装が簡単になり、サービスの負荷を分散できる点が挙げられます。逆にデメリットとして、クライアントの実装が複雑になり、ステートの内容によってはクライアントのプラットフォームが限定され、相互運用性が低下する場合があります。また、クライアントでステートを管理する場合、サービスを呼び出す都度に、最新のステートをサービスへ送らなければならない場合も多いので、ステートの構造によってはネットワークのトラフィックが増える場合もあります。ただし、WCF Data Services や WCF RIA Services が利用可能な状況では、これらを利用すると、比較的少ない工数で、クライアント側でステートを管理する実装を実現することができ、開発生産性の向上につながります。

・サンプルに見るセッション単位でのステート管理

予約管理サービス（AdwkRService プロジェクト）では、予約データの照会から削除に至る一連の処理の中で、セッション単位のステートを管理しています。クライアント側（AdwkWpfApp プロジェクト）から見た場合、次のユーザー インターフェイスを使用して、予約の確認から削除までの操作を行います。

図 25 予約の確認と削除のためのユーザー インターフェイス



註: この画面を表示するには、予約管理サービスである AdwkRService プロジェクトを実行したのち、クライアントの AdwkWpfApp プロジェクトを実行してください。（この実行の様子は、図 7 の構成に該当しますが、マイレージ管理サービス AdwkMService は不要です。）また、予約の確認や削除をするためには、あらかじめ、予約データを作成しておく必要があります。予約データの作成、および、確認と削除のより具体的な方法は、セットアップガイドの後半部分である「サンプル操作方法ウォークスルー」を参照してください。

このサンプルにおいて、セッション単位のステート管理をどのように行っているのか確認してみましょう。全体的な構成の特徴を図示すると次のようになります。

図 26 予約の確認と削除のための、セッション単位のステート管理

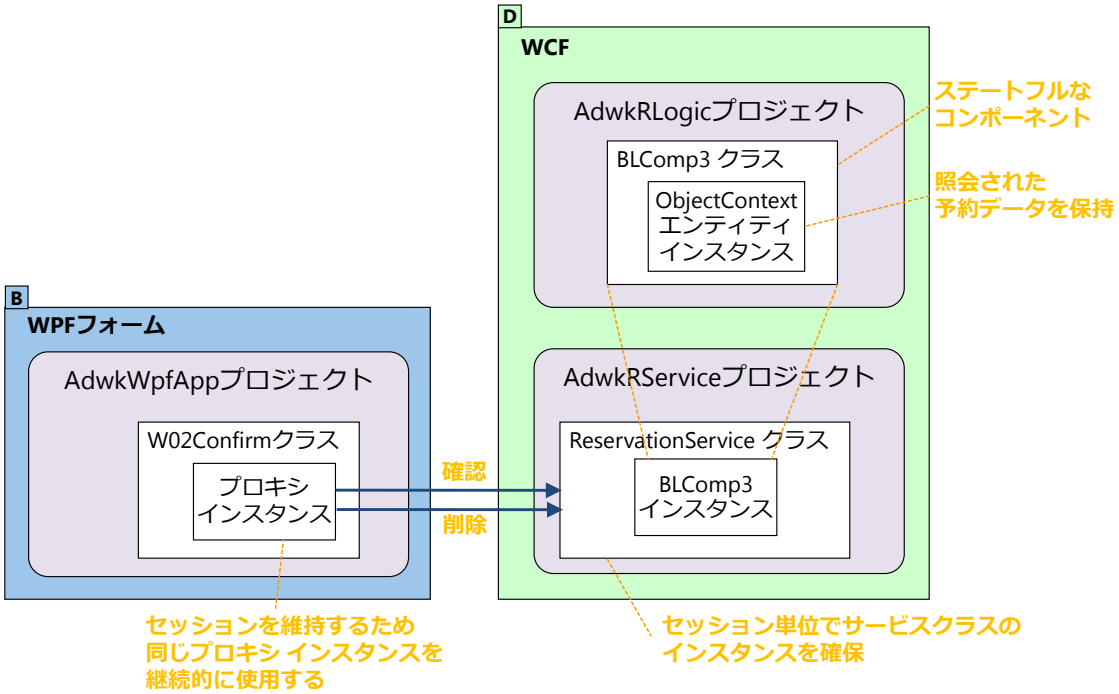


図 25 の「予約確認/取り消し」画面は、図 26 では W02Confirm クラス（図左下）に相当します。この W02Confirm クラスの実装では、プロキシを使用してサービスにアクセスしています。

この W02Confirm クラスの画面では、予約確認から予約取り消し（削除）までを一連の処理としてセッションを維持する必要があります。注意すべき点は、クライアントがサービスにアクセスする際にセッションを維持するには、同じプロキシのインスタンスを使用する必要がある点です。プロキシを使用してアクセスを開始すると、セッションが開始し、同一のプロキシの Close メソッドを呼び出すまでは、セッションが維持されます。（もちろん、前述のとおり、サービスやバインディングの構成が、セッションを利用可能なように設定されていなければなりません。）

このサンプルでも、W02Confirm クラスの実装では、このクラスの画面が開いている間は、同一のプロキシが使用できるようにするため、プロキシ用にメンバー変数を用意しています。逆に言えば、セッションが必要ない場合は、ローカル変数でも構いません。

例 19. 同一のプロキシ インスタンスを使用するため、メンバー変数として確保

W02Confirm.xaml.cs（AdwkWpfApp プロジェクト）

```
    public partial class W02Confirm : Window
    {
        private ReservationClient proxy = null;
```

また、呼び出されたサービス（図 26 の右下）である ReservationService クラスでは、例 17 のようにセッション単位にインスタンスが確保できるように構成してあります。この ReservationService クラスでステートを管理する必要がありますが、このサービスクラスは、ビジネスロジックを公開する

ためのファサードであるので、予約データをステートとして直接保持していません。その代わりに、図 26 に示すとおうり、ビジネス ロジックを実装したコンポーネントである BLComp3 インスタンスをメンバー変数として保持しています（例 20）。

例 20. ビジネス コンポーネントのインスタンスをメンバー変数として保持

ReservationService.cs（AdwkRService プロジェクト）

```
public class ReservationService
    : IReservation
{

(略)

    private BLComp3 comp3 = null;
```

そして、図 26 の右上の BLComp3 コンポーネントは、ステートフルなコンポーネント（ステートを維持できるクラス）であり、この BLComp3 クラスの中で、予約データを保持するために、Entity Framework の ObjectContext のインスタンスをメンバー変数として保持しています（例 21）。

例 21. ビジネス コンポーネントのインスタンスをメンバー変数として保持

BLComp3.cs（AdwkRLogic プロジェクト）

```
public class BLComp3 : IDisposable
{

   (略)

    // このコンポーネントが維持するステート
    private AdwkReservationsEntities adwkRObjectContext;
    private 予約 currentReservation;  // 現在照会している予約
```

この例のメンバー変数 adwkRObjectContext の型である AdwkReservationEntities クラスは、ADO.NET Entity Framework の Entity Data Model に基づく ObjectContext（の派生クラス）であり、データベースから照会したデータをキャッシュする機能も備えています。ここでは、この ObjectContext を利用し、照会した予約データをステートとして保持しています。

その次行の予約クラスのメンバー変数 currentReservation は、ObjectContext 内の現在照会中の予約データを特定するための変数です。

なお、ObjectContext を用いてデータを削除する場合には、予め削除対象となるデータを照会し、ObjectContext へ読み込む必要があります。その手順は、このサービスで必要となる照会から削除の一連の手順にうまく合致する点も、ObjectContext を採用した理由の一つです。

> **註:** ObjectContext を用いてデータベースから予約データを照会する方法は、BLComp3 クラスの QueryReservation1ByRCode メソッド、および QueryReservation2ByRCode メソッドを参照してください。また、予約データの削除については、BLComp3 クラスの DeleteReservationByRCode メソッドを参照してください。

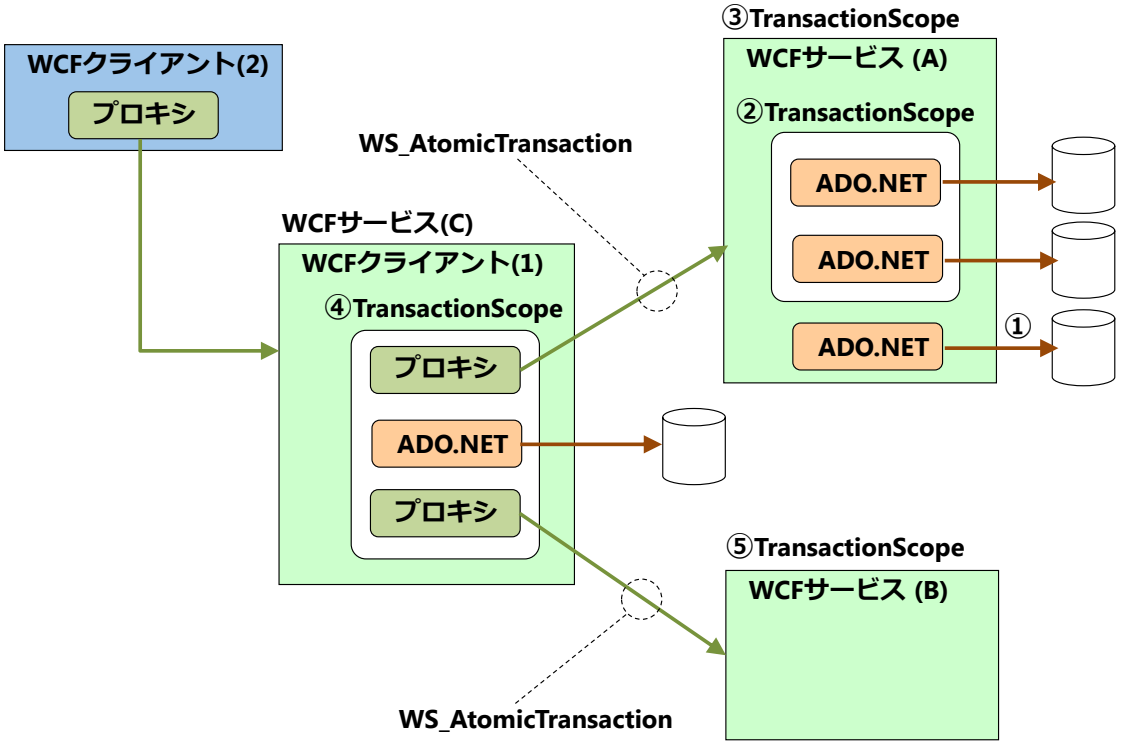
以上、クライアントや他のテクノロジとの関連もふまえ、WCF サービスにおけるステート管理の例を確認しました。

3.2 分散トランザクション

次に、分散トランザクションに対応した WCF サービスの実装方法について確認しましょう。

既に第 1 部で触れたように、WCF サービスは Enterprise Services（COM+）に基づく分散トランザクションに参加することができ、次図（第 1 部の図 3 と同じ）を用いて説明しました。

図 27 （再掲）WCF におけるトランザクションの利用



ここでは、この図の構成もふまえながら、AWTS サンプルプログラムの中で具体的にどのように実装されているかを確認します。

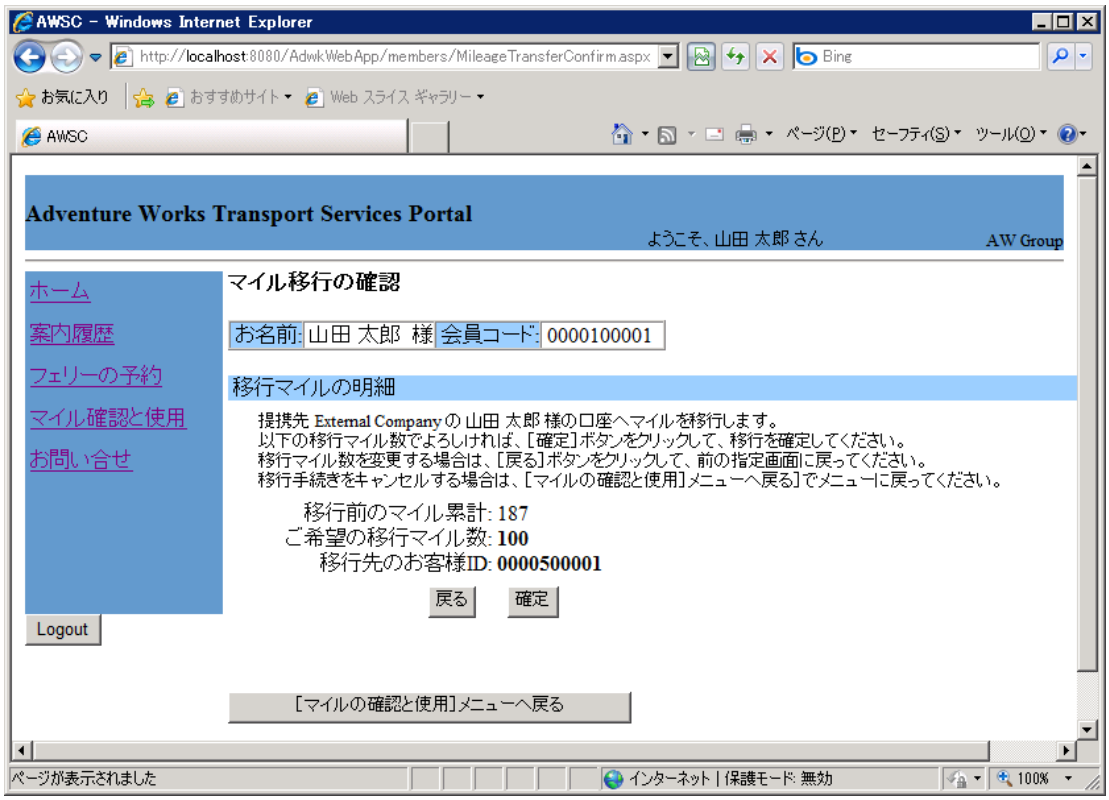
サンプルに見る分散トランザクション対応

AWTS サンプルプログラムでは、冒頭の図 2 の F、および E、I で取り上げたように、自社システムから他社システムへ、マイルを移行する際に分散トランザクションを使用しています。

このマイル移行処理は図 2 の C の ASP.NET 上の Web アプリケーションから、操作できるようになっています。エンドユーザーは、次図 28 のように、Web アプリケーション内の Web ページから、マイル移行の操作をすることができます。

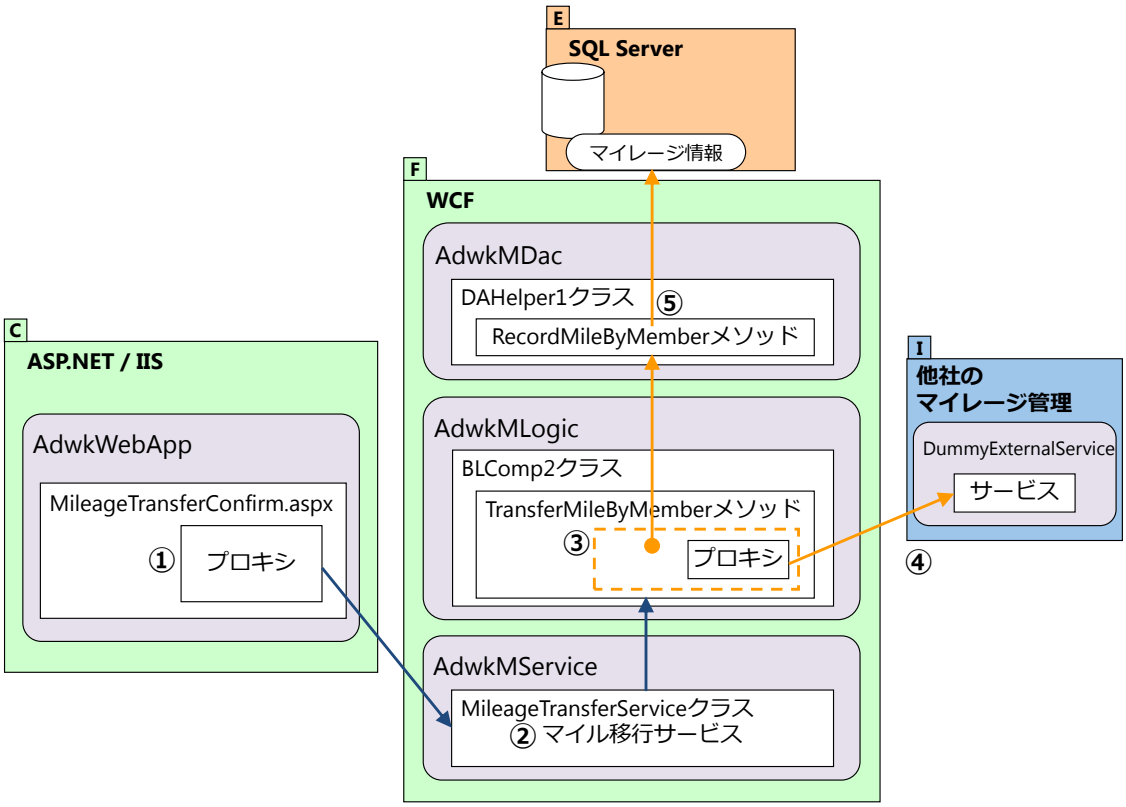
> **註:** この画面を再現するには、社内のマイレージ管理サービスである AdwkMService プロジェクト、および、社外のマイレージ管理システムである DummyExternalService プロジェクトを実行したのち、クライアントの AdwkWebApp プロジェクトを実行してください。（この実行の様子は、図 6 の構成に該当します。）また、マイル移行のために、会員にある程度のマイル実績を加算しておく必要があります（サンプルの初期状態では、既に一部の会員にある程度のマイル実績が記録されています。）。マイル加算、および、マイル移行のより具体的な方法は、セットアップガイドの後半部分である「サンプル操作方法ウォークスルー」を参照してください。

図 28 マイル移行を行う ASP.NET Web ページ



この Web ページ（MileageTransferConfirm.aspx）から起動されるマイル移行処理に関して、図 2 の構成をより具体的に示すと次図のようになります。

図 29 マイル移行における分散トランザクション



この Web ページには、図中①のように WCF サービスにアクセスするためのプロキシが使用されており、②の AdwkMService プロジェクトの WCF サービス（MilageTransferService クラス）を呼び出しています。

この WCF サービス内部では、③の AdwkMLogic プロジェクト内のビジネスロジック（BLComp2 クラス）において分散トランザクションを形成しています。このトランザクションでは、④の外部サービス操作と⑤のデータベースアクセスとを 1 つの論理的な処理として実行します（オレンジ色の部分）。

この図 29 の③の分散トランザクションは、図 27 の④の部分に相当し、1 つの TransactionScope ブロックを構成しています。この図 29 の③の TransactionScope の内部では、プロキシを使用した WCF サービス呼び出し（図 29 の④）と、ADO.NET を使用したデータベース操作（図 29 の⑤）を行っています。

このうち、図 29 の⑤は従来からある ADO.NET に基づく操作方法ですので詳細は割愛し、ここでは、図 29 の④の WCF サービスと、③の WCF クライアントについて、分散トランザクションの実装方法を確認することにします。

分散トランザクションに対応したサービス側の実装

まず、分散トランザクション対応の WCF サービスを実装した DummyExternalService プロジェクトを確認します。分散トランザクションに対応するには、サービスコントラクトとサービスクラスに必要な属性を付け、トランザクションに対応したバインディングを使用する必要があります。

DummyExternalService プロジェクトには、次のサービスコントラクトが定義されています。

例 22. トランザクションに対応したサービス コントラクト

IExternalService.cs（DummyExternalService プロジェクト）

```
// ポイント移行受付サービス
[ServiceContract(Namespace="http://DummyExternalService")]
public interface IExternalService
{
    // 加算すべきポイントの受付け
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]  //←[1]
    bool TransferPoint(
        string fromSystem,    //移行元
        string toCustomerId,  //顧客Id
        string lastName,      //姓
        string firstName,     //名
        int point,           //移行ポイント
        out int transferId,   //移行Id(移行処理単位の識別番号)
        out string message);  //メッセージ
}
```

上記の[1]ように、トランザクションに参加すべきメソッドに TransactionFlow 属性を付け、そのコンストラクターの引数に、TransactionFlowOption 列挙型を使用して、トランザクションを有効にするか否かを指定します。ここでは「Allowed」と指定してあるので、バインディングなどの構成において、トランザクション可能によるよう構成してあれば、トランザクションに参加できます。

このほか、トランザクション可能な構成を強制する「Mandatory」やトランザクションの使用を無効にした「NotAllowed」もあります。

さらに、サービスクラスのメソッドには、トランザクションのよりきめ細かい制御をするための属性が付きます。上記のコントラクトを実装したサービス クラスは、次のように定義されています。

例 23. トランザクションに対応したサービス クラス

ExternalService.cs（DummyExternalService プロジェクト）

```
// ポイント移行受付サービス
public class ExternalService
    : IExternalService
{
    // 加算すべきポイントの受付け
    [OperationBehavior(                    //←[2]
        TransactionScopeRequired = true,   //←[3]
        TransactionAutoComplete = true)]   //←[4]
    bool IExternalService.TransferPoint(   //←[5]
        string fromSystem,
        string toCustomerId, string lastName, string firstName,
        int point, out int transferId, out string message)
    {
        return PointBLComp1.TransferPoint(  //←[6]
            fromSystem,
            toCustomerId, lastName, firstName, point,
            out transferId, out message);
    }
}
```

この例の[5]の WCF サービス メソッドのように、[2]で OperationBehavior 属性を付け、さらにプロパティ（[3]や[4]）に必要な値を設定します。

プロパティのうち、[3]の TransactionScopeRequired プロパティは、TransactionScope を構成するためのものであり、この属性が付いた[5]の TransferPoint メソッド全体が、1 つの TransactionScope のブロックになります。

この TransactionScopeRequired プロパティが True の場合は、呼び出し元が既に TransactionScope の中で実行されていると、このメソッドも同じトランザクションの中で実行されます。呼び出し元が TransactionScope の中にないと、呼び出されたこのメソッドは、新規にトランザクションを作成します。また、TransactionScopeRequired プロパティが False の場合には、トランザクションに参加しません。

この例では、呼び出し元である社内システム（AdwkMLogic プロジェクト）が TransactionScope を作成するので、呼び出されたこの社外システム（DummyExternalService）も同じトランザクションに参加します。

また、[4]の TransactionAutoComplete プロパティは、分散トランザクションの投票（vote）を制御するためものです。このプロパティが True の場合は、何も特別な制御をしなければ、文字通り、自動的にトランザクションの成功に投票することになり、TransactionScope の終了時に、トランザクションに参加したすべての操作がコミットされます。さらに[4]の設定では、このメソッドから外部へ例外をスローすると、この TransactionScope で行われているトランザクションはロールバックします。つまり、例外発生の有無によって、トランザクションのコミットやロールバックの制御を行うことができます。

なお、TransactionAutoComplete 属性を False に設定した場合は、トランザクションをコミットするめたに明示的に投票する必要があり、OperationContext.Current.SetTransactionComplete メソッドを呼び出す必要があり、何もしなければロールバックします。

これで、サービスクラスのメソッドは、1 つの TransactionScope のブロックとして構成されました。このメソッドの中で行うリソースへの操作は、TransactionScope 配下のトランザクションに参加できるようになります。この例 23 では、[6]の PointBLComp1.TransferPoint メソッドの中で、移行後のポイントをデータベースに加算する処理が記述されています。よって、移行後のデータベースへの加算も、この WCF サービスのトランザクションに参加します。（正確に言えば、呼び出された[6]のメソッドの中には、さらに入れ子で TransactionScope のブロックがありますが、これは呼び出し元である[5]のメソッドと同じトランザクションに参加します。）

また、サービスのバインディングもトランザクション対応のものを使用する必要があります。ここでは、次の[1]のように wsHttpBinding を使用しています。

例 24. トランザクション対応のバインディングを使用する

app.config（DummyExternalService プロジェクト）

```
  // ポイント移行受付サービス
    <service name="DummyExternalService.ExternalService"
             behaviorConfiguration="MyBehaviors">
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost:8900/ExternalService/"/>
        </baseAddresses>
      </host>
      <endpoint address=""
                binding="wsHttpBinding"    ←[1]
                bindingConfiguration="MyBinding"
                contract="DummyExternalService.IExternalService" />
      <endpoint contract="IMetadataExchange"
                binding="mexHttpBinding"
                address="mex" />
    </service>
```

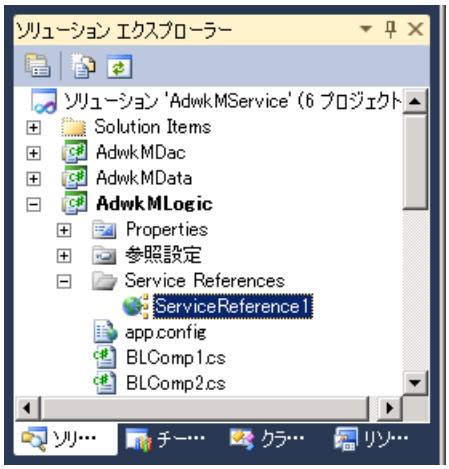
註: 各種バインディングのトランザクションのサポート状況に関しては、以下のアドレスのページに表が掲載されています。
http://msdn.microsoft.com/ja-jp/library/ms731092.aspx
システムが提供するバインディングの構成

分散トランザクションに対応したクライアント側の実装

次に、分散トランザクション対応の WCF サービスにアクセスするクライアントについて確認します。WCF サービスがトランザクション対応であれは、そのことはメタデータに反映されるので、クライアント側で「サービス参照の追加」を行えば、このトランザクション対応サービスを利用できるプロキシが生成されます。

クライアント側である AdwkMLogic プロジェクト（図 29 の③）には、前述のサービスに対して「サービス参照の追加」を既に行っており、プロキシが用意されています。ソリューション エクスプローラーでは、図 30 のように表示されています。

図 30 AdwkMLogic プロジェクトでのサービス参照の追加



生成されたプロキシが使用するコントラクトにも、次の[1]ように、TransactionFlow 属性が付いています（紙面の都合で、途中改行しています）。

例 24. トランザクション対応のバインディングを使用する

Reference.cs（AdwkMLogic プロジェクト）

```
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel", "4.0.0.0")]
[System.ServiceModel.ServiceContractAttribute(
    Namespace="http://DummyExternalService",
    ConfigurationName="ServiceReference1.IExternalService")]
public interface IExternalService {
    [System.ServiceModel.OperationContractAttribute(
      Action="http://DummyExternalService/IExternalService/TransferPoint",
      ReplyAction=
        "http://DummyExternalService/IExternalService/TransferPointResponse")]
    [System.ServiceModel.TransactionFlowAttribute(          //←[1]
        System.ServiceModel.TransactionFlowOption.Allowed)]
    bool TransferPoint(
        out int transferId, out string message, string fromSystem,
        string toCustomerId, string lastName, string firstName, int point);
}
```

> **註:** このほか、クライアント側の構成ファイルも、サービス参照の追加を行った AdwkMLogic プロジェクトの中に生成されます。このとき、AdwkMLogic プロジェクトが DLL ファイルのプロジェクト（クラスライブラリ）である点に注意してください。というのは、.NET Framework では 1 つのアプリケーションの中に、アプリケーション構成ファイルは 1 つしか持つことができず、特別なことをしなければ、EXE のプロジェクトの構成ファイルが利用されます。そのため、生成されたクライアントの構成ファイルの内容を、最終的に使用する EXE ファイルのプロジェクトへ移行する必要があります。この例でも、一旦、AdwkMLogic プロジェクトに生成された構成ファイルの内容を、あとから AdwkRService プロジェクトの構成ファイルへコピーしています。

クライアント側の TransactionScope のブロックの中で、このプロキシを使用して WCF サービスを操作するのであれば、その WCF サービスもトランザクションに参加させることができます。

ここでは、AdwkMLogic プロジェクト、BLComp2 クラスの TransferMileByMember メソッド（図 29 の③）がクライアント側に相当し、このメソッドでは、社内システムのデータベースと社外の WCF サービスとを含めた分散トランザクションが、次のように実装されています。

例 25. WCF サービスを含む分散トランザクションの制御

```
// 社内マイル減算、および、外部へのマイル移行の分散トランザクション
using (TransactionScope ts =
        new TransactionScope(TransactionScopeOption.Required))  //←[1]
{
    bool result;

    // 社内マイル減算を記録する
    int subMile = -mile;
    result = DAHelper1.RecordMileByMember(  //←[2]
                  memberCode, appliedDate, subMile, cmt, out stat);
    if (!result)
    {
        switch (stat)
        {
            case AdwkMStatus.Success:
                systemMessage = "";
                break;
            case AdwkMStatus.MemberNotFound:
                systemMessage = msgMemberNotFound;
                break;
            case AdwkMStatus.DBError:
                systemMessage = msgDBError;
                break;
            case AdwkMStatus.MileNotEnough:
                systemMessage = msgMileNotEnough;
                break;
            default:
                systemMessage = msgEtcError;
                break;
        }
        // 記録失敗のため処理中止(SetCompleteしない)
        return false;
    }

    // この時点で、社内の記録は暫定的に正常終了
    System.Diagnostics.Debug.Assert(
        result && (stat == AdwkMStatus.Success));
    systemMessage = "";

    // 社外移行のための名前取得(既存のDACを流用)
    // (この移行では、移行先と名義が一致しないと移行できない)
    string toLastName, toFirstName;
    var table1 = new AdwkMileagesDataSet.会員マイル累計DataTable();
    var adapter1 = new 会員マイル累計TableAdapter();
    int count = adapter1.Fill(table1, memberCode);
    System.Diagnostics.Debug.Assert(count > 0); //該当会員がいることはチェック済み
    toLastName = table1[0].姓;
    toFirstName = table1[0].名;

    // 社外のポイント管理システムへの移行
    int transferId;
    string externalMessage;
    var proxy = new ExternalServiceClient();  //←[3]
    result = proxy.TransferPoint(             //←[4]
              out transferId, out externalMessage,
```

```
                fromSystem,
                toCustomerId, toLastName, toFirstName, mile);
        if (!result)
        {
            systemMessage = msgExternalError + (externalMessage ?? "");
            stat = AdwkMStatus.ExternalError;
            // 外部システムでのエラーのため処理中止(SetCompleteしない)
            return false;
        }

        // この時点で、両方の処理が成功
        systemMessage = String.Format(msgTrannsferSuccess, transferId);
        stat = AdwkMStatus.Success;

        //◆◆実験用コード◆◆
        //  移行マイル(移行ポイント)が「77」マイルの場合は、SetCompleteせずに終了。
        //  外部サービスもロールバックできる。
        if (mile == 77)  //←[5]
        {
            systemMessage = "ロールバックの実験";
            stat = AdwkMStatus.UndefinedError;
            return false;
        }

        //正常終了に投票(vote)
        ts.Complete();  //←[6]
    }                   //←[7]
```

この例では、[1]から[7]までが、1 つの TransactionScope のブロックとして、論理的な 1 つのトランザクションが構成されています。

この TransactionScope の内部では、まず、[2]の DAHelper1.RecordMileByMember メソッドの呼び出しによって、ADO.NET を使用し、社内のデータベースから移行マイルを減算しています。そして、[3]では外部システムの WCF サービスにアクセスするためにプロキシのインスタンスを作成し、[4]では、その WCF サービスのメソッドを呼び出して、移行マイルを加算しています。

これらの処理のいずれかの内部で不都合が発生し、成功に投票しなければ、全体をロールバックできます。最終的には[6]で成功に投票することで（Complete）を呼び出すことで、TransactionScope の終了時にすべてがコミットされます。

なお、[5]の if ブロックは実験用のコードであり、移行マイルが「77」の場合は、[6]の Complete メソッドを実行する前に、return 文によって処理を強制終了しています。つまり、「77」の場合は、社内システムでの減算や、社外サービスでの加算については、個別に正常終了したのち、最後の Complete ソメッドだけが実行されないようになっています。この場合は、全体がロールバックするはずなので、減算と加算の 2 つの処理が 1 つのトランザクションとして扱われていたことが確認できます。（図 28 のマイル移行の際に、「77」を入力すれば、この部分が実行されます。）

これで、WCF サービスにおける分散トランザクションの実装方法が、一通り確認できました。

註: これ以外にも分散トランザクションが実装されています。図 2 において、⑤の乗船手続きの際に、⑨のマイレージ管理サービスを呼び出して、マイル加算を行っている部分です。この処理では、⑨のマイレージ管理サービスのデータベース⑦にマイル加算するだけでなく、⑤の搭乗手続きに使用するデータベース⑩にも、加算済みの記録が残ります。そのため、⑦と⑩の両者のデー

タベースの整合性を保つために、⑤のビジネスロジックと⑨の WCF サービスを 1 つのトランザクションに参加させています。余力のある方は、復習として、以下の各実装も確認してみてください。

トランザクション対応コントラクト -- IMileageUpdate.cs （AdwkMService プロジェクト）
トランザクション対応サービス -- MileageUpdateService.cs（AdwkMService プロジェクト）
クライアント側のトランザクション制御 -- BLComp4.cs（AdwkRService プロジェクト）

3.3 ASP.NET Web サイトでのホスティング

第 1 部の「2.2 WCF（基本部分）の実装概要と開発環境」では、ホスティング環境の種類について列挙し、それぞれの特徴について概説しました。このうち、ここでは (b) を用いたパターンについて、典型的な実装方法の 1 つである ASP.NET Web アプリケーションのサイト上でホスティングを行う方法を説明します。

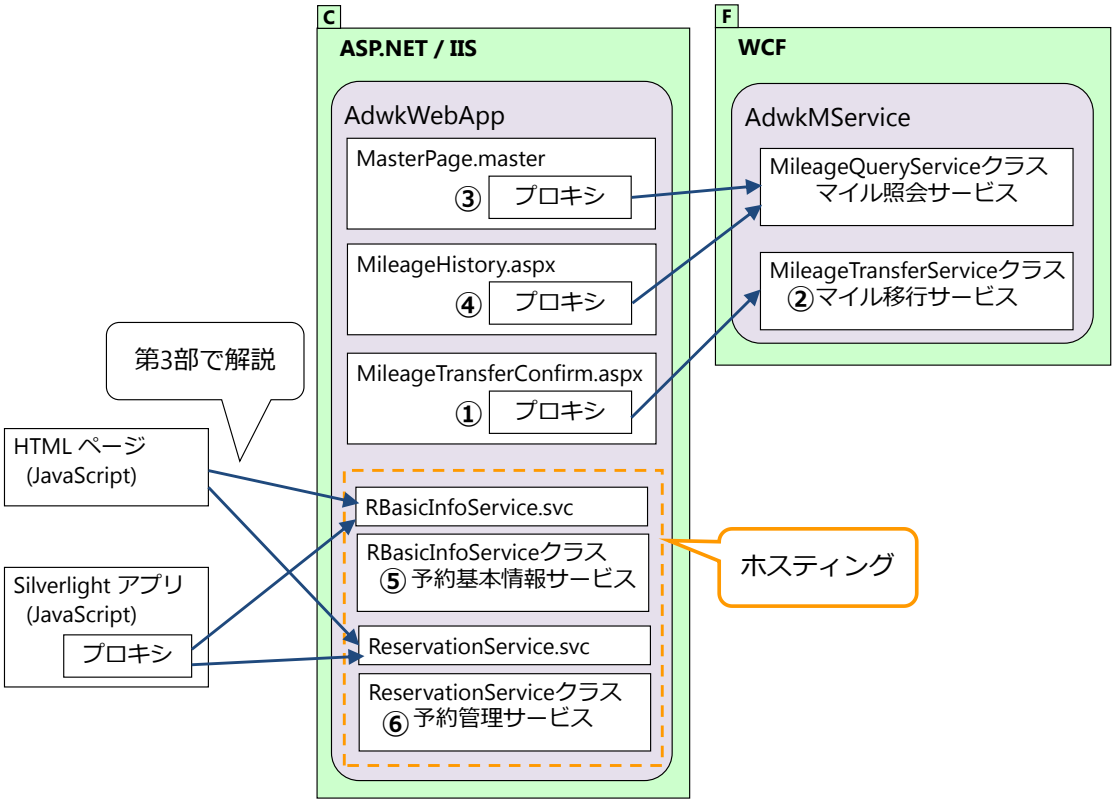
（a）自己ホスト（スタンドアローンのマネージアプリケーション）
（b）.NET Framework を用いて実装した Windows サービス（NT サービス）
（c）インターネットインフォメーションサービス（IIS 5.1、IIS 6.0、IIS 7.0）
（d）Windows プロセス アクティブ化サービス（WAS）

ただし、ASP.NET Web サイトと共に WCF サービスの利用方法としては、ASP.NET の Web サイト上で直接ホスティングしない方法もあるので、まずは、ASP.NET に関連した WCF サービスの利用方法について、AWTS サンプルプログラムを用いながら改めてパターンを整理したのち、ASP.NET におけるホスティングの構成方法について解説します。

サンプルに見る ASP.NET Web アプリケーションでの WCF サービスの利用

次図は、このサンプルの ASP.NET Web アプリケーションについて、WCF サービスの利用している箇所を示しています。（必要に応じて、図に記載されたファイル名の実装を調べてみてください。）

図 31. ASP.NET Web アプリケーションでの WCF の利用



この図の①と②は、図 29 の①および②と同じであり、Web サイト上の ASP.NET Web ページ（.aspx ファイル）からプロキシを介して、WCF サービスにアクセスする方法です。正確には、このサンプルの Web ページでは、プログラムコードが分離コードファイルに書かれているので、①の場合であれば、MileageTransferConfirm.aspx.cs に実装されています。

このほか、同様の実装が、③のマスターページ（MasterPage.master の分離コードファイル）、および、④の「マイルの確認（加算と使用の履歴）」（MileageHistory.aspx の分離コードファイル）に記述されています。

いずれにしても、ASP.NET Web ページの実装自体は、サーバーサイド上で実行されるので、Web サーバー上の実装が WCF クライアントとなって、別のサーバーの WCF サービスと連携する形態になっています。この場合、Web サーバー自体は WCF のホスティング環境ではありません。

この実装パターンでは、Web クライアントであるブラウザー自体は、WCF サービスに直接アクセスするわけではないので、ブラウザー側の実装は比較的簡単になり、より多くの種類のブラウザーで対応可能であり、汎用性があります。トレードオフとしては、仮にブラウザーが WCF サービスを直接呼び出せば、単純なデータのやり取りで済むような状況でも、一般的にページ単位にサーバーにアクセスすることになるので、サーバーとのやり取りの負荷が、比較的大きくなることがあります。

図中の⑤や⑥が、ASP.NET Web サイト上で WCF サービスをホスティングするパターンです（ファイル構成については後述）。AWTS サンプルプログラムでは、フェリー予約の基本情報の提供や予約に関する操作について、WCF サービスとして実装されています。

この実装パターンでは、Web クライアントであるブラウザーから JavaScript や Silverlight アプリケーションを使用して、サービスにアクセスします。この方法では、必要な情報だけを WCF サービスから

取得できるので、一般的なページ単位のコンテンツの取得に比べて、サーバーとの無駄なやり取りがなくなります。また、WCF サービスはページの部分的な更新の際に利用できるので、ページ単位の更新に比べて、よりきめ細かいユーザーインターフェイスの制御が可能になります。このほか、ASP.NET Web サイトで WCF サービスをホスティングすることによって、ASP.NET 互換モードを利用することが可能になるので、ASP.NET の認証やセッションの仕組みを、そのまま、WCF サービスで利用できるようになります。

トレードオフとして、JavaScript や Silverlight を使用して WCF サービスにアクセスするため、ブラウザー側はそれが可能な環境であるという制約を受けますが、このような実装に REST を用いることで、ほとんどのブラウザーから利用できるようになります。

REST に関する実装方法は、そのクライアント側の実装を含め、第 3 部で改めて取り上げるので、このあとは、ASP.NET Web サイトでの基本的なホスティングや WCF サービスの構成方法について説明します。

ASP.NET でのサービスの構成とホスティング

ASP.NET Web サイトで WCF サービスを実装する場合でも、サービスコントラクトとサービスクラスの実装方法は同じです。このほか、この WCF サービスを Web クライアントからアクセス可能にするには、アクセスポイントとなる拡張子.svc のファイルが原則として必要になります。図 31 の⑤に挙げた予約の基本情報を提供するサービスの場合は、以下のファイルから構成されています。

IBasicInfo.cs ---- サービス コントラクト

RBasicInfoService.svc.cs ---- サービス クラス

RBasicInfoService.svc ---- クライアントからのアクセスポイントとなるファイル

このうち、.svc ファイルには、次のようにサービスの基本構成を表す専用のディレクティブが記述されています。

例 26. .svc ファイル

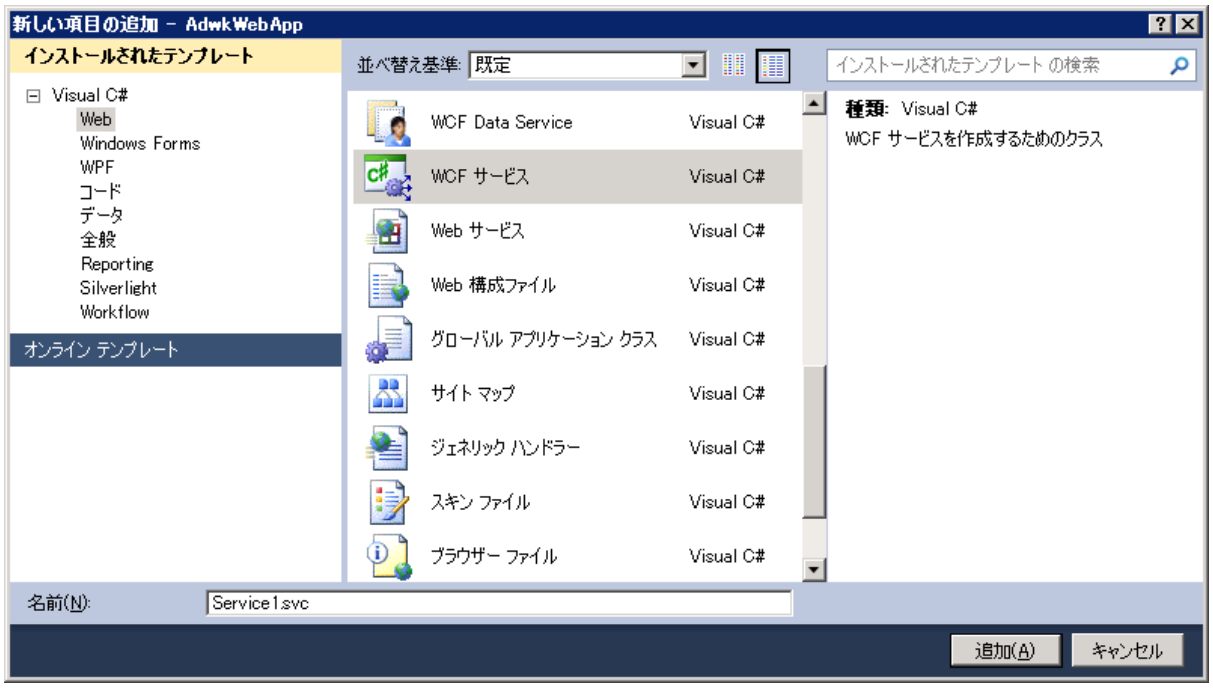
```
<%@ ServiceHost Language="C#" Debug="true" Service="Adwk.WebApp.RBasicInfoService"
 CodeBehind="RBasicInfoService.svc.cs"
 %>
```

このファイルに記述された意味は想像が付くでしょう。特にキーとなる属性は Service 属性であり、サービスクラスの名前を記述します。

> **註: .**svc ファイルの属性についてのバリエーションは、第 3 部で REST に関する実装の中で説明します。

実際には、この .svc ファイルをプログラマーが手作業で記述する必要はありません。Visual Studio では、Web サイトに追加する項目テンプレートの中に「WCF サービス」が用意されており、このテンプレートを使用すると、.svc ファイルの名前を指定することで、上記の 3 種類のファイルのひな形が自動生成され、特に.svc ファイルは修正せずに、そのまま利用することができます。

図 32. WCF サービスの追加



なお、.svc ファイル自体は Web クライアントが実際にアクセスするサイト上に配置します。

一方、サービスコントラクトやサービスクラスについては、2 つの配置方法があります。1 つは、コンパイルしてアセンブリファイルにした上で、Web サイト上の bin サブフォルダーに配置する方法です。もう 1 つは、App_Code サブフォルダーにソースコードのまま配置する方法です。

Visual Studio では、Web 関連のプロジェクトの種類に応じて、2 つの配置方法のいずれかが既定で使用されます。

今回の例のように「Web アプリケーション プロジェクト」を使用した場合、図 32 の WCF サービスの追加をサイトのルート フォルダーに対して行うと、サービス コントラクトとサービス クラスのソースコードは、Web サイトのルート フォルダー直下に生成されますが、ビルドすることによって、bin サブフォルダーに配置されます。

一方、「Web サイト プロジェクト」を使用した場合、図 32 の WCF サービスの追加をサイトのルート フォルダーに対して行うと、.svc ファイルはルートフォルダーに作成されますが、サービス コントラクトとサービス クラスは、App_Code サブフォルダーに生成されます。運用時には、この App_Code サブフォルダーのソースコードをそのまま使うことになります。

WCF サービスの構成については、Web アプリケーションの構成ファイルに記述します。記述方法は、スタンドアローン アプリケーションの構成ファイルと同様です。今回のサンプルの Web プロジェクトでは、予約の基本状況を提供するサービスでは、次のように記述されています。

例 27. Web.config での WCF サービスの構成（抜粋）

```
<system.serviceModel>

   (略)

  <services>
    <!-- 予約管理サービスの基本情報を返すサービス -->
```

```
    <service name="Adwk.WebApp.RBasicInfoService"
             behaviorConfiguration="MyBehaviors">
      <endpoint contract="Adwk.WebApp.IBasicInfo"
                binding="webHttpBinding"
                address="" behaviorConfiguration="MyBehav"/>
      <endpoint contract="IMetadataExchange"
                binding="mexHttpBinding"
                address="mex" />
    </service>
  </services>

</system.serviceModel>
```

WCF サービスの構成は、<system.serviceModel>要素ブロックの中の、<services>要素ブロックの中に記述する点は同じです。このサービスの構成では、REST に関する構成も含むので、第 3 部で改めて取り上げます。

註: 本来は、1 つのサービス クラスごとに、1 つの .svc ファイルを作成する必要があります。つまり、3 つのサービス クラスを実装するのであれば、3 つの .svc ファイルを作成する必要があります。しかし、.NET Framework 4 では新機能である「構成ベースのアクティブ化」を用いると、わざわざサービスの数だけ、.svc ファイルを用意する必要はありません。「構成ベースのアクティブ化」では、.svc ファイルを用いずに、Web.config の中に、.svc ファイル相当の記述を行うことができます。この方法によって、複数の .svc に相当する実装を、一か所に集約することができます。前述の予約の基本情報のサービスの場合であれば、<system.serviceModel>要素の中に、次のように記述します。

```
    <serviceHostingEnvironment>
        <serviceActivations>
            <add relativeAddress="RBasicInfoService.svc"
                 service="Adwk.WebApp.RBasicInfoService" />
        </serviceActivations>
    </serviceHostingEnvironment>
```

3.4 ASP.NET 互換モードの利用

前項でも触れたように、ASP.NET Web アプリケーションの Web サイト上で、 WCF サービスのホスティングを行う場合、ASP.NET 互換モードが利用できます。これによって WCF サービスは、ASP.NET Web アプリケーションの認証やセッションを併用でき、改めて WCF サービスだけのために、このような仕組みを実装する必要はなく、開発生産性が向上します。また、Web ページと WCF サービスとで個別にユーザー管理を行う必要はなく、管理も簡素化します。

特に REST 対応のサービス（RESTful なサービス）では、クライアントからサービスへの 1 回の呼び出しが、単純な HTTP 要求に基づくものであり、それ自身にセキュリティやステート管理の仕組みがありません。そのため、他のセキュリティの仕組みを併用する必要があり、そのような状況では、ASP.NET 互換モードが役立ちます。

ASP.NET 互換モードにおける RESTful なサービスの具体的な実装方法は、第 3 部で扱うので、ここでは、ASP.NET 互換モードに関して最低限必要な構成方法について確認します。

ASP.NET 互換モードの構成方法

ASP.NET 互換モードを WCF サービスで利用するための構成として、以下の 2 つ点が必要になります。

（a）ASP.NET Web アプリケーション単位での、アプリケーション構成ファイルにおける構成

（b）WCF サービスクラス単位での、互換モードを利用可能にする属性の指定

まず（a）については、この AWTS サンプルプログラムでは、AdwkAppWeb プロジェクトのアプリケーション構成ファイル（Web.config）ファイルの中で、WCF の構成ブロックに次のように指定されています。

例 28. Web アプリケーションでの ASP.NET 互換モードの構成（抜粋）

Web.config（AdwkWebApp プロジェクト）

```
<system.serviceModel>

  <!-- ASP.NET 互換モード -->
  <serviceHostingEnvironment aspNetCompatibilityEnabled="true" />   ←[1]

   （略）

</system.serviceModel>
```

[1]のように、<serviceHostingEnvironment>要素の aspNetCompatibilityEnabled 属性に true を設定することで有効になります（既定値は false であり、ASP.NET 互換モードは無効になっています）。

また、（b）のサービスクラス単位の指定については、AdwkAppWeb プロジェクトの members フォルダーに含まれるファイル ReservationService.svc.cs のサービスクラスに、次例のように専用の属性が付加されています。

例 29. ASP.NET 互換モード環境下でのサービスの利用を可能にする（抜粋）

ReservationService.svc.cs（AdwkWebApp プロジェクト）

```
[AspNetCompatibilityRequirements(
    RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]
public class ReservationService : IReservation
{

 （略）

}
```

この例のサービス クラスの冒頭ように、AspNetCompatibilityRequirements 属性の RequirementsMode プロパティに対して、AspNetCompatibilityRequirementsMode 列挙型の必要な値を指定します。この例では、「Allowed」と指定してあるので、ASP.NET 互換モードでのサービスの使用が可能になります。

なお、既定値では「NotAllowed」であり、明示的に属性で指定しないまま ASP.NET 互換モードでサービスを実行すると、実行時エラーになります。また、このプロパティの値として「Required」と指定することもでき、この場合は、このサービスクラスを ASP.NET 互換モードで使用することが必須となり、ASP.NET 互換モード以外でサービスを実行すると、実行時エラーになります。

これらの構成をするだけで、ASP.NET Web アプリケーションでの認証などの仕組みが WCF サービスでも利用できるようになります。

たとえば、この AWTS サンプルプログラムでは、次のようにアプリケーション構成ファイルにおいて、[1]でフォーム認証が指定されており、また、[2]および[3]で members フォルダーへの未認証ユーザーのアクセスを禁止しています。これを WCF サービスでも利用しています。

例 30. フォーム認証とアクセス制御の指定（抜粋）

Web.config（AdwkWebApp プロジェクト）

```
<system.web>

   (略)

  <!-- フォーム認証を使用 -->
  <authentication mode="Forms"/>    ←[1]

   (略)

</system.web>

   (略)

<!-- membersフォルダー -->
<location path="members">    ←[2]
  <system.web>
    <authorization>
      <deny users="?" />      ←[3]
    </authorization>
  </system.web>
</location>
```

クライアントの Web ページ上から、JavaScript などを用いて、members フォルダーの WCF サービス ReservationService.svc にアクセスする際には、この構成があるため、未認証の状態ではアクセスに失敗します。（厳密に言えば、サーバー側からログインページへのリダイレクトが要求されますが、いずれにしても、未認証のままでは JavaScript から直接 WCF サービスを呼び出せません。）

アクセスするためには、あらかじめ、Web サイト上のログインページからログインして、認証済みの状態にしておく必要があります。

> **註:** ASP.NET Web アプリケーションと WCF サービスとのより詳しい情報の受け渡しなど、両者の連携は第 3 部で扱います。

サンプルに見るサービスでの ASP.NET 認証の効果確認

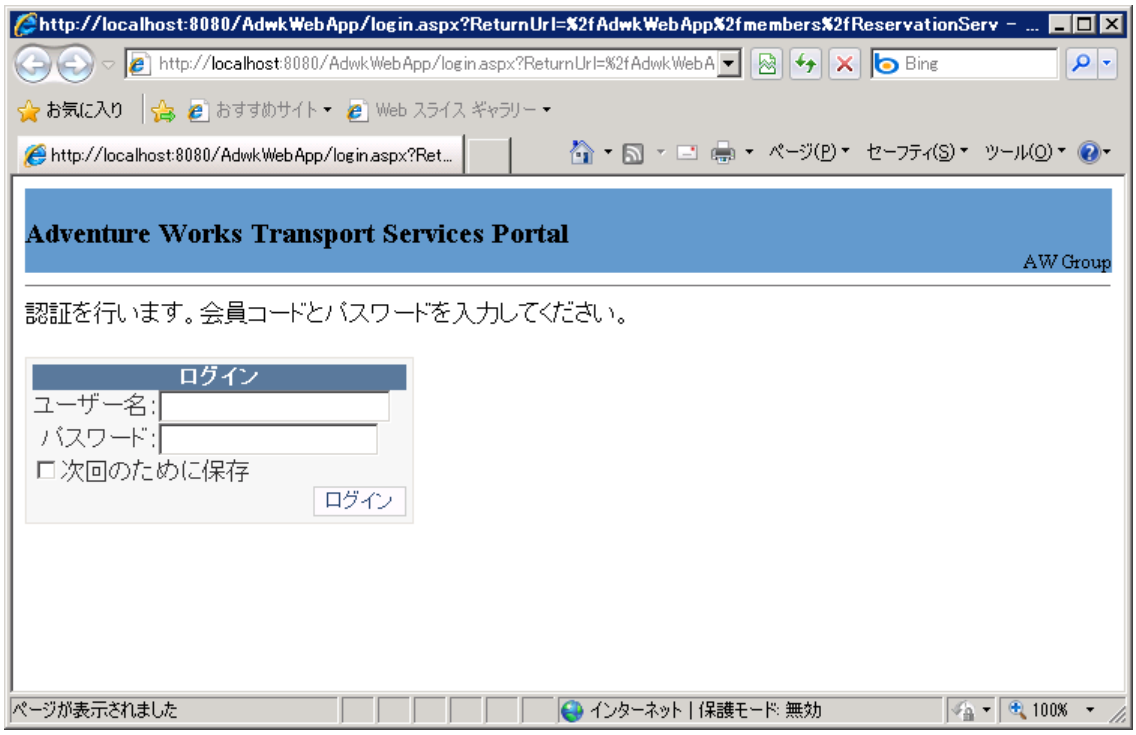
ここで、ASP.NET 互換モードが有効であることを確認するため、簡単な実験を行ってみましょう。本来であれば、WCF サービスを使用する際に、.svc ファイルのアドレスに対して、手動でブラウザーから直接開くことはありませんが、ここでは実験のため、ブラウザーから手動で開いてみます。

1. 念のため未認証の状態にするため、現在開いているブラウザーをすべて閉じます。
2. AdwkWebApp プロジェクトのソリューション エクスプローラー上で、members フォルダーの中にある ReservationService.svc を右クリックして、ショートカットメニューから［ブラウザーで表示］をクリックします。

.svc ファイルをブラウザーで開くと、本来は WCF サービスの「HTML ヘルプ ページ」が開くはずです。しかし、ここでは ASP.NET 互換モードが有効になっており、ASP.NET において未認証の状態なので、ログイン ページが表示されます。（逆に言えば、ASP.NET 互換モードが無効になっていると、認証の如何にかかわらず、「HTML ヘルプ ページ」が開きます。）

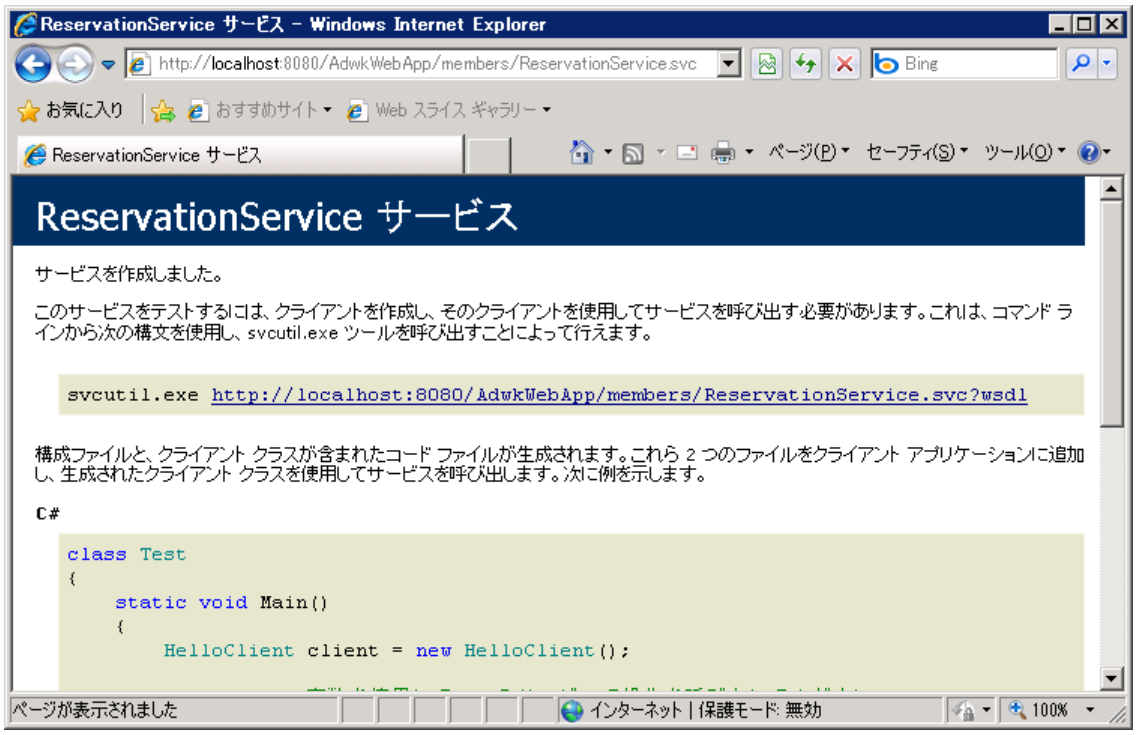
3. HTML ヘルプ ページは直ぐには表示されずに、次図のようにログイン ページにリダイレクトされることを確認します。

図 33. 未認証のためログイン ページにリダイレクトされる



4. ユーザー名「0000100001」、パスワード「password」でログインします。
5. 認証に成功すると、アクセス先である .svc ファイルの HTML ヘルプ ページが開くことを確認します。

図 34. 認証に成功すると .svc ファイルの HTML ヘルプページが開く



以上の手順で確認したように、通常の ASP.NET の認証プロセスに、WCF サービスも参加することができます。確認が済んだら、ブラウザーを閉じておいてください。

> **註:** 余力があれば、例 28 の ASP.NET 互換モードの設定をコメントアウトして、ASP.NET 互換モードを無効化し、WCF サービスへの影響を確認してみてください。無効化した状態では、前述の手順 2 のように members フォルダーの .svc ファイルを開いた場合、ログインページへのリダイレクトはなく、直ぐに .svc ファイルの HTML ヘルプページが開きます。もちろん、この場合でも、依然として、通常の ASP.NET Web ページに関しては、フォーム認証のプロセスが利用でき、あくまで WCF サービスが ASP.NET の認証プロセスに参加できない状態になります。

3.5 Windows 認証の構成と利用

クライアントが WCF サービスにアクセスにする際に、サービスがクライアントを認証する方法には様々なものがあります。たとえば、Windows 認証 (NTLM および Kerberos)、X.509 証明書を用いた認証、ユーザー名/パスワードによるカスタム認証などを利用できます。これらの認証方法を使用する構成は、バインディングの詳細を構成する中で行います。さらに、認証に使用するクレデンシャル(ユーザー名とパスワード）のやり取りは、トランスポートレベルで行うことも、メッセージレベルで行うことも可能です。また、WCF の構成では認証方法を明示せずに、前節のように、ASP.NET 互換モードを用いて、ASP.NET の認証をそのまま流用する方法もあります。

ここでは、トランスポートレベルで Windows 認証を使用する方法を確認します。併せて、認証に基づいて、サービス側でのアクセス制御（承認）である、ロールベース セキュリティを用いる方法も確認します。

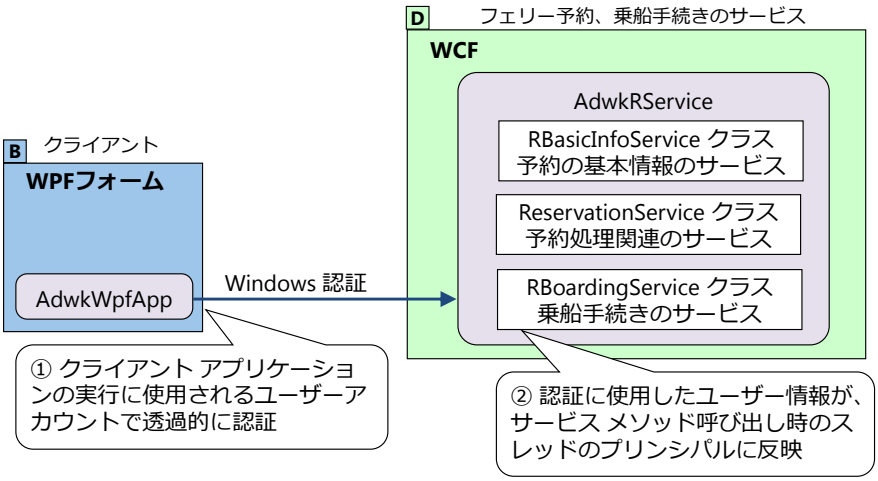
> **註:** WCF サービスにおける認証の様々な構成方法が、以下のアドレスにも記載されています。
> http://msdn.microsoft.com/ja-jp/library/ms733082.aspx 認証

註: そもそも、WCF サービスにおける認証では、サービスとクライアントが相互に認証する方式を採用しています。つまり、サービスがクライアントを認証する「クライアントの認証」だけでなく、クライアントがサービスを認証する「サービスの認証」も行っています。サービスの認証は、クライアントがフィシングサイトに実装されたサービスへの接続を防ぐことにつながります。
ここで取り上げている認証は、サービスがクライアントを認証する方法です。前節の ASP.NET 互換モードにおけるフォーム認証の流用も、Web サイト側（サービス側）がクライアントを認証する方法の 1 つです。
サービスの認証にも複数の方法があり、認証方法によって、サービスを認証する際に使用する情報が異なります。たとえば、サービスを識別する情報として、サービス プロセスのユーザー アカウント、サービスのドメイン名、または、サービスの証明書などが使用されます。サービスの認証に関しては、特に証明書を用いたものについて、説明の都合上、「3.6 認証のカスタマイズ」の中で取り上げています。

サンプルに見る Windows 認証の利用

この AWTS サンプル プログラムでは、すでに取り上げた図 7 の社内システムにおいて、フェリーの予約管理サービス（D）に対して、クライアント（B）がアクセスする際に、トランスポートレベルでの Windows 認証を用いています。次に、図 7 の中から抜粋し、主な特徴を示します。

図 35. Windows 認証を用いた WCF サービスへのアクセス



このサンプルでは、D のサービス側と B のクライアント側、それぞれで Windows 認証を指定するように構成してあります（構成方法は後述）。D には 3 つのサービスがあり、このサンプルでは参考までに、それぞれ異なるバインディングを用いています。のちほど、D の中の乗船手続きのサービス（図 35 の 3 番目のサービス）について、Windows 認証の構成方法を確認します。

B のクライアントを使用するユーザーは、まず、PC 上で Windows OS に対して、特定のユーザー アカウントでログオンしているはずです。そして、B のクライアントのアプリケーション プロセスは、そのユーザーアカウントを用いて実行されます。このクライアントがサービスにアクセスする際には、図中の①に示したように、クライアント アプリケーションのプロセスのアカウントを使用して、透過的にサービスへの認証を試みます。

認証が成功して、サービスのメソッドが呼び出されると、図中の②に示したように、呼び出しによって実行されるスレッドのプリンシパル（Principal）には、クライアントが認証に使用したユーザー アカウントの情報が反映されます。このプリンシパルを利用すれば、サービスを呼び出したユーザーに応じて、アクセス制御を行うことができます。このプリンシパルを用いて制御するには、通常の .NET Framework のロールベース セキュリティを用います（後ほどサンプルコードを示します）。

サービス側での Windows 認証の構成

図 35 の D の乗船手続きのサービス（RBoardingService クラス）では、バインディングとして TCP プロトコルを使用する netTcpBinding が指定されています。このバインディングは、Windows 認証が利用できるバインディングの 1 つです。次のように AdwkRService プロジェクトの構成ファイルに設定されています。

例 31. サービス側での Windows 認証の構成（抜粋）

App.config（AdwkRService プロジェクト）

```
  <!-- 予約管理サービス(乗船関連の処理) -->
  <service name="Adwk.Reservation.Service.RBoardingService"
     behaviorConfiguration="MyBehaviors">
    <host>
      <baseAddresses>
        <add baseAddress="http://localhost:8092/AdwkRService/Boarding/"/>
        <add baseAddress="net.tcp://localhost:8093/AdwkRService/Boarding/"/> ←[1]
      </baseAddresses>
    </host>
    <endpoint contract="Adwk.Reservation.IRBoarding"
              binding="netTcpBinding"                          ←[2]
              bindingConfiguration="NetTcpBinding_IRBoarding"  ←[3]
              address="" />
    <endpoint contract="IMetadataExchange"
              binding="mexHttpBinding"
              address="mex" />
  </service>

     (略)

<bindings>
  <!-- 乗船手続きのサービスでの Windows 認証の構成 -->
  <netTcpBinding>
    <binding name="NetTcpBinding_IRBoarding">       ←[4]
      <security mode="Transport">                    ←[5]
        <transport clientCredentialType="Windows" />  ←[6]
      </security>
    </binding>
  </netTcpBinding>
```

上記の[2]では、このサービスのエンドポイントのバインディングに「netTcpBinding」が指定されており、これに合わせて、[1]のベースアドレスにも TCP ネットワークプロトコルを使用するように「net.tcp://」と指定されています。

認証に関する構成は、バインディングの詳細構成として行うので、改めて[4]の<binding>要素ブロックに記述されています。[4]の name 属性では、この構成に名前（NetTcpBinding_IRBoarding）を付けており、この構成をエンドポイントに適用するために、[3]ではその名前を参照しています。

特に認証の構成に必要な部分は、[5]と[6]です。[5]の<security>要素の mode 属性には、トランスポートレベルのセキュリティを使用するか、または、メッセージレベルのセテキュリィを使用するか、それとも併用するかなど、セキュリティの基本設定を行います。ここでは、トランスポートレベルで十分なので、"transport" と指定しています。この基本設定は、認証だけでなく、暗号化や署名にも影響する全般的な設定なので、改めて[6]では、認証に関する設定を行っています。この[6]では、clientCredentialType 属性に "Windows" と指定しており、トランスポートレベルで Windows 認証のクレデンシャルをやり取りすることになります。この設定によって、実質的にクライアントが認証される際の認証方法が決定します。

ここでは、tcpNetBinding に関する認証の設定を行いましたが、他のバインディングでも、[4]から[6]までの設定は同様です。

> **註:** このサンプルでは参考として、図 35 の D の 3 つのサービスのうち、残りの 2 つのサービスでは異なるバインディングを使用しています。
> 予約の基本情報のサービス（RBasicInfoService クラス）では、basicHttpBinding が指定されており、一方、予約処理関連のサービス（ReservationService クラス）では、wsHttpBinding が指定されています。
> これら 2 つのバインディングについては、このサンプルでは、認証方法を明示的には指定していません。指定しない際の固定値は、basicHttpBinding では認証無しになり、wsHttpBinding ではメッセージレベルでクレデンシャルをやり取りする Windows 認証が使用されます。

クライアント側での Windows 認証の構成

WCF クライアントでも、認証方法を指定するには、クライアント側のエンドポイントに、例 31 と同様の指定（[4]から[6]）を行います。

ただし、Visual Studio を用いた開発環境であれば、クライアントの開発環境から「サービス参照の追加」を行うことによって、プロキシが生成されるだけでなく、アプリケーション構成ファイルにも、Windows 認証を使用する構成が自動的に生成されます。

その際、Windows 認証では、前述のとおり、クライアント アプリケーションのプロセスのユーザー アカウントを使用して認証を試みます。このサンプルのクライアント（AdwkWpfApp）もそのように構成されています。

また、クライアント アプリケーションのプロセスのアカウントを使用せずに、クライアント アプリケーション側で、明示的に特定のユーザーアカウントを認証で使用するよう指定することができます。環境の都合によっては、現在のプロセスが使用しているアカウントとは別のアカウントで認証を受けたい場合もあるでしょう。その場合には、この方法が利用できます。以下に、クライアント側で明示的にユーザー アカウントを指定する例を示します。

例 32. クライアント側での Windows 認証向けのクレデンシャルの指定（抜粋）

W03CheckIn.xaml.cs（AdwkWpfApp プロジェクト）

```
    // [乗船受付]ボタン
    private void CheckInButton_Click(object sender, RoutedEventArgs e)
    {

     （略）
```

```
        // 乗船手続き用のサービスプロキシの作成
        var proxy = new RBoardingClient();

        // 明示的な Windows ユーザーアカウントの指定
        proxy.ClientCredentials.Windows.ClientCredential.UserName = "aaa"; //←[1]
        proxy.ClientCredentials.Windows.ClientCredential.Password = "bbb"; //←[2]
        //
```

この例は、乗船手続きのサービスにアクセスする際に、Windows 認証を行うものです。サンプルのウィンドウ（W03CheckIn クラス）のイベントハンドラーの中では、[1]と[2]のように、プロキシ（変数 proxy）の特定のプロパティに、Windows 認証用のユーザー名とパスワードを指定しています。（実際のサンプルでは、この 2 つの設定はコメントアウトしてあります。）

ロールベース セキュリティでの利用（アクセス制御の利用）

ここでは、Windows 認証に基づいて、ロールベース セキュリティを使用した実装例について、乗船手続きのサービスで確認してみましょう。

すでに触れたように、クライアントがサービスにアクセスする際に認証されると、そのユーザーに関する情報が、サービス メソッドを呼び出す際のスレッドのプリンシパルへ自動的に反映されます。（カスタマイズされた認証など一部の認証では、自動的に反映されない場合もあります。）

このサンプルでは、乗船手続きのサービスが呼び出された際に、認証を受けたクライアントが特定のロールに属しているか評価し、属さない場合には、乗船手続き処理を行わないように実装されています。以下に例を示します。

例 33. サービス側での認証ユーザーに基づくロールベース セキュリティの制御（抜粋）

RBoardingService.cs（AdwkRService プロジェクト）

```
    // アクセス可能なロール
    const string validRole1 = @"builtin¥users";    //←[1]

     （略）

    bool IRBoarding.CheckIn(  //←[2]
        string scheduleCode, string levelCode,
        string reservationCode, string memberCode,
        string lastName, string firstName,
        out string systemMessage)
    {
        // ロールを用いたアクセス権の確認
        var principal1 = Thread.CurrentPrincipal;  //←[3]
        Console.WriteLine("[ReservationService.CheckIn]");
        Console.WriteLine(
            "  Type : {0}", principal1.Identity.AuthenticationType);  //←[4]
        Console.WriteLine(
            "  User : {0}", principal1.Identity.Name);                //←[5]
        Console.WriteLine("  {0} ? : {1}",
            validRole1, (principal1.IsInRole(validRole1)) ? "OK" : "NG" );  //←[6]
        if (! Thread.CurrentPrincipal.IsInRole(validRole1))  //←[7]
        {
            systemMessage = msgAccessError; //←[8]
            return false;
        }
        //
```

```
    （略）
  }
```

上記の[2]の CheckIn メソッドは、乗船手続きを行う際のメソッドです。このメソッドの中の処理を行う際には、特定のロールに属しているか評価を行っています。

ここでは、実験のため、[3]で実行中スレッドの現在のプリンシパルを参照できるように、変数 principal1 に設定しています。[4]や[5]では、そのプリンシパルに関して、それぞれ、クライアントの認証の種類、認証に使用されたユーザー名を表示しています。

また、[6]のように IsInRole メソッドを呼び出せば、現在のプリンシパルが特定のロールに属しているか判断できます。つまり、認証を受けたクライアントが、特定のロールに属しているか判断しています。引数には、属しているか判断すべきロール名（定数 validRole1）を指定しており、ここでは[1]のように、組み込みの users グループに属しているか判断しています。Windows 認証を使用した場合、そのユーザーアカウントが属するユーザーグループが、所属するロールとして、自動的にプリンシパルに反映されます。

実際のロールベース セキュリティに基づく制御は、[7]で行っています。ここでも、現在のプリンシパルが特定のロールに属しているか判断するため、IsInRole メソッドを呼び出しています。そして、[7]の if 文の評価によって、ロールに属していない場合は、そのあとの乗船手続きの処理を行わないようにするため、return 文によって、処理を強制終了しています。

参考までに、このロールベース セキュリティの作用を確認する手順を示しておきます。余力があれば確認してみてください。

1. 予約管理サービス（AdwkRService プロジェクト）を実行します。
2. クライアント（AdwkWpfApp プロジェクト）を実行します。
3. ［予約管理・乗船手続き］ウィンドウが表示されたら、［乗船手続き］ボタンをクリックします。
4. 次図に示す［乗船手続き］ウィンドウが表示されたら、「経路」や「日付」は初期値のままにして、［運行予定照会］ボタンをクリックします。

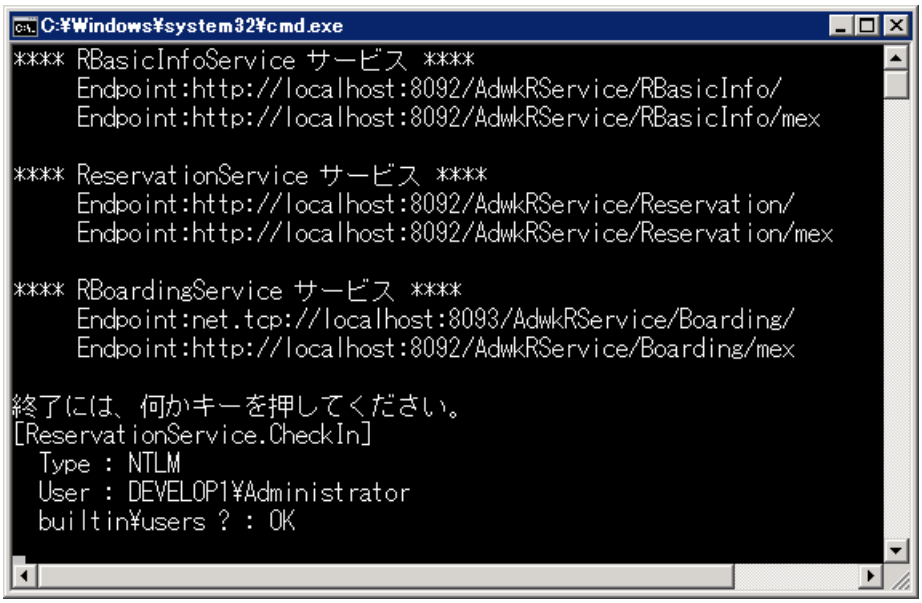
図 36. ［乗船手続き］ウィンドウ

5. すると次図のように、運行予定の一覧が、DataGrid に表示されるので、適当な行を選択した後、このウィンドウ下部にある「予約コード」、「姓」、および「名」には、適当な数字と名前を入力して、［乗船受付］ボタンをクリックします。（ここでは、正しい予約データであるか否かは重要ではありません。）

図 37. 運用予定を選択して、予約データを入力

6. すると、現在のクライアントのアカウントを使用して、乗船手続きのサービスを呼び出し、このサービスのコンソールには、認証の種類とユーザーアカウントが表示されます。また、ロールに属しているかどうかの評価が表示されます。この実行結果（true）は、アカウントが「users」ロールに属していたことを表しています。

図 38. Windows 認証の結果が反映されたプリンシパル



なお、ここでは重要ではありませんが、ロールに属することが確認できた後、CheckIn メソッド内のそれ以降の処理も進んだ際、実際には存在しない予約データを入力したので、乗船手続き自体は行われません。クライアントの［乗船手続き］ウィンドウには、「（内部エラー）システムメッセージ：指定した予約は存在しません。」というエラーメッセージが表示されます。

> **註:** さらに余力があれば、クライアントのサンプルである例 32 の[1]と[2]において、別の有効なユーザー名とパスワードを明示的に指定して認証するように変更し、上記の手順を実行してみてください。クライアントで指定されたアカウントが、図 38 のプリンシパルの実行結果に反映されるはずです。
> また、サービス側のサンプルである例 33 の[1]において、存在しないロール名に修正し、上記の手順を実行してみてください。例 33 の[7]において、ロールに所属さないと判断されるので、[8]のメッセージ（msgAccessErorr）がクライアントに返され、それ以降の処理は中止されます。クライアントには、「乗船手続きのサービスを利用する権限がありません。」とメッセージが表示されます。

3.6 カスタム認証の実装と利用

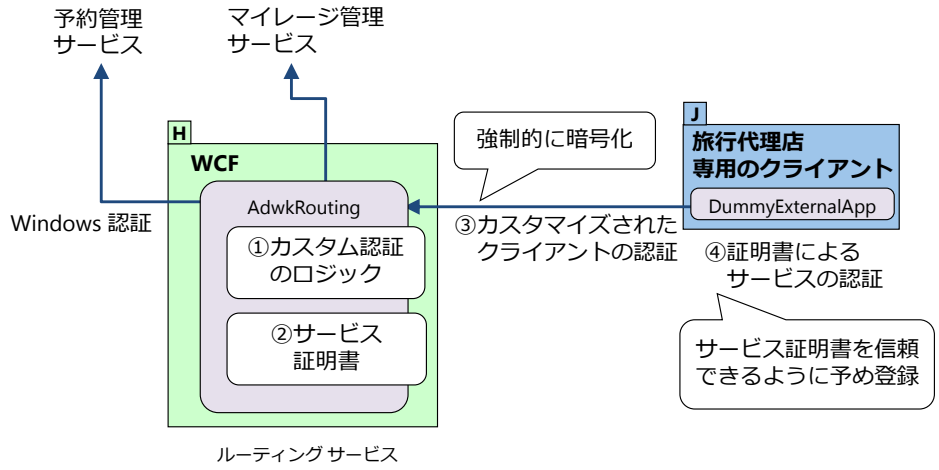
この節では、WCF のクライアントの認証方法として、独自のロジックで認証するカスタマイズされた認証方法について取り上げます。また、サンプルコードの都合上、併せてサービスの認証についても取り上げます。（クライアントの認証とサービスの認証のそれぞれの構成自体は、独立して行うことができます。）

サンプルに見るクライアントのカスタム認証とサービスの認証

AWTS サンプル プログラムでは、すでに取り上げた図 8 のように、社内システムのサービスの一部に対して、外部のパートナー企業（図 8 の J）からアクセスできるようになっています。その際には、ルーティングサービス（図 8 の H）を外部からのアクセスポイントとして利用します。このとき、外

部のパートナー企業のクライアント アプリケーションは、カスタム認証を使用しています。主な特徴を以下に示します。

図 39. カスタマイズされたクライアントの認証、および、証明書によるサービスの認証



この図の左上部には、社内システムの 2 つのサービス（予約管理、および、マイレージ管理）があります。この 2 つのサービスに対しては、外部のパートナー企業のクライアント（図中の J）から直接アクセスするのではなく、ルーティングサービス（図中の H）を介してアクセスしています。つまり、クライアント（J）が直接アクセスするサービスは、ルーティングサービスであり、その際に、③のように、カスタマイズされたクライアントの認証を使用しています。

カスタム認証のロジックは、①のようにルーティングサービスに実装されています。また、このサンプルのカスタム認証では、正確には wsHttpBinding の「UserName」という種類の認証を使用しており、この認証のやり取りには強制的にセキュリティの保護（暗号化や署名）が行われます。

また、このサンプルでは、④に示すようにクライアントでは、サービスの認証を行います。その際には、③のサービス証明書が、クライアントによってサービスを識別する情報として使用されます。このとき、クライアントがサービスを有効であると認識するためには、予めクライアント側で、サービス証明書が信頼できるものになるよう構成しておく必要があります。

また、③のサービスの証明書は、カスタマイズされたクライアント認証の暗号化や署名にも利用されます。

なお、ここではルーティングサービスに関して、カスタム認証やサービスの認証を実装しますが、ルーティングサービスも WCF サービスの一種なので、ここでの構成は、他の一般的な WCF サービスにそのまま適用できます。つまり、図 39 の H の部分は、ルーティングサービスであるという構成を除き、そのまま一般的な WCF サービスに置き換えて、カスタム認証やサービス証明書の構成を行うことができます。

> **註:** ルーティング サービスについては、改めて「3.8 ルーティング サービス」で取り上げます。サービス証明書に関する、サービス側とクライアント側の証明書ストアの構成は、「3.9 暗号化と署名」の中で説明します。

サービス側でのカスタム認証の構成

AdwkRouting プロジェクトのルーティングサービスは、次のように認証が構成されています。

例 34. カスタム認証の構成（抜粋）

App.config（AdwkRouting プロジェクト）

```
   <!-- クラスライブラリ標準のRoutingServiceクラスを使用した
        ルーティングサービスクラスの構成 -->
   <service behaviorConfiguration="routingConfiguration"        ←[1]
            name="System.ServiceModel.Routing.RoutingService">

    (略)

    <!-- ルーティングサービスへのアクセスするための
         エンドポイントの定義(予約サービス) -->
    <endpoint address="adwkr" binding="wsHttpBinding" name="adwkrEndpoint" ←[2]
              contract="System.ServiceModel.Routing.IRequestReplyRouter"
              bindingConfiguration="Binding1"/>  ←[3]

    <!-- ルーティングサービスへのアクセスするための
         エンドポイントの定義(マイレージサービス) -->
    <endpoint address="adwkm" binding="wsHttpBinding" name="adwkmEndpoint" ←[4]
              contract="System.ServiceModel.Routing.IRequestReplyRouter"
              bindingConfiguration="Binding1"/>  ←[4]

    </service>

   (略)

 <bindings>
   <wsHttpBinding>
     <!-- 二つのエンドポイントの bindingConfiguration から参照 -->
     <binding name="Binding1">    ←[5]
       <security mode="Message">  ←[6]
         <message clientCredentialType="UserName" />  ←[7]
       </security>
     </binding>
   </wsHttpBinding>
 </bindings>

 <behaviors>
   <serviceBehaviors>

     <!-- ルーティングサービスのための Service Behavior の定義 -->
     <behavior name="routingConfiguration">  ←[8]

    (略)

       <serviceCredentials>
         <!-- カスタム認証でのモジュールの指定 -->
         <userNameAuthentication userNamePasswordValidationMode="Custom" ←[9]
             customUserNamePasswordValidatorType=                        ←[10]
                "AdwkRouting.CustomUserNameValidator, AdwkRouting"/>
         <!-- security mode="Message" かつ "UserName" の場合に必要 -->
         <serviceCertificate findValue="localhost"            ←[11]
                             storeLocation="LocalMachine"
                             storeName="My"
```

```
                        x509FindType="FindBySubjectName"/>
      </serviceCredentials>

    </behavior>
```

すでに触れたように、ルーティングサービスも WCF サービスの一種であり、他の WCF サービスと同様に、構成ファイル内で<service>要素ブロックを用いて、必要な構成を行うことができます。このサンプルでは、[1]の<service>要素ブロックに、ルーティングサービスの構成が記述してあります。このルーティングサービスでは、[2]および[4]に、外部のパートナー企業が利用する 2 つのエンドポイントが定義されています。（このエンドポイントへアクセスすると、社内システムのサービスへ要求が転送されます。その転送の構成は別途取り上げます。）

この 2 つのエンドポイントでは、バインディングが wsHttpBinding に設定されています。認証方法を指定する記述は、前節で説明した方法と同様です。まず、この 2 つのエンドポイントでは、[3]および[4]の bindingConfiguration 属性に、バインディングの詳細を構成する構成名 "Binding1" が指定されています。実際のその構成は、[5]の<binding>要素ブロックに記述されています。この<binding>要素ブロックの中で、具体的な認証方法が指定されています。

wsHttpBinding でのカスタム認証である「UserName」を指定するには、[6]でメッセージレベルのセキュリティを使用するように指定する必要があります。そして、[7]では使用するクレデンシャルとして clientCredentialType 属性に "UserName" を指定します。

また、カスタム認証のために、独自に実装したプログラムコードを使用するためには、[9]の<userNameAuthentication>要素に必要な設定を行います。特に、[10]の属性には使用するアセンブリ名とクラス名を指定します。ここでは、このプロジェクト（AdwkRouting.dll）に含まれる CustomUserNameValidator クラスが認証ロジックの実装として指定されています。

[11]の<serviceCertificate>要素ブロックはサービス証明書の指定です。この指定では、ローカルコンピューター（"LocalMachine"）の「個人（"My"）」という名前の証明書ストアに配置された、名前が "localhost" という証明書（CN=localhost のもの）が、サービス証明書になるよう構成しています。

カスタム認証が実装された CustomUserNameValidator クラスのコードは、同じ AdwkRouting プロジェクトの中に、次のように実装されています。

例 35. カスタム認証の実装コード（抜粋）

Program.cs（AdwkRouting プロジェクト）

```
public class CustomUserNameValidator : UserNamePasswordValidator  ←[1]
{

    public override void Validate(string userName, string password)  ←[2]
    {
        if (userName == null || password == null)
        {
            throw new ArgumentNullException();
        }

        if (!(userName == "adwk" && password == "P@ssw0rd"))
        {
            throw new FaultException(
                "adwkRouting サービス :ユーザー名、パスワードが違います");
```

```
            }
        }
    }
```

この例の[1]のように、UserNamePasswordValidator から継承した派生クラスを定義し、[2]の Validate メソッドに任意の認証ロジックを実装します。このメソッドでは、認証すべきユーザー名とパスワードが引数として渡って来るので、その情報をもとに評価し、認証を認めない場合には、例外を発生させます。認証を成功させる場合には、そのままメソッドの呼び出しを終了します。

ここでの認証ロジックでは、簡単にするため、ユーザー名とパスワードは、それぞれ、"adwk" と "P@ssw0rd" に一致するか評価するようハードコーディングしましたが、任意の実装が可能であることが、この例から読み取れるでしょう。

クライアント側でのカスタム認証の構成

カスタム認証を行う際に、ユーザー名とパスワードは、プロキシのプロパティとして指定できます。たとえば、外部のパートナー企業のクライアント（DummyExternalApp プロジェクト）の、［予約照会］ボタンのイベントハンドラーには、次のように実装されています。

例 36. カスタム認証でのクレデンシャルの指定

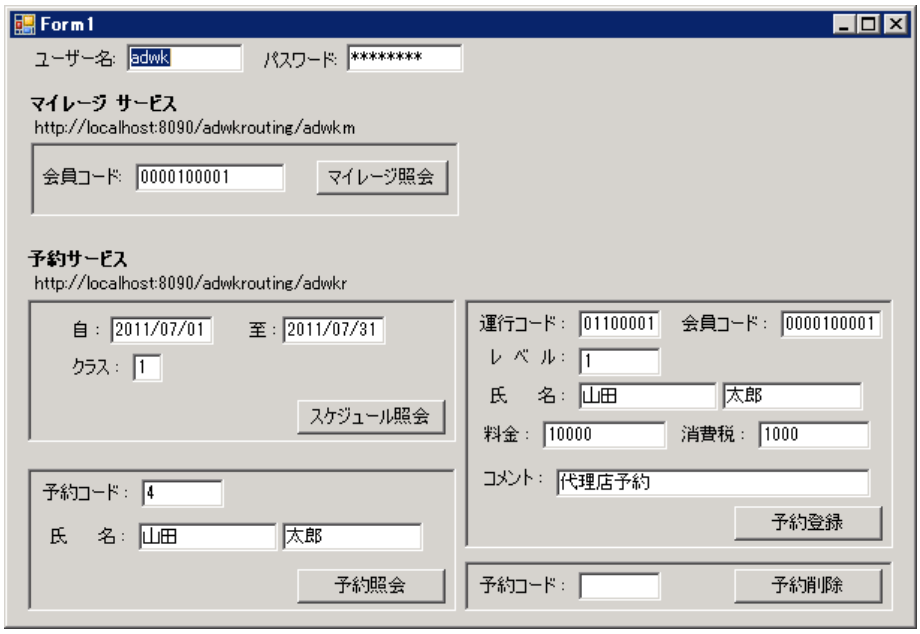
Form1.cs（DummyExternalApp プロジェクト）

```
    // 予約の照会
    private void btnMileageQuery_Click(object sender, EventArgs e)
    {
        var clientM = new MileageQueryClient(custBinding, epaM);  //←[1]

        try
        {
            clientM.ClientCredentials.UserName.UserName = txtSID.Text;    //←[2]
            clientM.ClientCredentials.UserName.Password = txtSpass.Text;  //←[3]
            clientM.ClientCredentials                                     //←[4]
                .ServiceCertificate.Authentication.CertificateValidationMode =
                    X509CertificateValidationMode.PeerOrChainTrust;

            MemberInfo minfo = new MemberInfo();
            bool resultM = clientM.QueryInfo1ByMember(out minfo, txtMCode.Text);
```

この例では、予約管理サービスに関するルーティングサービスにアクセスするため、[1]でプロキシのインスタンスを作成しています（変数名は clientM）。

そして、 [2]と[3]では、カスタム認証で使用するユーザー名とパスワードを指定しています。ここでは簡単にするため、この 2 つ値は、次図のクライアントアプリケーションの左上部のテキストボックス（txtSID.Text および txtSpass.Text）から取得しています。

図 40. 外部パートナー企業のクライアントアプリケーション



なお、次項で説明するサービス認証において、例 36 の[4]の 3 行は、サービス証明書の有効性を検証する方法を指定するものです。

サービス認証

すでに触れたように、このクライアント（DummyExternalApp プロジェクト）では、サービス証明書を使用して、サービスを認証しています。サービス側（AdwkRouting プロジェクト）では、例 34 の[11]のように、構成ファイルの中でサービス証明書を指定しています。

この証明書は、サービス側の環境の証明書ストアに配置された、秘密キーを持つ証明書です。つまり、このサービス固有の証明書です。認証の際には、その証明書の情報がクライアントに伝えられます（秘密キー自身は渡るけではありません）。

クライアント側には、その証明書が信頼できるものであると認識できるように予め構成しておく必要があります。典型的な方法としては、公開キー付き（秘密キーを含めない）サービスの証明書のファイルを、予めサービス側の組織から何らかの方法で受け取り、クライアント側の「信頼されたユーザー」という名前の証明書ストアに配置しておく方法です。クライアントがサービスを認証する際に、「信頼されたユーザー」に配置されいる証明書を、サービス証明書としてサービスが使用している場合、そのサービスは認証されます。例 36 の[4]の設定はそのためのものであり、[4]の 3 行の設定によって、「信頼されたユーザー」に配置された証明書が、サービス証明書として有効になります。

この仕組みを利用することで、有効なサービス証明書を持たないフィシングサイトへのアクセスを防止することができます。

3.7 暗号化と署名

すでに第 1 部でも触れたように、WCF の暗号化や署名には、HTTPS などのトランスポートレベルのものだけでなく、メッセージレベルでも行うことができます。前節で取り上げた外部パートナー企業からアクセスされるサービスでも、カスタム認証と併せて、メッセージの暗号化と署名を使用しています。ここでは、同じサンプル（図 39）について、改めてメッセージの暗号化や署名の構成方法を確認します。

サービス側での暗号化と署名の構成

前節の図 39 のルーティングサービスとクライアントとの間で、メッセージのやり取りに、暗号化と署名を使用するように構成されています。この暗号化と署名の構成も、Windows 認証やカスタム認証と同様に、バインディングごとの詳細の構成の中で行います。このルーティングサービスの構成では、例 34 に示した構成ファイルの中で行っています。改めて、該当する部分を以下に再掲します。（ソース内の番号は、便宜上、例 34 のままにしました。特に、暗号化と署名に関する記述は[6]と[11]です。）

例 37. メッセージの暗号化と署名の構成（抜粋）

App.config（AdwkRouting プロジェクト）

```
<bindings>
  <wsHttpBinding>
    <!-- 二つのエンドポイントの bindingConfiguration から参照 -->
    <binding name="Binding1">    ←[5]
      <security mode="Message">  ←[6]
        <message clientCredentialType="UserName" />  ←[7]
      </security>
    </binding>
  </wsHttpBinding>
</bindings>

<behaviors>
  <serviceBehaviors>

    <!-- ルーティングサービスのための Service Behavior の定義 -->
    <behavior name="routingConfiguration">  ←[8]

   (略)

      <serviceCredentials>

   (略)

        <!-- security mode="Message" かつ "UserName" の場合に必要 -->
        <serviceCertificate findValue="localhost"            ←[11]
                            storeLocation="LocalMachine"
                            storeName="My"
                            x509FindType="FindBySubjectName"/>
      </serviceCredentials>

    </behavior>
```

ここでは、WCF サービスとしてのルーティングサービスにおける暗号化と署名の構成を示していますが、他の WCF サービスの場合も、これと同様の構成（[6]および[11]）になります。

この例の[5]の<binding>要素ブロックには、このルーティングサービスのエンドポイントで使用する wsHttpBinding に関する詳細が構成されています。このブロックの中でセキュリティに関する構成を行います。実質的に、暗号化と署名を有効にしている箇所は、[6]の<security>要素に記述されたモード（mode 属性）の指定です。

バインディングの種類によって、このモードの指定が、どのように作用するかは異なります。ここで使用している wsHttpBinding の場合は、[6]のようにモードを "message" に設定することで、メッセージレベルのセキュリティが有効になり、メッセージの暗号化と署名が行われるようになります。また、暗号化と署名には、前節で触れた[11]のサービス証明書が使用されます。

これで、このバインディングを使用したエンドポイントに関して、メッセージの暗号化と署名が行われるようになります。

クライアント側での暗号化と署名の構成

クライアント側での暗号化と証明の構成も、サービス側と同様です。バインディングの詳細を構成する中で行います。

もっとも簡単な方法としては、サービスに対してクライアントの開発環境から「サービス参照の追加」を行って、プロキシと構成ファイルを生成する方法があります。（ただし、AWTS サンプルプログラムでは、ルーティンング サービス「AdwkRouting」とクライアント「DummyExternalApp」の間で、暗号化と署名を使用しており、ルーティング サービスに対してのサービス参照の追加ができないので、この方法を使用していません。）

AWTS サンプル プログラムの場合は、クライアントである DummyExternalApp プロジェクトに次のように構成されています。次の例 38 の[1]は、予約管理サービスにアクセスする際に使用される wsHttpBinding の詳細構成です。ここでも、[2]ては、<security>要素の mode 属性に、"message" と指定されています。この点は、前述のサービス側の構成と同じです。

例 38. クライアント側におけるメッセージの暗号化と署名の構成（抜粋）

App.config（DummyExternalApp プロジェクト）

```
<bindings>

    （略）

  <wsHttpBinding>
    <binding name="WSHttpBinding_IReservation">   ←[1]

        （略）

      <security mode="Message">  ←[2]
        <transport clientCredentialType="Windows"
                   proxyCredentialType="None" realm="" />
          <message clientCredentialType="Windows"
                   negotiateServiceCredential="true" algorithmSuite="Default" />
      </security>
    </binding>
```

証明書の構成

ここで改めて、証明書の構成について確認します。

すでに触れたように、AWTS サンプル プログラムのルーティング サービス「AdwkRouting」と、そのクライアント「DummyExternalApp」との間では、クライアントのカスタム認証やサービス認証、また、メッセージの署名と暗号化に関して、サービス証明書を使用しています。サービス証明書を使用するにあたり、典型的な構成方法のポイントをまとめると次のようになります。

（a）サービス側 ---- そのサービスを識別するための証明書（秘密キー付き）を「ローカルコンピューター（LocalMachine）」の「個人（My）」に格納する

（b）クライアント側 ---- サービス証明書がクライアントにとって有効になるように、その証明書（秘密キーは不要）を「現在のユーザー」の「信頼されたユーザー」に配置する

サービス側の秘密キー付証明書を作成するには、証明機関に作成を依頼したり、Windows の証明書サービスを使用したり、証明書作成ツール（Makecert.exe）を使用したりと、複数の選択肢があります。AWTS サンプル プログラムでは、Makecert.exe を使用しています。サンプルのインストール手順の中で、上記の（a）の構成にするため、次のコマンドを実行しています。

例 39. 証明書の作成（抜粋）

```
set SERVER_NAME=localhost
makecert.exe -sr LocalMachine -ss My -a sha1 -n CN=%SERVER_NAME% -sky exchange -pe
```

これで、証明書はサービス側マシンの証明書ストアに格納されるので、このあと、（b）のクライアントの構成をするために、この証明書の情報をもとに、証明書ファイルを配布する必要があります。

そのためには、MMC スナップインの「証明書」ツールや証明書マネージャー ツール（certmgr.exe）などを使用して、秘密キー無しの状態で証明書をエクスポートして、証明書ファイル（.cer ファイル）を作成し、これをクライアントに配布します。そして、クライアントも「証明書」ツールや certmgr.exe などを使用して、証明書ファイルを（b）のストアに格納します。クライアント側では、ユーザーごとの構成が必要なので、格納場所は「ローカルコンピューター」ではなく「現在のユーザー」です。

この AWTS サンプル プログラムでは、簡単にするため、certmgr.exe の以下のコマンドを使用して、ローカル コンピューターの証明書ストアに格納されている（a）のサービス証明書から、現在のユーザーの証明書ストアへ、つまり、前述の（b）の証明書ストアへ、単純にコピーしています。

例 40. 証明書のコピー（抜粋）

```
set SERVER_NAME=localhost
certmgr.exe -add -r LocalMachine -s My -c -n %SERVER_NAME%
   -r CurrentUser -s TrustedPeople
```

このコマンドによって、「信頼されたユーザー」（TrustedPeople）にコピーされます。なお、このコマンドでは、秘密キー付きでコピーされますが、本来であれば、秘密キーまで配布すべきではありません。秘密キー付きで証明書を配布すれば、その証明書を利用して、同じサービスに成りすまして、サービスを公開できてしまいます。

暗号化されたメッセージの確認

最後に、メッセージをログとして出力し、実際に暗号化されたメッセージの様子を確認してみましょう。ここでは、「2.4 メッセージ ログの基本操作」で取り上げたログの出力手順を、DummyExternalAppプロジェクトに対して行います。

1. 「2.4 メッセージ ログの基本操作」の 1 から 9 までの手順を、「AdwkMService プロジェクト」を「DummyExternalApp」プロジェクトに置き換えて行います。

これによって、DummyExternalApp が WCF サービスにアクセスする際のログが出力されるようになります。

註:「2.4 メッセージ ログの基本操作」の手順 5 の直後において、メッセージレベルが「Transport」となっている点に注意してください（図 41）。この表示部分をクリックすると、メッセージ レベル（サービス メッセージ）でのログを指定することもできます（図 42）。しかし、メッセージ レベルでのログの場合は、暗号化される前のメッセージをログに出力するため、暗号化の様子を確認できません。トランスポート レベル（トランスポート メッセージ）に設定しておけば、トランスポート レベルでやり取りされる暗号化した後のメッセージをログに出力できます。

図 41. ログ レベルは「Transport」に設定

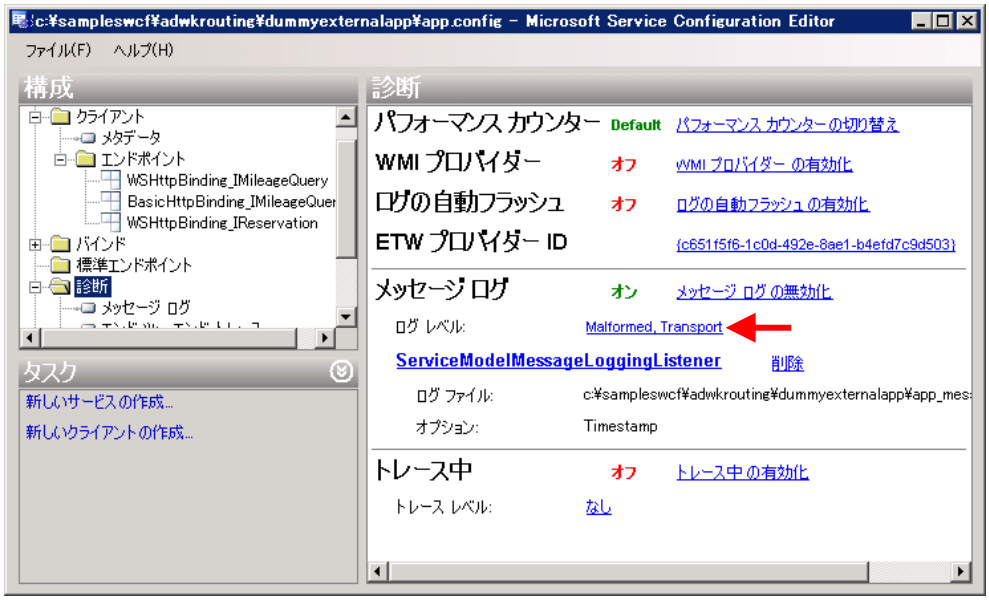
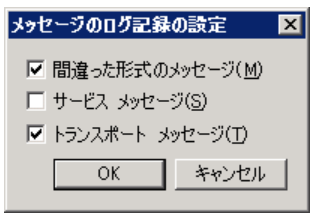


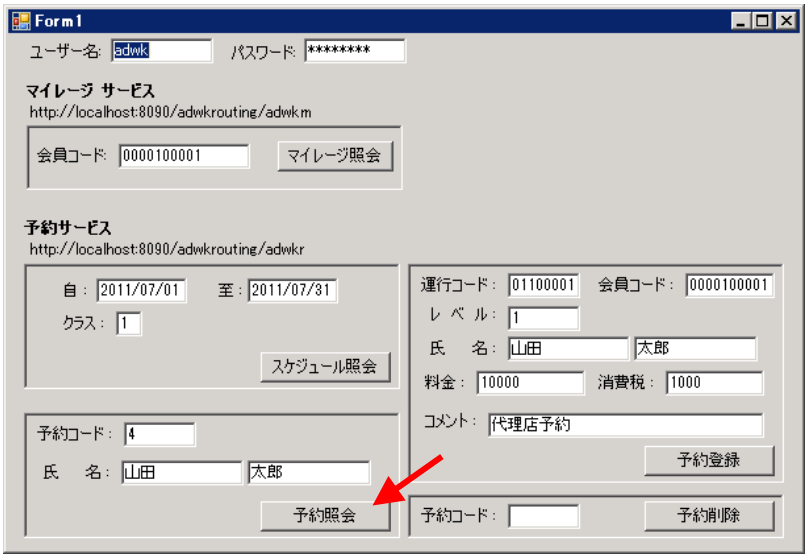
図 42. ログ レベルの設定ダイアログボックス



それでは、次の手順で実際にログを確認してみましょう。

2. サービス側で必要となる AdwkRouting、AdwkMService、および、AdwkRService の 3 つのプロジェクトを実行します。
3. クライアントである DummyExternalApp プロジェクトを実行します。
4. 図 43 のように、DummyExternalApp プロジェクトのフォームが表示されたら、テキストボックスの内容は初期状態のままにして、[予約照会] ボタンをクリックします。

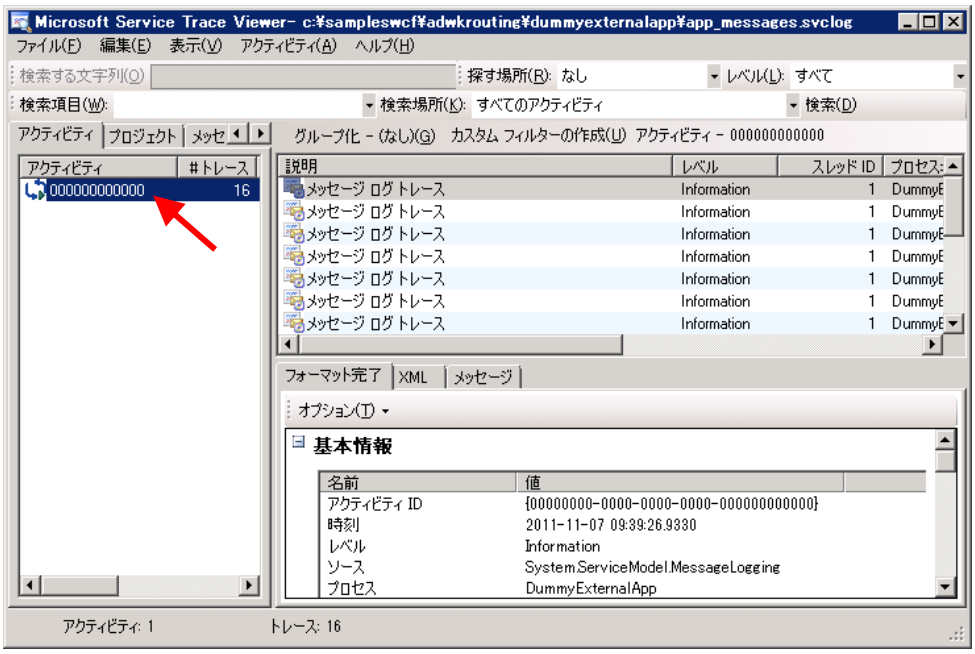
図 43. DummyExternalApp プロジェクトのクライアント アプリケーション



5. 予約に関する問い合わせが、ルーティングサービスを経由して呼び出され、その結果がメッセージボックスに表示されるので、閉じておきます。(この呼び出しによってメッセージ ログが作成されます。)
6. DummyExternalApp プロジェクトのフォームを閉じて、アプリケーションを終了します。
7. DummyExternalApp プロジェクトのフォルダーの中に、次のログ ファイルが作成されていることを確認します。(このファイルが作成される場所は、ソリューション フォルダーではなく、プロジェクト フォルダーです。)

   app_messages.svclog
8. このファイルをダブルクリックして、サービス トレース ビューアー (SvcTraceViewer.exe) で開きます。
9. サービス トレース ビューアーが起動したら、左側の [アクティビティ] タブにある 1 つ目のアクティビティをクリックして選択します。すると、右ペインには 16 個の「メッセージ ログ トレース」の行が表示されることを確認します。(このクライアントは、ルーティング サービスとの間で、サービスのアドレスを問い合わせるなど、複数のやり取りを行っています。)

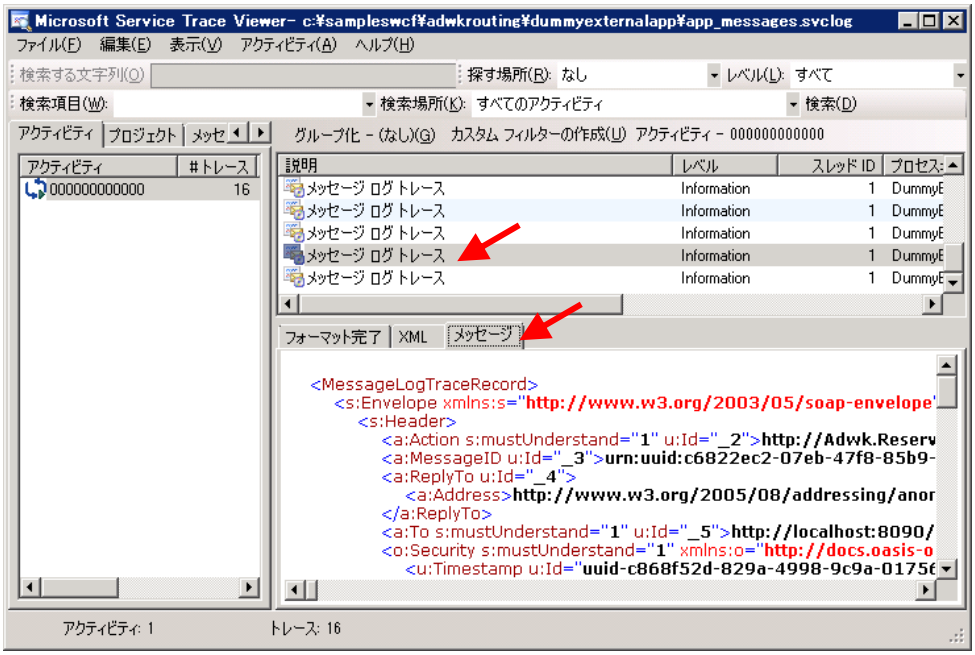
図 44. アクティビティを選択



今回の例では、WCF サービスに対して、予約の問い合わせをしています。予約の問い合わせは 16 個のログのうち、最後の 2 つが要求メッセージと応答メッセージのログです。

10. 右ペインで 15 行目の「メッセージ ログ トレース」をクリックし、右下半分のペインにある [メッセージ] タブをクリックします。まず、次のような形式で、このタブが表示されることを確認します。

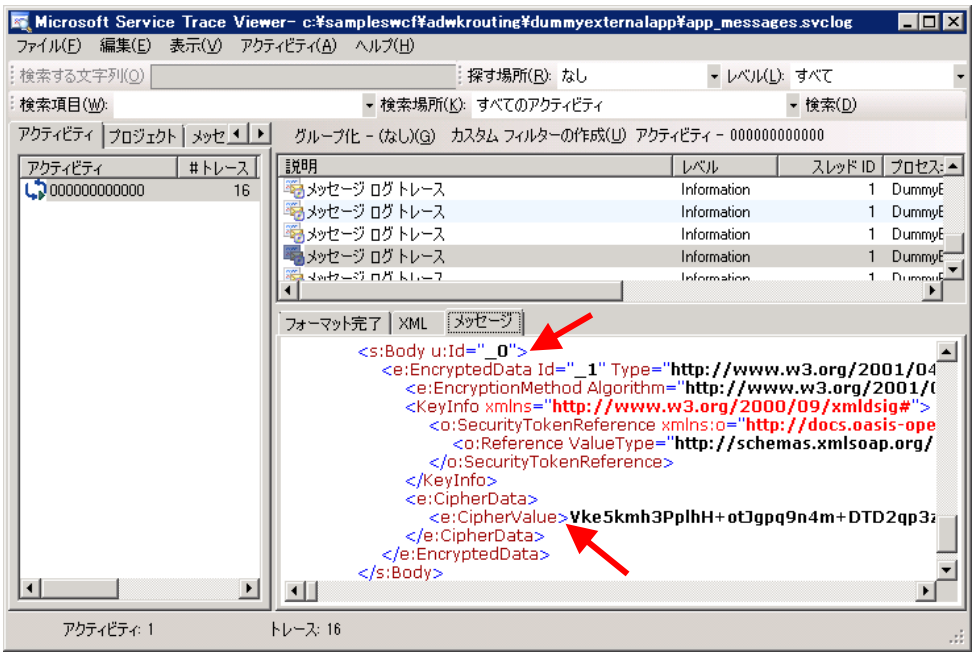
図 45. 予約の問い合わせのメッセージ ログ



11. 表示されたログの中で、<action>要素の値が以下に示すものであることを確認したのち、ログを下方にスクロールして、<body>要素内の、<CipherData>要素の中に暗号化されたデータがあることを確認します（紙面の都合で、途中改行しています）。

```
<a:Action s:mustUnderstand="1"  u:Id="_2">
http://Adwk.Reservation/IReservation/QueryReservation1ByRCode</a:Action>
```

図 46. 暗号化されたデータ



トランスポート レベルのデータ全体が暗号化されているわけではなく、<Body>要素などのタグはそのままに、データ本体が暗号化されていることが分かります。16 番目のメッセージログ（予約問い合わせの応答メッセージ）にも、暗号化されたデータが含まれます。余力がある方は、同様に確認してください。

12. 確認が済んだら、サービス トレース ビューアーを終了しておきます。
13. AdwkRouting、AdwkRService、および、AdwkMService の各サービスを終了します。

3.8 ルーティング サービス

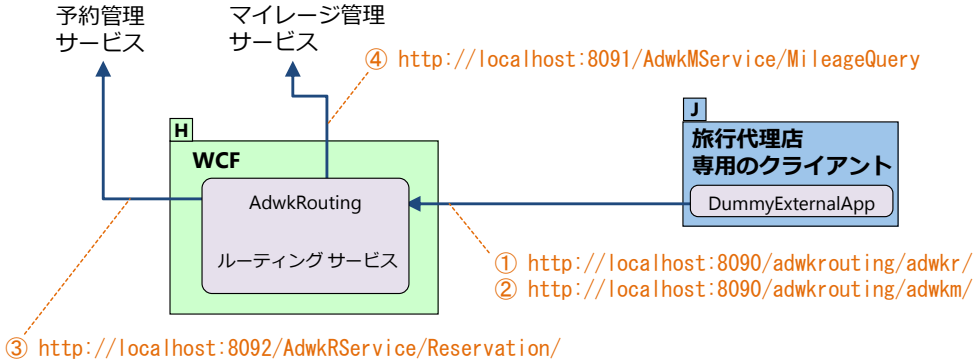
この節では、AdwkRouting プロジェクトのルーティング サービスの構成方法について、改めて取り上げます。まず、この AdwkRouting プロジェクトにおいて、どのようなルーティングを行っているか確認しましょう。

サンプルに見るルーティング サービスの構成

AWTS サンプル プログラムでは、すでに取り上げた図 8 のように、社内システムのサービスの一部に対して、外部のパートナー企業（図 8 の J）からアクセスできるようになっています。パートナー企業が必要とする WCF サービスは、フェリー予約のサービス（図 8 の D）と、マイレージ管理サービス（図 8 の F）の 2 つであり、これらは別々のサーバーに分散しています。しかし、ルーティング サービス（図 8 の H）のみを外部からのアクセスポイントとして一本化しており、外部のクライアントは、このアクセスポイントだけへの接続を配慮すればよく、サービス連携の運用管理を簡素化しています。

具体的には、図 47 のようにアクセスに使用する URL の違いによって振り分けています。

図 47. URL の違いによる振り分け



外部のクライアントがアクセスする URL は①または②であり、この 2 つの URL は末尾の仮想ディレクトリの部分に違いがあります（adwkr または adwkm）。①の URL の場合は、③の予約管理サービスに振り分けられます。また、②の URL の場合は、④のマイレージ管理サービスに振り分けられます。

また、実際の予約管理サービスには、以下に示す URL のエンドポイントを持った、登場手続きのサービスもありますが、このルーティングサービスでは、その URL へルーティングするように構成されていないため、実質的に登場手続きサービスを外部から隠すことができます。つまり、ルーティング サービスを使うことで、アクセスポイントを一か所に集約するだけでなく、必要なサービスだけを外部に公開することができ、必要がないものは隠ぺいできます。

```
http://localhost:8092/AdwkRService/Boarding/
```

それでは、これらをどのように構成するのか、具体的な構成方法を確認しましょう。

ルーティング サービスの構成

ルーティング サービスは、ルーティングの機能を備えた WCF サービスの一種です。このサービスの実装は、すでに WCF のクラスライブラリに定義済みの「RoutingService クラス」として用意されています。そのため、ルーティング サービスを構築するには、この RoutingService クラスをそのまま使うか、このクラスの派生クラスを定義して利用します。

AdwkRouting プロジェクトでは、クラスライブラリが提供する RoutingService クラスをそのまま利用して、ホスティングしています。ホスティング環境を構築する Main メソッドには、次のように記述されています。

例 41. ルーティング サービスのホスティング（抜粋）

Program.cs（AdwkRouting プロジェクト）

```
    var serviceHost = new ServiceHost(typeof(RoutingService));  //←[1]
    try
    {

        serviceHost.Open();  //←[2]
        Console.WriteLine("*** ルーティングサービスが開始しました。***");
        Console.WriteLine("Press <ENTER>...");
        Console.ReadLine();

        serviceHost.Close();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        serviceHost.Abort();
    }
```

サービス ホストやサービス クラスの扱いは、他の WCF サービスと同じです。[1]のように、ServiceHost クラスのインスタンスを作成し、その際のコンストラクターの引数に、ルーティング サービスの型情報を渡します。そして、[2]のように Open メソッドを呼び出せば、クライアントからの要求を受け付けるようになります。

また、構成ファイルには、ルーティング サービスの構成詳細が記述されています。

例 42. ルーティング サービスの構成（抜粋）

App.config（AdwkRouting プロジェクト）

```
<services>

  <!-- クラスライブラリ標準のRoutingServiceクラスを使用した
       ルーティングサービスクラスの構成 -->
  <service behaviorConfiguration="routingConfiguration"        ←[1]
           name="System.ServiceModel.Routing.RoutingService">  ←[2]

    <!-- ルーティングサービスへのアクセスするための
         エンドポイントの定義(予約サービス) -->
    <endpoint address="adwkr" binding="wsHttpBinding" name="adwkrEndpoint"  ←[3]
              contract="System.ServiceModel.Routing.IRequestReplyRouter"
              bindingConfiguration="Binding1"/>

    <!-- ルーティングサービスへのアクセスするための
         エンドポイントの定義(マイレージサービス) -->
    <endpoint address="adwkm" binding="wsHttpBinding" name="adwkmEndpoint"  ←[4]
              contract="System.ServiceModel.Routing.IRequestReplyRouter"
              bindingConfiguration="Binding1"/>

  </service>
</services>

<behaviors>
  <serviceBehaviors>
    <!-- ルーティングサービスのための Service Behavior の定義 -->
    <behavior name="routingConfiguration">       ←[5]

      <!-- ルーティングに使用する filterTable
           フィルター条件とルーティング先が定義されている -->
      <routing filterTableName="filterTable1"/>  ←[6]
```

```
    </behavior>
  </serviceBehaviors>
</behaviors>

<!-- ルーティング先に使用する、通常のクライアントエンドポイントの定義 -->
<client>
  <endpoint address="http://localhost:8092/AdwkRService/Reservation/"
      binding="wsHttpBinding" contract="*" name="Binding_IReservation">  ←[7]
  </endpoint>
  <endpoint address="http://localhost:8091/AdwkMService/MileageQuery/"
      binding="wsHttpBinding" contract="*" name="Binding_IMileageQuery"> ←[8]
  </endpoint>
</client>

<!-- フィルター条件に基づく、ルーティング先の定義 -->
<routing>
  <!-- 各フィルタ条件の定義(フィルタ名の定義) -->
  <filters>
    <filter name="PrefixAddressFilterR"             ←[9]
            filterType="PrefixEndpointAddress"
            filterData="http://localhost:8090/adwkrouting/adwkr/"/> ←[10]
    <filter name="PrefixAddressFilterM"
            filterType="PrefixEndpointAddress"      ←[11]
            filterData="http://localhost:8090/adwkrouting/adwkm/"/> ←[12]
  </filters>
  <!-- 各フィルター名と
       ルーティング先クライアントエンドポイント名との対応付け -->
  <filterTables>
    <filterTable name="filterTable1">          ←[13]
      <add filterName="PrefixAddressFilterR"    ←[14]
          endpointName="Binding_IReservation" priority="1"/>  ←[15]
      <add filterName="PrefixAddressFilterM"    ←[16]
          endpointName="Binding_IMileageQuery" priority="1"/> ←[17]
    </filterTable>
  </filterTables>
</routing>
```

上記の[1]の<service>要素ブロックが、ルーティング サービスに関する構成を行うブロックであり、通常の WCF サービスとしての構成も、このブロックに記述できます。このブロックのルーティング サービスとしての固有な点は、[2]の name 属性にルーティング サービスのクラス名 RoutingService が記述されている点です。

[3]と[4]は、ルーティング サービスとしてのエンドポイントであり、クライアントが直接アクセスするアクセスポイントです。contract 属性には、ルーティング サービス固有のインターフェイスである IRequestReplyRouter が記述されています。ただし、これにアクセスするクライアントは、最終目的のサービスのコントラクトを使用してアクセスできます。

[1]の behaviorConfiguration 属性で参照している [5]のサービス ビヘイビアでは、[6]の<routing>要素を使用して、実質的にどのように転送するか定義しています。[6]の filterTableName 属性で参照している "filterTable1" という名前のテーブルには、特定の条件を満たすクライアントからの要求を、どのサービスへ転送するかという対応付けの一覧が定義されています。

具体的なテーブル（"filterTable1"）は、[13]の<filterTable>要素ブロックです。これは「フィルターテーブル」と呼ばれています。この例のフィルター テーブルでは、[14]と[16]に 2 つの定義があります。1 つめの定義である[14]では、満たすべき条件の名前（条件に基づきフィルタリングを行うフィルター名）が指定され、[15]では、その条件に対応する転送先のエンドポイントの名前が定義されています。[16]と[17]も同様です。

[14]に指定された条件（フィルター名が "PrefixAddressFilterR" のもの）は、[9]の<filter>要素に定義されています。このフィルターでは、[10]にあるように、クライアントが指定したアドレスの先頭部分が [10] であることが条件です。

一方、[15]に指定された転送先のエンドポイント（名前が "Binding_IReservation" のもの）は、[7]に定義されています。[7]は、予約管理サービスのエンドポイントです。

よって、[10]のアドレスを使用してクライアントがアクセスすると、[7]の予約管理サービスのエンドポイントに転送されます。

同様に、[16]のテーブルの定義によって、[16]のフィルター名を満たす場合、[17]のエンドポイントに転送に転送されます。つまり、[12]のアドレスを用いてアクセスすると、[8]のエンドポイントへ転送されます。

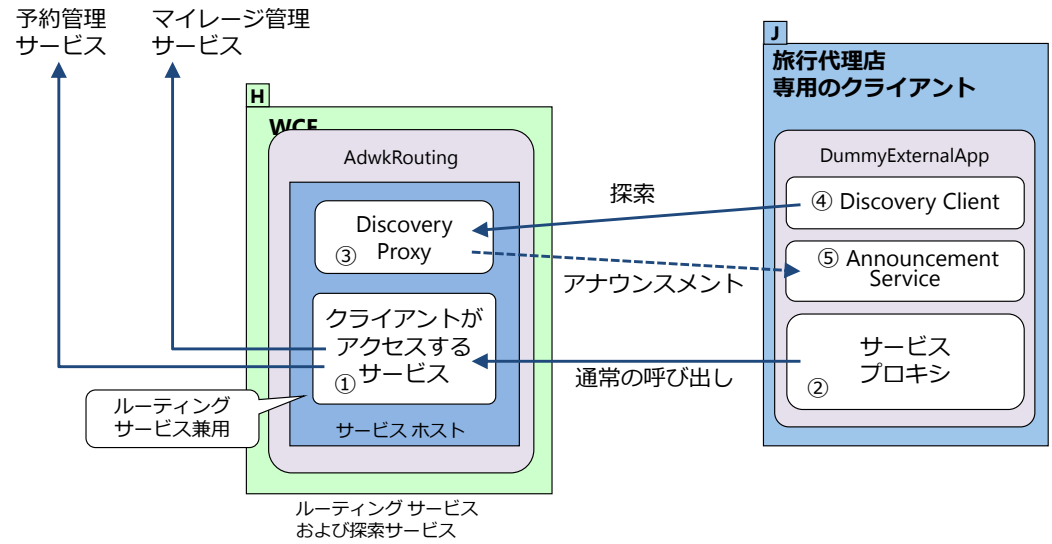
3.9 探索

この節では、WCF クライアントが目的のサービスを見つけるために使用する「探索」（Discovery）について改めて取り上げます。WCF におけるサービスの探索は、WS-Discovery に基づくもので、相互運用性を備えています。また、構成ファイルやクラスライブラリを用いて、効率よく探索を実装することができます。ここでは、AdwkRouting プロジェクトを題材にして、WCF における、より具体的な探索の仕組みや機能、実装方法について確認しましょう。

サンプルに見る探索の構成

AdwkRouting プロジェクトに実装された探索の構成を次に示します。

図 48. AdwkRouting における探索の構成



この図 48 では、ルーティング サービスが①であり、そのクライアントが②です。しかし、ルーティングと探索はそれぞれ独立した個別の機能であり、ここでは、①がルーティング サービスであることは重要ではありません。①と②は、一般的な WCF サービスと WCF クライアントの構成の 1 つと考えてください。

そして、②の WCF クライアントから、①の WCF サービスを見つけ出せるよう探索可能に構成した点が、この図のポイントです。

この例では、サービス側（AdwkRouting）に探索機能を実装して、クライアント側（DummyExternalApp）が目的のサービスの URL を見つけることや、目的のサービスがオンラインか否かを認識できるように構成されています。

サービス側に必要なのは、③の Discovery Proxy と呼ばれる実装です。Discovery Proxy は、サービスの情報を持つレポジトリとしての役割を持ち、クライアント側の④の Discovery Client からの問い合わせに対して、サービスに関するメタデータを提供できます。④のクライアントからの問い合わせを受け付けるには、Discovery Proxy 側に、探索問い合わせのエンドポイント（Discovery Endpoint）を用意する必要があります。

Discovery Endpoint では、探索用に特定の URL を公開して、要求/応答形式のやり取りが可能であるほか、UDP を使用してクライアントから問い合わせを発信し、サービスがその要求をリッスンして、要求に答えることもできます。その場合、クライアントは、明示的に探索を問い合わせるホスト名を意識する必要はありません。

また、③の Discovery Endpoint では、①のサービスがオンライン、もしくはオフラインになったことを検知して、クライアントに通知（Announcement）を送信できます。このときは、サービス側に Announcement Endpoint と呼ばれるエンドポイントを用意します。このエンドポイントに標準で用意されているプロトコルも UDP です。よって、サービス側はクライアントの正確なアドレスを知らなくとも、複数のクライアントにマルチキャストを使用して通知することができます。アナウンスメントをクライアントがリッスンするには、⑤に示す Announcement Service と呼ばれる実装がクライアントに必要です。

全体的な構成を確認したところで、サンプルのコードの中で、実際の構成例を見ていきましょう。

サービス側における探索とアナウンスメントの構成

クライアントからの問い合わせに答える「探索」と、サービスから必要に応じて通知する「アナウンスメント」の構成は、お互いに非依存であり、同時に構成する必要はありません。それぞれ、構成ファイでの簡単な記述で実装することができます。以下は、AdwkRouting プロジェクトの構成ファイルには、次のように記述されています。

例 43. 探索とアナウンスメントの構成

App.config（AdwkRouting プロジェクト）

```
<services>

  <!-- クラスライブラリ標準のRoutingServiceクラスを使用した
       ルーティングサービスクラスの構成 -->
  <service behaviorConfiguration="routingConfiguration"
           name="System.ServiceModel.Routing.RoutingService">
```

```
    <host>
      <baseAddresses>
        <add baseAddress="http://localhost:8090/adwkrouting"/>  ←[1]
      </baseAddresses>
    </host>

    <!-- 探索用のエンドポイント -->
    <!-- wsHttpBinding使用の場合 -->
    <endpoint address="/mydisco" binding="wsHttpBinding"  ←[2]
              kind="discoveryEndpoint" />

  </service>
</services>

<behaviors>
  <serviceBehaviors>
    <!-- ルーティングサービスのための Service Behavior の定義 -->
    <behavior name="routingConfiguration">

      <!-- 探索 -->
      <serviceDiscovery>  ←[3]
        <!-- アナウンスメントの指定 -->
        <announcementEndpoints>  ←[4]
          <endpoint kind="udpAnnouncementEndpoint" />   ←[5]
        </announcementEndpoints>
      </serviceDiscovery>

    </behavior>

  </serviceBehaviors>

</behaviors>
```

まず、[3]の<serviceDiscovery>要素ブロックは、「探索」を使用するにしても、「アナウンスメント」を使用するにしても、共通して必要となります。サービス ビヘイビアの中に、この要素ブロックを追加する必要があります。

このほか、「探索」のために必要なのは、[2]の Discovery Endpoint の定義です。[2]のエンドポイントのアドレス（ここでは address="/mydicso"）は、クライアントが探索の問い合わせをする際に使用されます。 [1]にベースアドレスが定義されているので、探索のためのエンドポイントは、次のようになります。この例の探索は、特定の URL を使用した、クライアントからの要求/応答形式の例です。

```
http://localhost:8090/adwkrouting/mydisco
```

この[2]のエンドポイントで重要な設定は、[2]の次行の kind 属性が "discoveryEndpoint" と指定されている点です。これによって、このエンドポイントが Discovery Endpoint になります。

「アナウンスメント」で必要な記述は、[4]の<announcementEndpoins>要素ブロックです。さらに、このブロックの中に、[5]のように具体的な Announcement Endpoint を定義します。標準で用意されているのは、[5]の kind 属性が "udpAnnouncementEndpoint" であり、サービスからクライアントに向けて、UDP で通知します。

なお、「探索」のみ使用する場合は、[3]の要素ブロックの中身は空のままになります。

クライアント側における探索とアナウンスメントの構成

図 48 のとおり、クライアント側から探索を行うには、④の Discovery Client を使用します。この実装は、クラス ライブラリに DiscoveryClient クラスとして用意されています。クライアントである DummyExternalApp プロジェクトでの使用例を以下に示します。

例 44. DiscoveryClient の使用(抜粋)

Form1.cs（DummyExternalApp プロジェクト）

```
void FindService()  //←[1]
{
    // wsHttpBinding を使用するDiscoveryClient の場合
    var endpoint1 = new DiscoveryEndpoint();  //←[2]
    endpoint1.Address =                       //←[3]
        new EndpointAddress("http://localhost:8090/adwkrouting/mydisco");
    endpoint1.Binding = new WSHttpBinding();  //←[4]
    var discoveryClient = new DiscoveryClient(endpoint1);  //←[5]

    // サービスの探索
    FindResponse routingServices = null;
    try
    {
        routingServices = discoveryClient.Find(              //←[6]
            new FindCriteria(typeof(IRequestReplyRouter)));  //←[7]
    }
    catch
    {
    }

    if (routingServices == null ||
        routingServices.Endpoints.Count == 0) //←[8]
    {
        //MessageBox.Show("*** サービスが見つかりませんでした ***");
        labelMAddress.Text = offlineDisplay;
        labelRAddress.Text = offlineDisplay;
        UpdateUI(); // ボタンなどの UI更新
        return;
    }
    else
    {
        foreach (EndpointDiscoveryMetadata endpoint  //←[9]
            in routingServices.Endpoints)
        {
            switch (endpoint.ListenUris[0].AbsolutePath)
            {
                case mserviceURI: //←[10]
                    epaM = endpoint.Address; //マイレージサービスの絶対アドレス ←[11]
                    labelMAddress.Text = epaM.ToString();
                    isMServiceAvailable = true;
                    break;
                case rserviceURI:
                    epaR = endpoint.Address; //予約サービスの絶対アドレス
                    labelRAddress.Text = epaR.ToString();
                    isRServiceAvailable = true;
                    break;
                default:
                    break;
```

```
            }
        }

        UpdateUI(); // ボタンなどの UI更新
    }
}
```

[1]の FindService メソッドの中で、探索を行い、目的とするサービスの現在の URL を求めています。この FindService メソッドは、フォーム（Form1）がロードされる際、Load イベントハンドラーの中から呼び出されるので、アプリケーションの起動時に、サービスの現状のアドレスを求めるようになっています。

探索には、[5]の DiscoveryClient オブジェクトを使用し、そのコンストラクターの引数には、アクセス先となる Discovery Endpoint（変数 endpoin1）の情報を渡します。このエンドポイントは、[2]に示すように、DiscoveryEndpoint オブジェクトです。[3]では、このエンドポイントにサービス側のエンドポイントのアドレスを設定し、[4]ではサービスとの間で使用するバインディングを指定しています。

実際の探索の問い合わせは、[6]のように、DiscoveryClient オブジェクトの Find メソッドを呼び出します。その際の引数には、抽出条件を指定することができ、ここでは[7]のように、特定のインターフェイスを持つサービス（IRequestReplyRouter インターフェイスを持つサービス）を探索しています。これによって、今回の例では、ルーティング サービスが見つかります。

[8]では、探索結果にエンドポイントが含まれるか（0 個でないか）確認し、見つかった場合は、[9]のループを通じて、特定の URI を含むものを見つけています。[10]の場合は、サービスの URI に「/adwkrouting/adwkm」が含まれるものを調べ、見つかった場合は、[11]で絶対アドレスを求めています。（変数 epaM に退避して、のちほど、プロキシ経由のアクセスに使用します。）

なお、ここでの探索の抽出条件は、サービスの URI を使用しているため、対象となるサービスの URI（"/adwkrouting/adwkm"の部分）は、常に固定であることが前提です（サービス側のホスト名やポート番号は変更可能です）。

アナウンスメントに関しては、図 48 の⑤のとおり、クライアント側に Announcement Service を用意して、サービス側からの通知をリッスンする必要があります。Announcement Service も、クラスライブラリに AnnouncementService クラスとして用意されています。次に例を示します。

例 45. AnnouncementService の使用（抜粋）

Form1.cs（DummyExternalApp プロジェクト）

```
// アナウンスメントのリッスン用オブジェクト
private ServiceHost announcementServiceHost;    //←[1]
private AnnouncementService announcementService;   //←[2]

// アナウンスメントのリッスンの準備
private void SetupListen()  //←[3]
{
    // AnnouncementService のインスタンス作成
    announcementService = new AnnouncementService();  //←[4]

    // イベントハンドラーの設定
    announcementService.OnlineAnnouncementReceived += OnOnline;    //←[5]
    announcementService.OfflineAnnouncementReceived += OnOffline;  //←[6]

    // AnnouncementService の実行
```

```
    announcementServiceHost = new ServiceHost(announcementService); //←[7]
    announcementServiceHost.AddServiceEndpoint(  //←[8]
        new UdpAnnouncementEndpoint());
    announcementServiceHost.Open();  //←[9]
}

// アナウンスメントのリッスンの終了
private void CloseListen()  //←[10]
{
    if(announcementServiceHost != null)
        announcementServiceHost.Close();  //←[11]
}

// サービスがオンラインになった際
// (エンドポイントの数だけ呼び出される)
private void OnOnline(object sender, AnnouncementEventArgs e)  //←[12]
{
    switch (e.EndpointDiscoveryMetadata.Address.Uri.AbsolutePath)  //←[13]
    {
        case mserviceURI:  //←[14]
            epaM = e.EndpointDiscoveryMetadata.Address; //←[15]
            labelMAddress.Text = epaM.ToString();
            isMServiceAvailable = true;
            break;
        case rserviceURI:
            epaR = e.EndpointDiscoveryMetadata.Address;
            labelRAddress.Text = epaR.ToString();
            isRServiceAvailable = true;
            break;
        default:
            break;
    }

    // ボタンなどの UI更新
    UpdateUI();
}

// サービスがオフラインになった際
// (エンドポイントの数だけ呼び出される)
private void OnOffline(object sender, AnnouncementEventArgs e)  //←[16]
{
    switch (e.EndpointDiscoveryMetadata.Address.Uri.AbsolutePath)  //←[17]
    {
        case mserviceURI:  //←[18]
            epaM = null;
            labelMAddress.Text = offlineDisplay;  //←[19]
            isMServiceAvailable = false;
            break;
        case rserviceURI:
            epaR = null;
            labelRAddress.Text = offlineDisplay;
            isRServiceAvailable = false;
            break;
        default:
            break;
    }
```

```
    // ボタンなどの UI更新
    UpdateUI();
}
```

クライアントがアナウンスメントをリッスンするには、[2]の AnnouncementService のほか、[1]の ServiceHost クラスも必要です。

このサンプルでは、[3]の SetupListen メソッドでアナウンスメントのリッスンを開始し、[10]の CloseListen メソッドでリッスンを終了します。この 2 つのメソッドは、それぞれ、Load イベントハンドラー、FormClosing イベントハンドラーで呼び出されます。結局のところ、アプリケーションの開始から終了まで、アナウンスメントのリッスンを行うようになります。

[3]のリッスンを開始するメソッドでは、まず、[4]で AnnouncementService インスタンスを作成します。このオブジェクトが、サービスのオンラインやオフラインに切り替わった際にイベントを発生するので、[5]と[6]で、このオブジェクトに対して、イベントハンドラーを設定しています。[5]の OnlineAnnouncementReceived イベントは、サービス開始時のアナウンスメントをクライアントが受信すると発生するものであり、[6]の OfflineAnnouncementReceived は、サービス終了時のアナウンスメントをクライアントが受信した場合に発生します。

さらに、[7]のようにサービスホスト（ServiceHost）のインスタンス生成が必要であり、そのコンストラクターの引数（通常はサービスクラスを渡す引数）へ、AnnouncementService オブジェクト（変数 announcementService）を渡します。さらに[8]では、クライアント側の AnnouncementService のエンドポイントとして、UpdAnnoucementEndpoint オブジェクトを追加します。この結果、UDP のリッスンができるようになります。

そして、[9]の Open メソッド呼び出しによって、実際のリッスンが開始します。

また、[10]の CloseListen メソッドは逆にリッスンを終了ためのメソッドであり、[11]でサービスホストの Close メソッドを呼び出しています。

[12]の OnOnline メソッドは、サービス起動時のアナウンスメントによって駆動するイベントハンドラーであり、利用可能になったエンドポイントの数だけ、複数回呼び出されます。どのエンドポイントがオンラインになったかは、イベントハンドラーの引数 e で判別できます。[13]の分岐では、この引数 e をもとに、オンラインになったエンドポイントを求め、URI の値によって、求めるアドレスを格納する変数を変えています。たとえば、[14]ではマイレージ管理サービスの URI（変数 mserviceURI）であるか確認し、そうであれば、[15]でその絶対アドレスを変数 epaM に退避しています。

逆にオフラインになった場合は、[16]がエンドポイントの数だけ、複数回呼び出されます。同様に[17]の switch 文のように引数 e の値を調べて、URI の値によって分岐します。[18]では、マイレージ管理サービスの URI であるかを調べて、そうであるなら、[19]のように該当するラベルに "(offline)" （定数 offlineDisplay）と表示します。

・検索とアナウンスメントの動作確認

ここで、例 44（探索）や例 45（アナウンスメント）のコードを実行し、クライアントでの実際の効果を確認してみましょう。

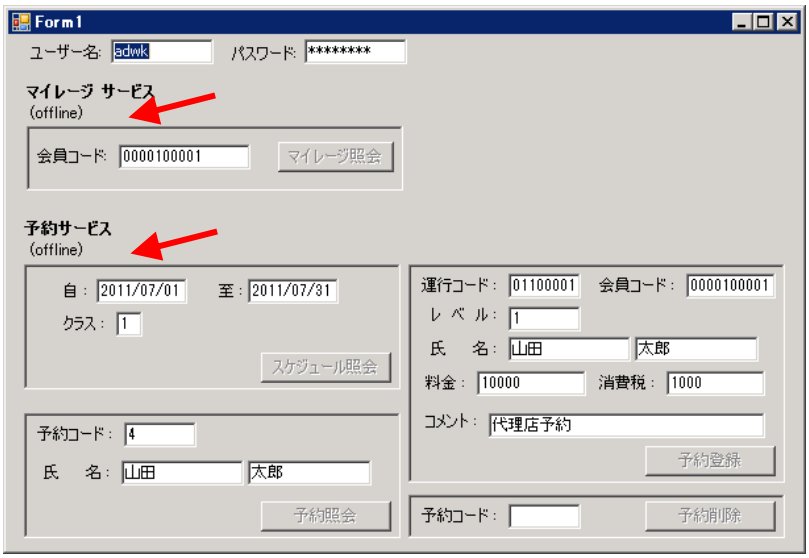
クライアントである DummyExternalApp では、起動時に探索を使用して、目的のサービスの最新のアドレスを取得しています。まずは以下の手順で、サービスを起動してない場合と、起動してある場合とで、クライアントの様子の違いを確認してみます。

1. AdwkRouting、AdwkMService、および、AdwkRService の 3 つのプロジェクトを実行している場合は、これらを終了し、実行していない状態にします。
2. クライアントである DummyExternalApp プロジェクトを実行します。

DummyExternalApp プロジェクトのフォーム Form1 は、ロード時に例 44 の FindService メソッドを実行します。しかし、今回の実行では、例 44 の[5]の探索（Find メソッド）を実行した際に、AdwkRouting 側の受付口となる DiscoveryEndpoint が起動していないため、探索に失敗し例外が発生します。そのため、[7]の直下の catch ブロックに制御が移り、探索結果を変数 routingServices に得られず、この変数は null のままです。その結果、[8]の if ブロックの中が実行され、アドレス欄の 2 つのラベル（labelMAdress と labelRAdress）には、定数 offlineDisplay の値 "(offline)" が設定されます。

3. フォーム Form1 が開き、次図のアドレス用のラベルには、"(offline)" と表示されることを確認します。

図 47. サービスがオフライン（利用不可）の状態



4. 確認が済んだら、DummyExternalApp プロジェクトのクライアント（フォーム Form1）を閉じて終了します。

次に、サービスが起動していた場合の様子を確認します。

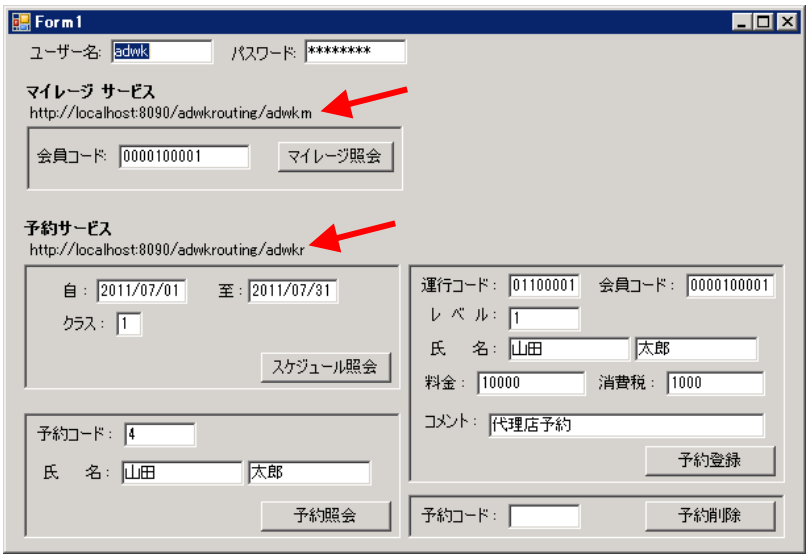
5. サービスに必要となる AdwkRouting、AdwkMService、および、AdwkRService の 3 つのプロジェクトを実行します。

6. クライアントである DummyExternalApp プロジェクトを実行します。

再び、フォーム Form1 のロード時に、例 44 の FindService メソッドが呼び出されます。今度は、サービス側の DiscoveryEndpoint が利用できるので、正しく探索できます。

7. フォーム Form1 が開き、次図のようにアドレス用のラベルには、それぞれのアクセスすべきエンドポイントのアドレス（ルーティング サービスに対するアクセスポイント）が表示されます。

図 48. サービスがオンライン（利用可能）の状態



8. フォーム上の［マイレージ照会］ボタンや［予約照会］ボタンなどをクリックし、エラーにならずに、メッセージボックスに照会結果を表示することを確認します。

このまま終了せずに、引き続き、アナウンスメントの検証を行います。

AdwkRouting プロジェクトのサービスの起動や終了時には、UDP を使用して、その旨のアナウンスメントが送信されます。これに呼応してクライアント側では、例 45 の 2 つのイベントハンドラー（OnOnline メソッド、および、OnOffline メソッド）が駆動します。

9. AdwkRouting のコンソールで、［Enter］キーを押し、AdwkRouting プロジェクトのサービスを正常に終了させます。（［Ctrl］＋［C］で強制終了しないでください。アナウンスメントが送信されません。）
10. すると、フォーム Form1 では、サービスがオフラインになった通知を受け取り、前述の図 47 のように、再び "(offline)" と表示されることを確認します。

このあとは次の要領で、ルーティングサービスが公開するサービスのエンドポイントを若干変更して、起動してみましょう。

11. AdwkRouting プロジェクトの構成ファイルの中で、予約管理サービスの URL を次のように変更します（太字部分）。

例 46. サービスのエンドポイントの URL を変更する

App.config（AdwkRouting プロジェクト）

```
  <host>
    <baseAddresses>
      <add baseAddress="http://localhost:8090/adwkrouting"/>  ←[1]
    </baseAddresses>
  </host>

  <!-- ルーティングサービスへアクセスするための
       エンドポイントの定義(予約サービス) -->
  <endpoint address="http://localhost:10000/adwkrouting/adwkr" ←[2]
            binding="wsHttpBinding" name="adwkrEndpoint"
            contract="System.ServiceModel.Routing.IRequestReplyRouter"
            bindingConfiguration="Binding1"/>

  <!-- ルーティングサービスへアクセスするための
       エンドポイントの定義(マイレージサービス) -->
  <endpoint address="adwkm" binding="wsHttpBinding" name="adwkmEndpoint"
            contract="System.ServiceModel.Routing.IRequestReplyRouter"
            bindingConfiguration="Binding1"/>

  <!-- 探索用のエンドポイント -->
  <!-- wsHttpBinding使用の場合 -->
  <endpoint address="/mydisco" binding="wsHttpBinding"  ←[3]
            kind="discoveryEndpoint" />
```

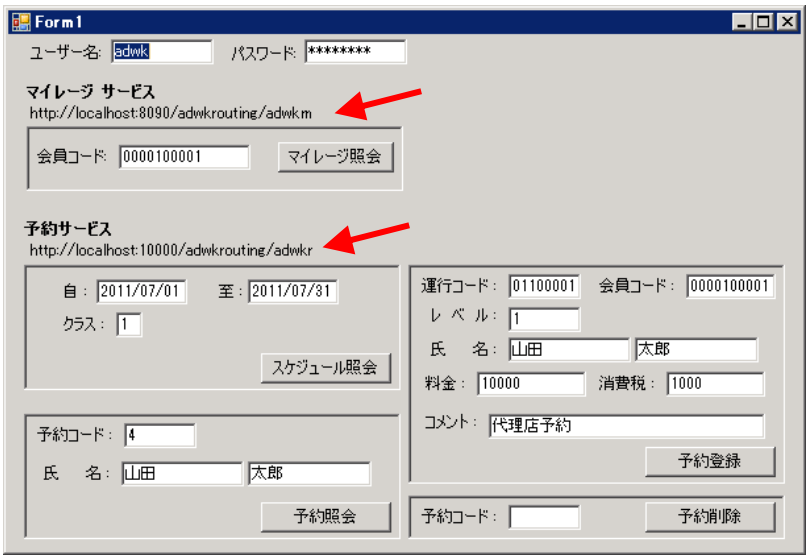
上記の[2]では、絶対アドレスとして address 属性を記述したので、[1]のベースアドレスは使用しなくなります。ただし、[3]の Discovery Endpoint のアドレスまでは変更しないでください。今回のクライアント側は、wsHttpBinding を用いて、[3]のアドレスに決め打ちで探索の問い合わせを行うので、[3]のアドレスを変えてしまうと、探索自体を行うことができなくなります。（仮に、UDP を使用する場合は、この点は問題ありません。）

12. 再び、AdwkRouting プロジェクトを実行します。

これで、サービスが起動した旨のアナウンスメントが、クライアント側に伝わりますが、単にオンラインになったことを通知するだけでなく、有効な最新の URL も伝えることができます。例 45 の[16]にあるオンライン時のイベントハンドラー（OnOnline メソッド）でも、引数 e からその時点の有効なアドレスを取得し、変数 epaM、または、変数 epaR に退避しています。

13. すると、フォーム Form1 の 2 つのアドレスのラベルには、再び、エンドポイントのアドレスが表示されます。特に予約サービスでは、変更後のアドレス（例 46 の[2]）になっていることを確認します。

図 49. 再びオンラインになり、最新のアドレスが反映



14. フォーム上の［マイレージ照会］ボタンや［予約照会］ボタンなどをクリックし、エラーにならずに、メッセージボックスに照会結果を表示することを確認します。

15. 確認が済んだら、AdwkRouting、AdwkMService、AdwkRService、および、DummyExternalApp の各プロジェクトを終了しておきます。

註: ここでは、探索やアナウンスメントの実装例の 1 つを照会しました。実装方法には、ほかにもバリエーションがあります。たとえば、サービス側の探索の構成も、構成ファイルに記述するのではなく、コードで実装することもできます。また、探索でも UDP を使用できます。探索やアナウンスメントの詳細は、以下のアドレスも参照してみてください。
http://msdn.microsoft.com/ja-jp/library/dd456782.aspx WCF Discovery

第 4 章 クライアント側の様々な実装とデータ操作

この章では、WCF クライアントに関する様々な実装方法を取り上げます。AWTS サンプルプログラムの中で利用されている各種実装方法について確認していきます。

なお、サービスで扱うデータの種類やその操作方法については、クライアントとのやり取りが関係するので、この点も本章で取り上げます。

4.1 クライアントでのチャネルファクトリの利用

第 1 部では、WCF クライアントを実装するいつくかの選択肢を挙げ、それぞれの特徴について概説しました。また、第 2 章では、サービス参照の追加を行って生成させた、プロキシを利用したクライアントの実装方法について取り上げました。この節では、主な選択肢の 1 つであるチャネル ファクトリを使用した実装方法を説明します。

既に第 3 章で取り上げたように、外部のパートナー企業に相当する DummyExternalApp プロジェクトは、AdwkRouting プロジェクトのルーティング サービスにアクセスしています。この両者の関係も、WCF クライアントと WCF サービスとの関係であり、クライアントである DummyExternalApp プロジェクトは、プロキシを使用して、サービスに相当する AdwkRouting にアクセスしていました。

ここでは練習として、現状のプロキシを使用している実装コードの 1 つを、チャネル ファクトリを使用する実装に変更してみます。

> **註:** 既に触れたように、DummyExternalApp のプロキシは、ルーティング サービスへの「サービス参照の追加」を行って作成したわけではありません。というのは、ルーティング サービス自体は、クライアントが目的とするサービスのメタデータを直接提供しないからです。
> この場合のプロキシを作成する方法としては、サービスを提供する側である Adventure Works Transport Services 社（AWTS 社）が、外部パートナー企業に対して、サービスを記述した WSDL ファイルを何らかの方法で渡した後、外部パートナー側のクライアント開発環境で、そのファイルに対して「サービス参照の追加」を行って読み込み、プロキシを生成する方法があります。
> または、AWTS 社の社内環境で、予めプロキシを作成し、不要な部分を取り除くなどして、外部パートナーに渡す方法も考えられます。
> そのほかの方法としては、必要なサービス コントラクトやデータ コントラクトを外部パートナーに渡し、外部パートナー側で、チャネルファクトリを用いたコーディングをする方法です。

チャネル ファクトリによる WCF クライアントからのアクセス

ここでは、DummyExternalApp プロジェクトの中のクライアントとしての実装のうち、顧客のマイレージを照会する箇所について、プロキシを使わずに、チャネル ファクトリを使用するように変更しましょう。

チャネル ファクトリを使用する際には、サービス コントラクト（インターフェイスの定義）とデータ コントラクト（使用するデータ型の定義）が必要です。サービスの提供者などから、これらのコントラクトのソースコード、または、アセンブリを入手する必要があります。

この例のマイル照会で必要なコントラクトは、IMileageQuery インターフェイスと MemberInfo クラスの定義です。ここでは、簡単にするため、既存の AdwkMService プロジェクトから DummyExternalApp プロジェクトへ、ソースコードを取り込むことにしましょう。

> **註:** このあとの手順では、DummyExternalApp プロジェクトを書き変えます。予め、現状の DummyExternalApp プロジェクトを含む AdwkRouting ソリューション全体をコピーして、バックアップを作成しておくことをお勧めします。

1. DummyExternalApp プロジェクトを開きます。

まず、次に示す手順を行い、コントラクトが定義された IMileageQuery.cs と MemberInfo.cs を、AdwkMService プロジェクトから DummyExternalApp プロジェクトに取り込みます。

2. 必要なサービス コントラクト（IMileageQuery インターフェイス）を DummyExternalApp プロジェクトへ取り込むため、まず、ソリューション エクスプローラーのツリー上で、［DummyExternalApp］プロジェクトノードをクリックして選択します。
3. ［プロジェクト］メニューの［既存項目の追加］をクリックします。
4. ［既存項目の追加］ダイアログボックスが表示されたら、AdwkMService プロジェクトのフォルダーに移動し、「IMileageQuery.cs」を見つけて選択し、［追加］ボタンをクリックします。
5. 再び、［プロジェクト］メニューの［既存項目の追加］をクリックします。
6. ［既存項目の追加］ダイアログボックスが表示されたら、AdwkMService プロジェクトのフォルダーに移動し、「MemberInfo.cs」を見つけて選択し、［追加］ボタンをクリックします。

これで DummyExternalApp プロジェクトには、IMileageQuery インターフェイスと MemberInfo クラスの定義が追加されました。

ここで、既存のプロキシや関連する型を使用しないようにして、代わりに追加した 2 つの定義を利用するようにするため、念のため、以下のように using ディレクティブを変更しましょう。

7. Form1.cs のソースコードをコードエディターで開きます。
8. Form1.cs の冒頭付近で、プロキシの名前空間を使用する using ディレクティブをコメントアウトし、さらに、チャネルファクトリで必要な名前空間を 2 つ追加します（太字部分）。

例 47. 使用する名前空間の変更

Form1.cs（DummyExternalApp プロジェクト）

```
// プロキシの名前空間
//using DummyExternalApp.ServiceAdwkM; ←[1]
using DummyExternalApp.ServiceAdwkR;
```

```
using Adwk.Mileage; // 入手したコントラクトのため ←[2]
using System.ServiceModel.Channels; // IChannelのため
```

上記の[1]が、もともとのプロキシの名前空間です。これをコメントアウトした時点で、プロキシに関連するコードは、構文エラーになります。また、[2]に指定された名前空間が追加された IMileageQuery インターフェイスと MemberInfo クラスの名前空間です。残りの 1 行は、チャネル ファクトリとともに使用する IChannel インターフェイスのために必要です。

次に、マイレージの照会を行う際に、プロキシを使用していたコードを、チャネルファクトリを使用するように変更します。（元のコードとあまり変わらないので、コピーして流用するとよいでしょう。）

9. Form1.cs の btnMileageQuery_Click イベントハンドラー（[マイレージ照会]ボタンの Click イベントハンドラー）の中に記述された、既存のソースコードをコメントアウトするか、削除します（後から確認できるよう、コメントアウトしておくほうがよいでしょう）。
10. 次の例 48 のように、btnMileageQuery_Click イベントハンドラーの内部全体を書き換えます。

例 48. チャネル ファクトリの利用

Form1.cs（DummyExternalApp プロジェクト）

```
// 予約の照会
private void btnMileageQuery_Click(object sender, EventArgs e)
{
    // チャネルファクトリを使用するように変更
    ChannelFactory<IMileageQuery> factoryM = null;
    IMileageQuery clientM = null;
    try
    {
        factoryM = new ChannelFactory<IMileageQuery>(custBinding, epaM);  //←[1]

        factoryM.Credentials.UserName.UserName = txtSID.Text;    //←[2]
        factoryM.Credentials.UserName.Password = txtSpass.Text;
        factoryM.Credentials
            .ServiceCertificate.Authentication.CertificateValidationMode =
                X509CertificateValidationMode.PeerOrChainTrust;

        clientM = factoryM.CreateChannel();    //←[3]
        MemberInfo minfo;   //←[4]
        bool resultM = clientM.QueryInfo1ByMember(txtMCode.Text, out minfo); //←[5]

        ((IChannel)clientM).Close();

        MessageBox.Show("  ***  マイレージ照会結果  ***  " +
            Environment.NewLine + Environment.NewLine +
            "MemberCode : " + minfo.MemberCode + Environment.NewLine +
            "LastName : " + minfo.LastName + Environment.NewLine +
            "FirstName : " + minfo.FirstName + Environment.NewLine +
            "TotalMile : " + minfo.TotalMile.ToString());
    }
```

```
    catch (Exception ex)
    {
        MessageBox.Show(
            ex.Message + "¥n" +
            (ex.InnerException.Message ?? ""));
        if(clientM != null) ((IChannel)clientM).Abort();
    }
}
```

主なオブジェクトとしては、［1］で作成している「チャネル ファクトリ」と、そのチャネル ファクトリから作成する[3]の「チャネル」です。WCF サービスのメソッド呼び出す際に直接使用するオブジェクトは、後者のチャネルのほうですが、アクセスする際の様々な構成は、前者のチャネルファクトリに対して行います。

まず、[1]のチャネル ファクトリのインスタンスを作成する過程で、アクセスに使用するエンドポイントの 3 つの要素（ABC）を指定しています。

この 3 つのうち、コントラクト（C）については、ChannelFactory<T>クラスの型パラメーターT として IMileageQuery インターフェイスが指定されています。このインターフェイスは、手順 4 で取り込んだ「IMileageQuery.cs」に定義されたインターフェイスです。

また、チャネル ファクトリのコンストラクターの引数に渡された、変数 custBinding と変数 epaM は、それぞれバインディング（B）とアドレス（A）です。変数 custBinding は、このサンプルの中で、次のように、変数定義と Load イベントハンドラーにおいて、すでに wsHttpBinding の構成がされています。（もともと、プロキシで使用していたものを流用しています。）

例 49. コードによるバインディング wsHttpBinding の構成（抜粋）

Form1.cs（DummyExternalApp プロジェクト）

```
WSHttpBinding custBinding = new WSHttpBinding();

    （略）

private void Form1_Load(object sender, EventArgs e)
{
    custBinding.Security.Mode = SecurityMode.Message;
    custBinding.Security.Message.ClientCredentialType =
        MessageCredentialType.UserName;
```

また、変数 epaM のアドレスは、前章で取り上げた探索やアナウンスメントによって、最新のサービスのアドレスが設定されています。

[2]から始まる 5 行は、プロパティの名前が異なる点を除けば、もともとのプロキシを使用していたサンプルコードと同様で、クライアント認証用のクレデンシャルとサービス証明書の扱い方について指定しています。

そして、[3]でチャネル（変数 clientM）を作成しています。このオブジェクトの型は、IMileageQuery インターフェイスであり、ChannelFactory<T>に指定した型パラメーターIMileageQuery と同じになります。このチャネルは、プロキシに相当するもので、このチャネルのメソッドを呼び出すことで、WCF サービスのメソッドを呼び出すことができます。実際にサービスのメソッドを呼び出しているのは、[5]の部分です。その際、出力引数の受け皿として[4]の MemberInfo 型の変数を用意しています。この

変数の型は、このサービスでやり取りするデータコントラクトの 1 つであり、手順 6 で取り込んだ「MemberInfo.cs」に定義されています。

なお、参考までに、もともとのプロキシを使用したソースコードを以下の例 50 に掲載しておきます。ほとんど手順は類似していることが分かります。

ただし、この例 50 の[6]では、WSDL を元にプロキシを生成したので、出力引数の順番は、例 48 の[5]とは同じではありません。既に触れたように、WSDL では出力引数と入力引数との間の順番が定義されていないからです。これでも問題はありませんが、厳密に元のサービスの引数と順番を合わせたい場合、前述のように、コントラクトのソースコードを流用し、チャネル ファクトリを使用する方法があります。

例 50. プロキシを使用していた元のコード（参照）

Form1.cs（DummyExternalApp プロジェクト）

```
private void btnMileageQuery_Click(object sender, EventArgs e)
{
    var clientM = new MileageQueryClient(custBinding, epaM);

    try
    {
        clientM.ClientCredentials.UserName.UserName = txtSID.Text;
        clientM.ClientCredentials.UserName.Password = txtSpass.Text;
        clientM.ClientCredentials
            .ServiceCertificate.Authentication.CertificateValidationMode =
                X509CertificateValidationMode.PeerOrChainTrust;

        MemberInfo minfo;
        bool resultM = clientM.QueryInfo1ByMember(out minfo, txtMCode.Text); //←[6]

        clientM.Close();

        MessageBox.Show("  ***  マイレージ照会結果  ***  " +
            Environment.NewLine + Environment.NewLine +
            "MemberCode : " + minfo.MemberCode + Environment.NewLine +
            "LastName : " + minfo.LastName + Environment.NewLine +
            "FirstName : " + minfo.FirstName + Environment.NewLine +
            "TotalMile : " + minfo.TotalMile.ToString());

    }
    catch (Exception ex)
    {
        MessageBox.Show(
            ex.Message + "¥n" +
            (ex.InnerException.Message ?? ""));
        clientM.Abort();
    }
}
```

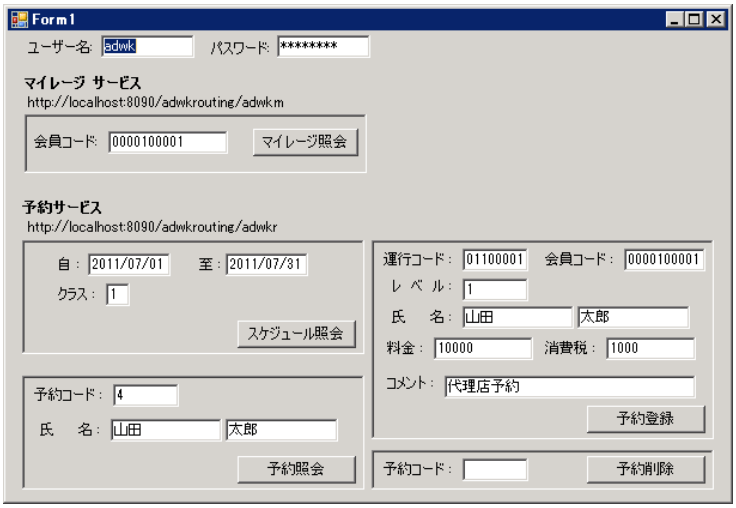
それでは、最後に正しく動作するか確認しておきましょう。

11. サービスに必要となる AdwkRouting、AdwkMService、および、AdwkRService の 3 つのプロジェクトを実行します。

12. DummyExternalApp を起動します。

13. 同様のフォーム Form1 が開くことを確認します（図 50）。

図 50. DummyExternalApp プロジェクトのフォーム Form1



14. フォーム左上部の［マイレージ照会］ボタンをクリックします。図 51 のように、今までと同様に、マイレージの照会結果が表示されることを確認します。（TotalMile の数値は、サンプルで様々な操作実験を行うことにより異なります。）

図 51. マイレージの照会結果

15. 確認が済んだら、メッセージボックスを閉じます。AdwkMService、AdwkRService、AdwkRouting、および、DummyExternalApp の各アプリケーションを終了します。

4.2 クライアントからの同期呼び出し

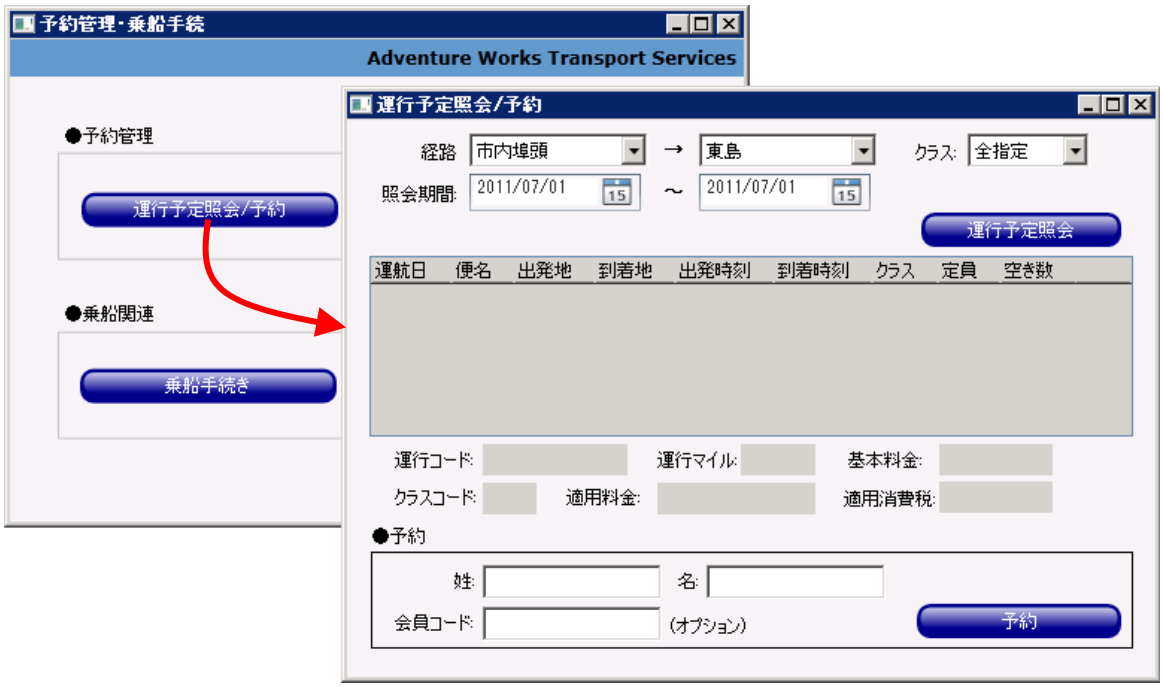
第 1 部では、クライアントからサービスを呼び出す際に、同期と非同期の 2 種類の形態がある点について説明しました。今まで取り上げたサンプルでは、同期呼び出しを使用していましたが、ここでは、非同期呼び出しの実装例を取り上げます。

まずは、既存の同期呼び出しのコードを確認し、そののち、実際にそのコード部分を非同期呼び出しに変更してみます。

現時点のサンプルでの同期呼び出し

予約管理サービスを利用する社内システムのクライアント（AdwkWpfApp プロジェクト）では、次図のように、最初に表示されるメインのウィンドウ上で、［運行予定照会/予約］ボタンをクリックすることによって、［運行予定照会/予約］ウィンドウが開きます。

図 52. 社内システムでの予約管理サービスのクライアント（AdwkWpfApp）



このときの処理の流れとしては、［運行予定照会/予約］ウィンドウが開く直前に、WCF サービスを呼び出して、［運行予定照会/予約］ウィンドウの各ドロップダウンリストのために、データ（経路やクラス）を取得しています。このデータ取得の際には、次の例 51 の[1]のように、BuildUI メソッドの中で同期呼び出しを用いて、サービスにアクセスしています。

例 51. WCF サービスの同期呼び出し

W01Reserve.xaml.cs（AdwkWpfApp プロジェクト）

```
// ユーザーインターフェイスに必要な予約関連情報の取得と設定
private void BuildUI()
{
    // 経路とクラス情報を取得してコンボボックスに設定する
    var proxy = new RBasicInfoClient();
```

```
    PlaceInfo[] places = null;
    LevelInfo[] levels = null;
    bool result;
    try
    {
        result = proxy.GetBasicRouteInfo3(out places, out levels); //←[1]
    }
    catch
    {
        result = false;
    }
    if (!result)
    {
        MessageBox.Show(
            "経路およびクラス情報の読み込みに失敗しました。");
        return;
    }

     （略）

}
```

よって、例 51 の[1]のところで、サービスに要求を送ると、プログラムの実行は一旦停止し、サービスが応答を返すまで待機します。そして、サービスが応答を返すと、[1]の次へと実行が続きます。

このため、サービスの応答が遅いと、このクライアント アプリケーションは、一時的にハングしたようになり、この時点ではウィンドウも表示されません。その結果、ユーザーにとっては、サービスに問い合わせ中という状況が理解しづらく、使い勝手はよくないといえます。

参考のため、実際の確認手順を以下に記載しておきます。

1. AdwkRService プロジェクト、および、AdwkMService プロジェクトを実行します。
2. AdwkWpfApp プロジェクトを実行します。
3. 図 52 の左側のメイン ウィンドウが表示されたら、［運行予定照会/予約］ボタンをクリックします。
4. ［運行予定照会/予約］ウィンドウが表示されるまで、しばらく時間（数秒）がかかることを確認します。（特に、AdwkRService を起動した直後であると、数秒かかることが分かります。）
5. AdwkRService、AdwkMService、AdwkWpfApp の各プロジェクトのアプリケーションを終了します。

註: 手順 4 でサービスの応答に時間がかかることが、はっきり確認できない場合は、呼び出し先のサービスである AdwkRService プロジェクト内で、意図的に時間がかかるように、スレッドを数秒停止させるのコード（例 52 の[1]）を追加してみてください。すでにサンプルには、コメントアウトした状態で、この記述があります。

例 52. WCF サービスの呼び出し時に、意図的に応答を遅くする

RBasicInfoService.cs（AdwkRService プロジェクト）

```
bool IRBasicInfo.GetBasicRouteInfo(out PlaceInfo[] places, out LevelInfo[] levels)
{
    System.Threading.Thread.Sleep(5000);  //←[1]

    return BLComp1.GetBasicRouteInfo(out places, out levels);
}
```

このあとは、このアプリケーションの使い勝手をよくするため、非同期呼び出しに変更します。非同期呼び出しにすることで、データの取得はバックグランドで行うようになり、ただちにウィンドウは表示されます。その際にウィンドウには、現在読み込み中である旨のメッセージも表示でき、よりユーザーフレンドリーな UI になります。（さらに必要があれば、データの取得中に、ユーザーが別の作業を行えるように UI を実装することもできます。）

プロキシにおける非同期呼び出し

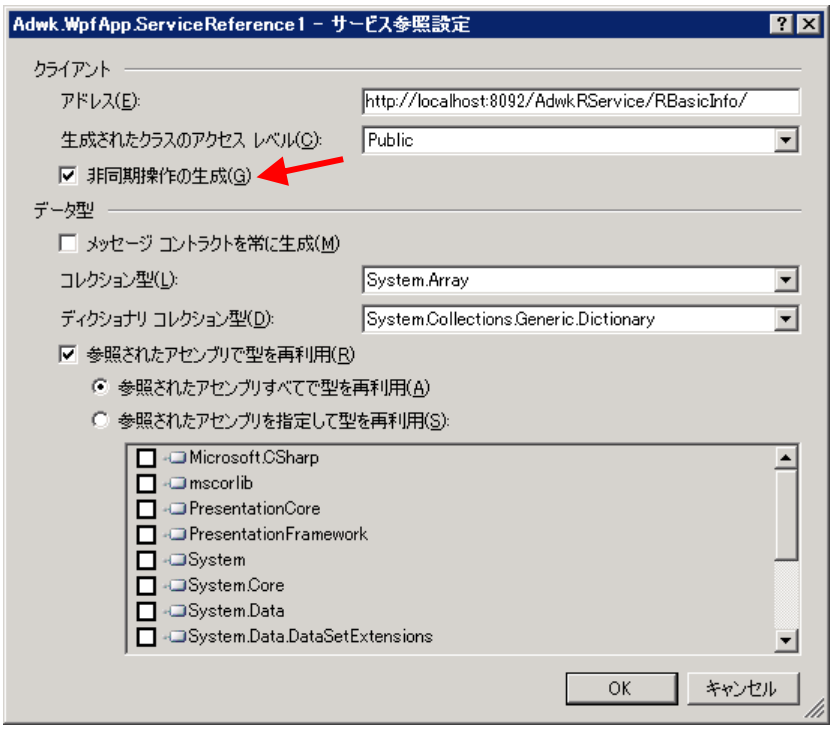
プロキシで非同期呼び出しを行うには、非同期のためのメソッドを追加する必要があります。既定オプションのまま「サービス参照の追加」を行って作成したプロキシには、非同期呼び出しのためのメソッドがありません。そこで、以下の手順で非同期呼び出し用のメソッド（非同期操作用のメソッド）を改めて追加してみましょう。

1. AdwkRService プロジェクトを実行します。（AdwkWpfApp プロジェクトでのサービス参照の追加の際に、メタデータを再取得するために必要です。）
2. AdwkWpfApp プロジェクトを開きます。
3. ソリューション エクスプローラーのツリー上で、［Service References］ノードの配下にある［ServiceReference1］ノードを右クリックして、ショートカットメニューから［サービス参照の構成］をクリックします。

上記の手順 3 は、一旦作成した「サービス参照の追加」の成果物（ここでは ServiceReferece1）を、後から設定変更する場合の手順です。

4. すると、次図のように［Adwk.WpfApp.ServiceReferece1 - サービス参照設定］ダイアログボックスが表示されます。ここで、非同期呼び出しのメソッドをプロキシの中へ生成させるため、ダイアログボックスの上部にある［非同期操作の生成］チェックボックスをチェックします。

図 53. サービス参照設定



5. ［OK］をクリックして、このダイアログボックスを閉じ、非同期操作のメソッドをプロキシ内に生成させます。

> **註:** 既存の「サービス参照の追加」の成果物に対して、非同期操作のメソッドを追加するのではなく、新規に「サービス参照の追加」を行う場合に、非同期操作のメソッドを生成させるのであれば、［サービス参照の追加］ダイアログボックスの左下部にある［詳細設定］ボタンをクリックしてください。すると、図 53 と同じダイアログボックスが表示されます。

これで、プロキシには非同期呼び出しに利用できる 2 つのメソッドとして、「Begin～」と「End～」という名前で始まるものが追加されました。次に、このメソッドを実際に使ってみます。

6. AdwkWpfApp プロジェクトの「W01Reserve.xaml.cs」をコード エディターで開き、次の例 53 のように、BuildUI メソッドの内部全体を変更し、また、[1]以降に挙げた 3 つのメンバーを追加し、さらに、OnEndCall メソッドと PopulateUIAfterCall メソッドを追加します。

例 53. WCF サービスの非同期呼び出しのための実装コード

W01Reserve.xaml.cs（AdwkWpfApp プロジェクト）

```
// 経路とクラス情報を取得してコンボボックスに設定するプロキシ
RBasicInfoClient proxy = new RBasicInfoClient();  //←[1]
// 呼び出し結果のデータを参照する変数
PlaceInfo[] places = null;  //←[2]
LevelInfo[] levels = null;

// ユーザーインターフェイスに必要な予約関連情報の取得と設定
private void BuildUI()  //←[3]
{
    IAsyncResult result = null;
```

```
    try
    {
        result = proxy.BeginGetBasicRouteInfo3(           ←[4]
                  new AsyncCallback(OnEndCall), null);
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message);
        return;
    }

    // 関連するUIの初期化と無効化
    QueryMessageLabel.Content = "路線とクラスの基本情報を読み込み中です..."; //←[5]
    arrvCombo.IsEnabled = false;
    deptCombo.IsEnabled = false;
    levelCombo.IsEditable = false;
    fromDate.IsEnabled = false;
    toDate.IsEnabled = false;
    QueryButton.IsEnabled = false;
}

// 呼び出し完了時のイベント
private void OnEndCall(IAsyncResult ar)  //←[6]
{
    try
    {
        bool result;
        result = proxy.EndGetBasicRouteInfo3(out places, out levels, ar); //←[7]
        this.Dispatcher.Invoke(new Action(PopulateUIAfterCall)); //←[8]
        proxy.Close();
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message);
        proxy.Abort();
    }
}

// 呼び出し完了後にUIを設定する
private void PopulateUIAfterCall()  //←[9]
{
    // 出発地のリストを作成
    deptCombo.DisplayMemberPath = "PlaceName";
    deptCombo.SelectedValuePath = "PlaceCode";
    deptCombo.ItemsSource = places;
    deptCombo.SelectedIndex = 0;
    // 到着地のリストを作成
    arrvCombo.DisplayMemberPath = "PlaceName";
    arrvCombo.SelectedValuePath = "PlaceCode";
    arrvCombo.ItemsSource = places;
    arrvCombo.SelectedIndex = places.Length > 1 ? 1 : 0;
    // クラス情報にワイルドカードを追加
    LevelInfo[] levelsExtend = new LevelInfo[levels.Length + 1];
    Array.Copy(levels, 0, levelsExtend, 1, levels.Length);
    levelsExtend[0] = new LevelInfo() { LevelCode = "", LevelName = "全指定" };
    levelCombo.DisplayMemberPath = "LevelName";
    levelCombo.SelectedValuePath = "LevelCode";
```

```
    levelCombo.ItemsSource = levelsExtend;
    levelCombo.SelectedIndex = 0;

    // 日付をコンボボックスに設定する
    // (サンプルデータの期間にあわせて、初期値値を 2011/07/01～2011/07/01とします)
    fromDate.SelectedDate = new DateTime(2011, 7, 1);
    toDate.SelectedDate = new DateTime(2011, 7, 1);

    // 関連するUIの設定と有効化
    QueryMessageLabel.Content = "";  //←[10]
    arrvCombo.IsEnabled = false;
    deptCombo.IsEnabled = false;
    levelCombo.IsEditable = false;
    fromDate.IsEnabled = true;
    toDate.IsEnabled = true;
    QueryButton.IsEnabled = true;
}
```

[1]の RBasicInfoClient オブジェクトが、このサンプルで使用するプロキシです。また、プロキシを呼び出して取得するデータは、[2]の二つ配列 places と levels で参照するようにします。今回は、プロキシやこれらの配列を、複数のメソッドをまたいで使用するので、このウィンドウ クラスのメンバー変数として、これらを宣言しています。

[3]の BuildUI メソッドの内部では、[4]の BeginGetBasicRouteInfo3 メソッドを呼び出すことで、非同期の呼び出しでの、サービスへの問い合わせを開始します。サービスへの問い合わせは、バックグランドで行われ、BeginGetBasicRouteInfo メソッド自体の呼び出しは、直ぐに終了して、それ以降の処理へ進みます。この結果、BuildUI メソッド自体は、サービスの応答を待たずして終了するので、すぐにウィンドウが表示されます。

なお、[4]の BeginGetBasicRouteInfo メソッドの 1 番目の引数には、サービスでの処理が終了して応答が返った際に、完了通知を受けるイベントハンドラー（のデリゲート）を指定します。ここでは、OnEndCall メソッドを、完了通知を受け取るメソッドとして渡しています。

この結果、バックグランドでのサービス問い合わせが完了すると、[6]の OnEndCall メソッドが呼び出されます。このメソッドの中で、[7]のように、EndGetBasicRouteInfo3 メソッドを呼び出せば、問い合わせ結果の出力引数や戻り値を受け取ることができます。

このとき、[7]の EndGetBasicRouteInfo メソッドの最後の引数には、[6]の OnEndCall の引数の ar を渡しています。これによって、この OnEndCall が呼び出された時点の、適切な問い合わせ結果を受け取ることができます。この[7]の呼び出しによって、出力引数 places と levels には、問い合わせ結果の経路情報とクラス情報が格納されます。

注意すべき点として、[6]の OnEndCall はウィンドウのユーザーインターフェイスとは別の、バックグランドスレッドで実行されている点です。WPF も Windows フォームも、ユーザーインターフェイス（UI）は、UI 専用の単一スレッド上で実行されるため、バックグランドのスレッドから直接 UI を操作するのは望ましくありません（UI はスレッドセーフではありません）。そのため、[7]を実行しているスレッドから、直接的にウィンドウのドロップダウンリストにデータを設定できません。

そのため、[8]のように UI 専用のスレッドに切り替えて処理させる必要があります。[8]では、このウィンドウの Disptacher オブジェクトプロパティの Invoke メソッドを呼び出し、引数に

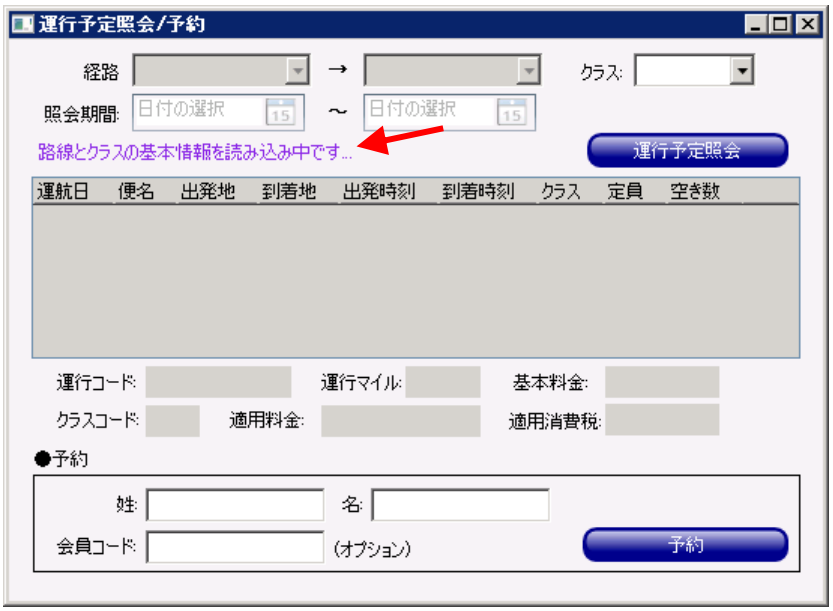
PopulateUIAfterCall メソッドのデリゲートを渡しています。これによって、明示的に UI 専用のスレッド上で、PopulateUIAfterCall メソッドが呼び出されます。

そして、UI 専用のスレッドで実行されている[9]の PopulateUIAfterCall メソッドでは、配列 places や levels の値をもとに、ドロップダウンリストに必要なデータを設定しています。さらに[10]以降では、読み込み中の表示をクリアし、必要な UI を有効化すべく、IsEnabled プロパティに true を設定しています。

これで完成しました。実際の動作を確認してみましょう。

7. 再び、AdwkRService プロジェクト、および、AdwkMService プロジェクトを実行します。
8. AdwkWpfApp プロジェクトを実行します。
9. 図 52 の左側のメイン ウィンドウが表示されたら、［運行予定照会/予約］ボタンをクリックします。
10. ［運行予定照会/予約］ウィンドウが、修正前よりも早く表示されるようになり、上部のラベルに読み込み中の旨のメッセージが表示されることを確認します。また、経路のドロップダウンリストや照会期間の日付指定欄の DatePicker コントロールなども、無効化されていることを確認します。

図 54. 非同期呼び出しによるバックグランド処理中の表示



11. しばらくすると、データが読み込まれて、ドロップダウンリストに表示され、関連する UI も有効化されることを確認します。

図 55. 非同期呼び出しの完了

12. 確認が済んだら、このウィンドウを閉じ、メインのウィンドウも閉じて、AdwkWpfApp のクライアント アプリケーションを終了します。
13. 再び、AdwkRService プロジェクト、AdwkMService プロジェクト、および、AdwkWpfApp プロジェクトの実行を終了します。

4.3 型指定されたデータセットの利用

第 1 章でも触れたように、WCF サービスと WCF クライアントの間では、データ コントラクトに基づいて、様々な種類のデータがやり取りできます。その際に、ネットワーク上にデータを送信すると、データはシリアル化され、受信すると、逆シリアル化されます。このような WCF におけるシリアル化に対応したデータには、DataContract 属性などを明記したカスタムデータ型だけでなく、.NET Framework の基本的なデータ型（プリミティブなデータ型）のほか、様々なデータ型があります。

型指定されたデータセットも、WCF によるシリアル化がサポートされたデータ型の 1 つです。この節では、型指定されたデータセットを WCF サービスとクライアントとの間でやり取りする際の特徴や注意点について取り上げます。

註: WCF によるシリアル化は大きく分類すると、DataContractSerializer によるシリアル化と、XmlSerializer によるシリアル化があります。それぞれの詳細やサポートされるデータ型については、以下のアドレスを参照してください。

http://msdn.microsoft.com/ja-jp/library/ms731072.aspx
データ コントラクト シリアライザー (DataContractSerializer の場合)
http://msdn.microsoft.com/ja-jp/library/ms733901.aspx
XmlSerializer クラスの使用 (XmlSerializer の場合)

データセットでは、DataContractSerializer によるシリアル化がサポートされた IXmlSerializable インターフェイスが実装されているので、WCF サービスとクライアントとの間でやり取りできます。

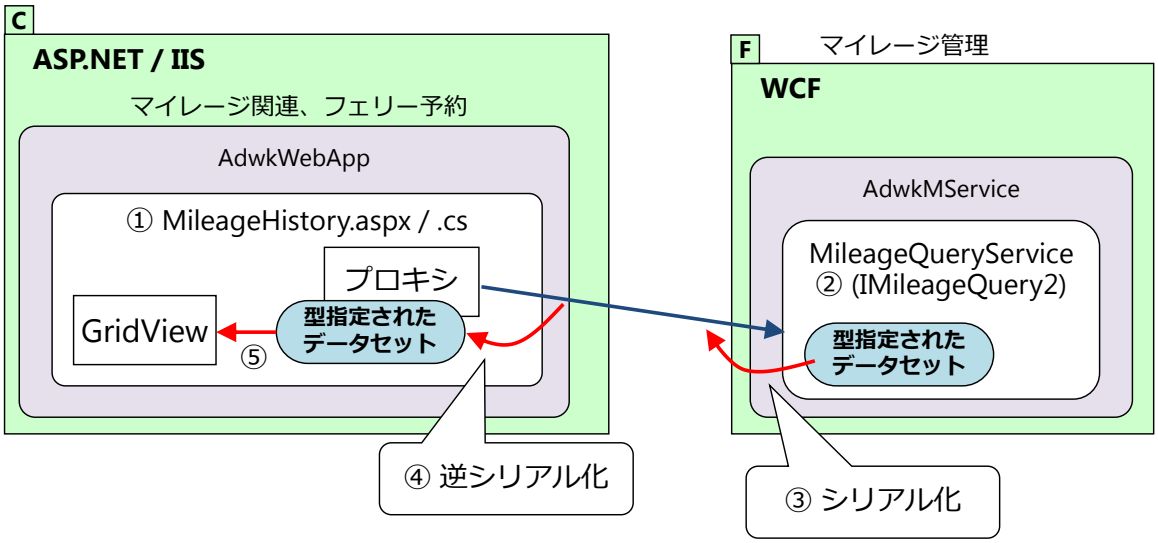
データセットは、.NET 固有のデータ型であるため、WCF サービスのデータとしてやり取りする場合は、他のデータ型よりも相互運用性は劣りますが、データセット自身は Visual Studio の様々なツールで利用されるデータ型なので、.NET の開発環境では扱いやすく、開発生産性の面では優れているといえます。

このあとは、データセットを WCF で使用する場合の留意点について、いくつか取り上げます。

・サンプルに見るデータセットの受け渡し

顧客向けの Web アプリケーションである AdwkWebApp（図 6 の C）は、マイレージ管理サービスである AdwkMService（図 6 の F）にアクセスします。その際、サービスとの一部のやり取りには、型指定されたデータセット（に含まれる型指定されたデータテーブル）を使用しています。この構成を次に示します。

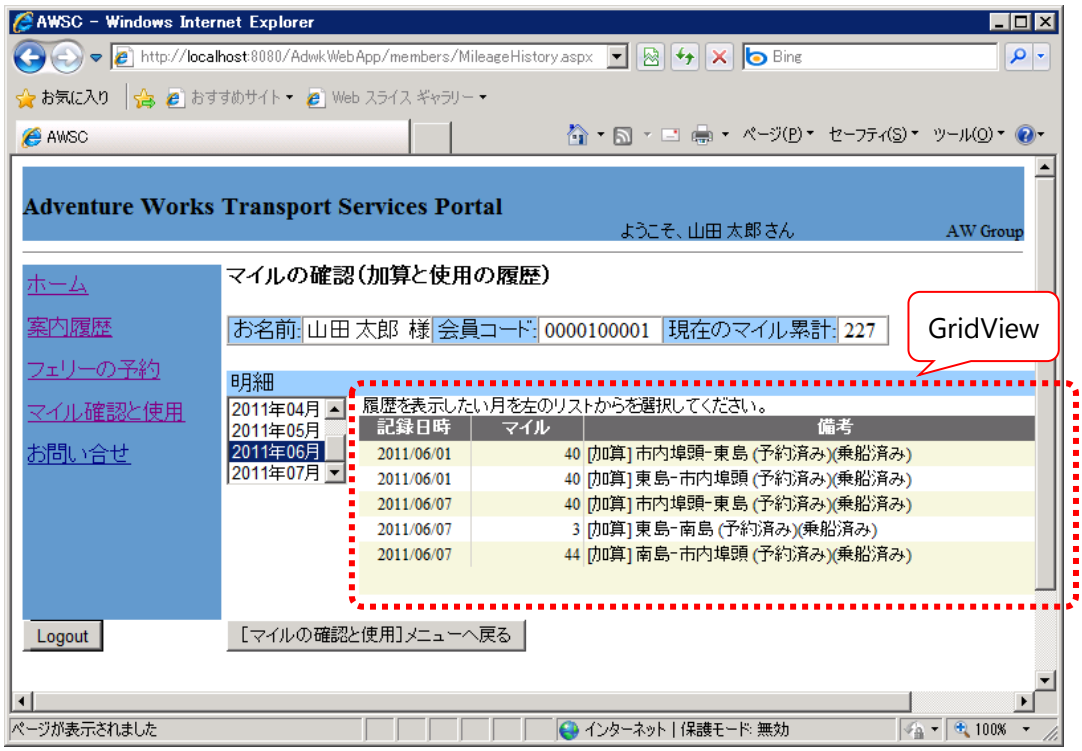
図 56. 型指定されたデータセットの利用



①の MileageHistory.aspx（図中の C の内部）は、マイレージの履歴を照会する ASP.NET Web ページであり、②の WCF サービス（図中の F の内部）に問い合わせています。このサービスでは、照会結果である履歴を型指定されたデータセット（に含まれる型指定されたデータテーブル）に格納し、これはシリアル化されて（③）、クライアントへ返されます。

受け取ったクライアントでは、逆シリアル化されて（④）、元の状態のデータセット（のデータテーブル）に戻ります。そして、型指定されたデータセットのマイレージ履歴は、Web ページ上の GridView に反映され（⑤）、次の図 57 のように表示されます。

図 57. 型指定されたデータセットのテーブルデータを GridView に表示



このようなデータセットのテーブルをやり取りするために、WCF サービスで使用するコントラクト（IMileageQuery2 インターフェイス）では、次のように、QueryHistoryByMemberAndPeriod メソッドの戻り値として、型指定されたデータテーブルである「会員マイル履歴 DataTable」型（太字部分）を使用しています。もともと、データセットは WCF におけるシリアル化がサポートされているので、このように戻り値に記述すれば十分であり、シリアル化のための特別な実装は必要ありません。

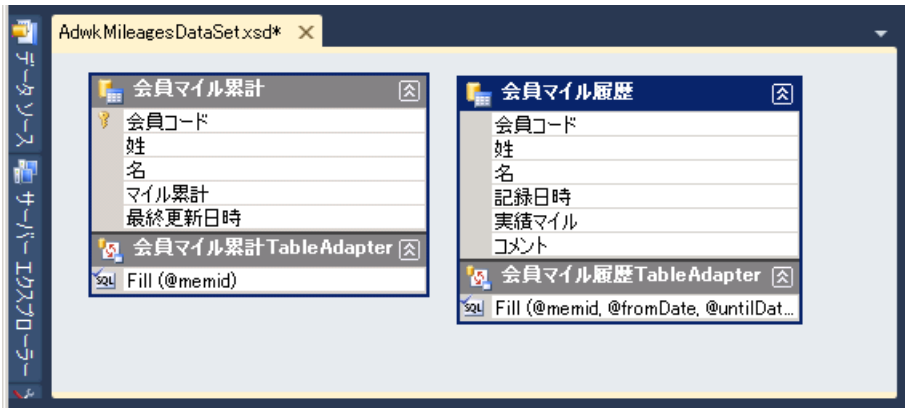
例 54. 型指定されたデータセットのテーブルを返す IMileageQuery2 インターフェイス（抜粋）

IMileageQuery2.cs（AdwkMService プロジェクト）

```
// ユーザーインターフェイスに必要な予約関連情報の取得と設定
 [ServiceContract(Namespace="http://Adwk.Mileage")]
public interface IMileageQuery2
    : IMileageQuery
{
    [OperationContract]
    AdwkMileagesDataSet.会員マイル履歴DataTable QueryHistoryByMemberAndPeriod(
        string memberCode,          // [in] 会員コード
        DateTime fromDate,          // [in] 期間開始日
        DateTime untilDate,         // [in] 期間終了日(の翌日0時)
        out AdwkMStatus stat,       // [out] ステータス
        out string systemMessage); // [out] システムメッセージ
}
```

なお、このサンプルの型指定されたデータセットは、既にデータセット デザイナーを用いて作成してあります。データセット デザイナーの直接の成果物は、AdwkMDac プロジェクトに含まれる AdwkMileagesDataSet.xsd であり、このファイルをソリューション エクスプローラーのツリー上でタプルクリックすると、次図のように、データセット デザイナーが開きます。このデザイナー内の「会員マイル履歴」というタイトルのブロックが、例 54 の戻り値で使用している「会員マイル履歴 DataTable」の定義であり、この定義に基づいてソースコードが自動的に出力されます。

図 58. データセット デザイナー



> **註:** データセットデザイナーの使用方法については、以下のアドレスを参照してください。
> http://msdn.microsoft.com/ja-jp/library/314t4see.aspx データセット デザイナー

このあとは、このようなデータセットを扱う際に注意すべき点を説明します。

型指定されたデータセットを必要とするサービス参照の追加

ここで取り上げている WCF サービスでは、例 54 のコントラクト（IMileageQuery2 インターフェイス）に示すように、戻り値として型指定されたデータセット（のデータテーブル）を使用しています。実は、このようなサービスに対して「サービス参照の追加」を行っても、クライアント側には、正しく型指定されたデータセットを自動生成させることができません。つまり、生成されるプロキシは、型指定されたデータセットのやり取りができません。というのは、サービスが返すメタデータは、.NET 固有の型指定されたデータセットの情報を持たないからです。

よって、型指定されたデータセットをやり取りするようなプロキシを生成させるためには、「サービス参照の追加」を行う際に、型指定されたデータセットを認識できるようにするため、予め、サービスを提供する組織から、型指定されたデータセットを含むアセンブリファイルを受け取り、クライアントの開発環境で参照できるように設定しておく必要があります。

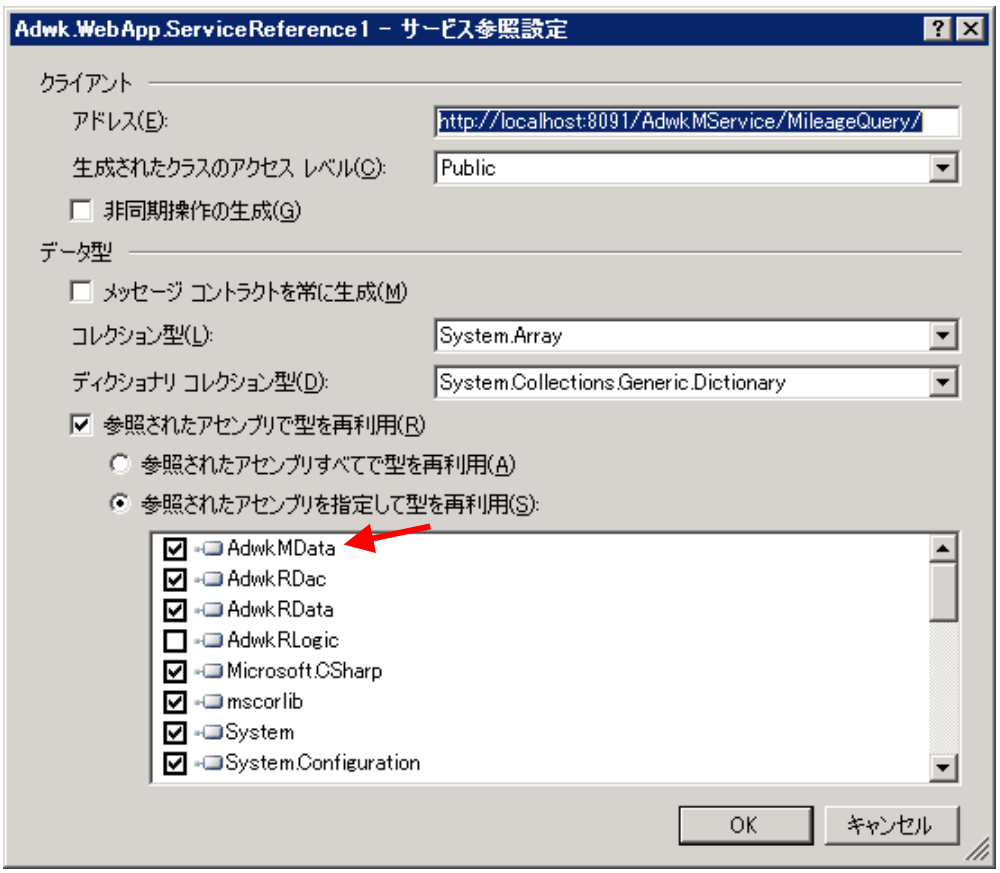
今回のサンプルでは、まず、WCF サービス側の実装である AdwkMService ソリューションの内部で、型指定されたデータセットを AdwkMData プロジェクトに作成しています。そして、WCF クライアントに相当する AdwkWebApp の開発環境へ、AdwkMData プロジェクトを配布すべく、プロジェクト全体をコピーして、AdwkWebApp ソリューションに含めています。（本来必要なのは、AdwkMData プロジェクトをビルドしたアセンブリファイルだけですが、ここではサンプルを調べやすくするため、AdwkMData プロジェクト全体をコピーしています。）

そして、WCF サービスのクライアントである AdwkWebApp プロジェクトでは、型指定されたデータセットを持つ AdwkMData プロジェクトに対して参照設定がされています。さらに、サービス参照の追加の際に、そのプロジェクトを参照するように設定しています。既存のこの設定は、次の手順で確認や変更ができます。

1. AdwkWebApp プロジェクトを開きます。

2. ソリューション　エクスプローラーのツリー上で、AdwkWebApp プロジェクトの Service References フォルダーの配下にある[ServiceReference1]ノードを右クリックして、ショートカットメニューから［サービス参照の構成］を選択します。
3. すると、［Adwk.WebApp.ServiceReference1 - サービス参照設定］ダイアログボックスが表示されるので、次図のように AdwkMData プロジェクトを参照するように、「AdwkMData」のチェックボックスがチェックされていることを確認します。

図 59. サービス参照の設定における AdwkMData プロジェクトの参照



このダイアログボックス内の下部のチェックボックス一覧には、参照設定されたプロジェクトやアセンブリの一覧が表示されます。この一覧で、型指定されたデータセットを含むアセンブリを指定します。ここで指定されたデータ型は、プロキシ生成の過程で必要に応じて流用され、生成されるプロキシのメソッドが持つ引数や戻り値の型には、適切な型指定されたデータセットが使用されるように、コードが生成されます。

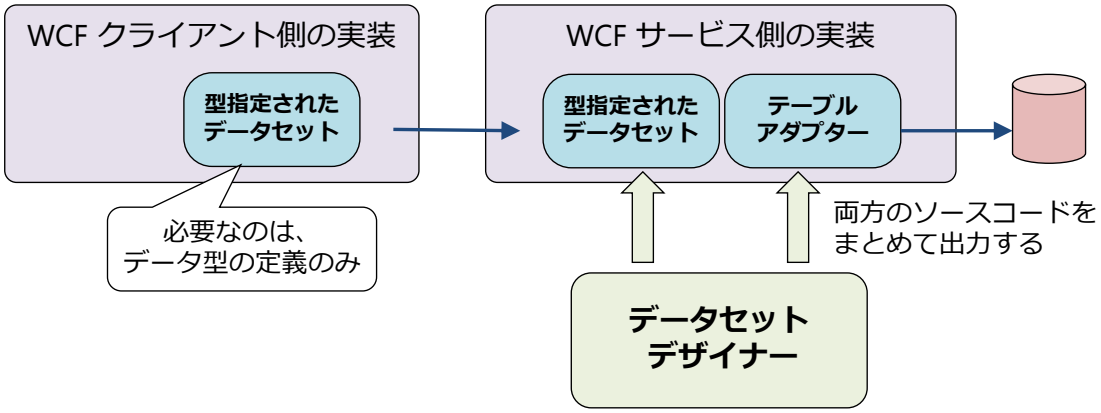
> 註: 上の説明では、既存の「サービス参照の追加」に対しての設定方法を確認しました。新規にサービス参照の追加を行う場合は、サービス参照の追加の際に表示される［サービス参照の追加］ダイアログボックスの左下部にある［詳細設定］ボタンをクリックすれば、図 59 と同様の設定画面を表示できます。

データセット デザイナーでのデータアクセスコードと型定義の分離

前述のサービス参照の追加で必要となるのは、型指定されたデータセットの定義だけですが、Visual Studio のデータセット デザイナーでコードを生成すると、型指定されたデータセットの定義だけでなく、テーブル アダプターなどのデータベースにアクセスのための実装コードもまとめて自動生成します。

このようなテーブル アダプターなどは、データベースにアクセスする実装コードであり、クライアントには、不要な実装コードです（図 60）。クライアントは、サービスとの間で引数や戻り値のやり取りをする際に必要なのは、そのデータの型定義だけです（ここでは、型指定されたデータセットの定義だけ必要です）。

図 60. データセット デザイナーが出力するコードと利用すべき箇所



このようなときデータセットデザイナーでは、テーブル アダプターなどのデータ アクセスの実装コードと、データセットなどのデータをやり取りするための型の定義を分離して、別々のソースコードとして出力できます。

今回の例で確認してみましょう。

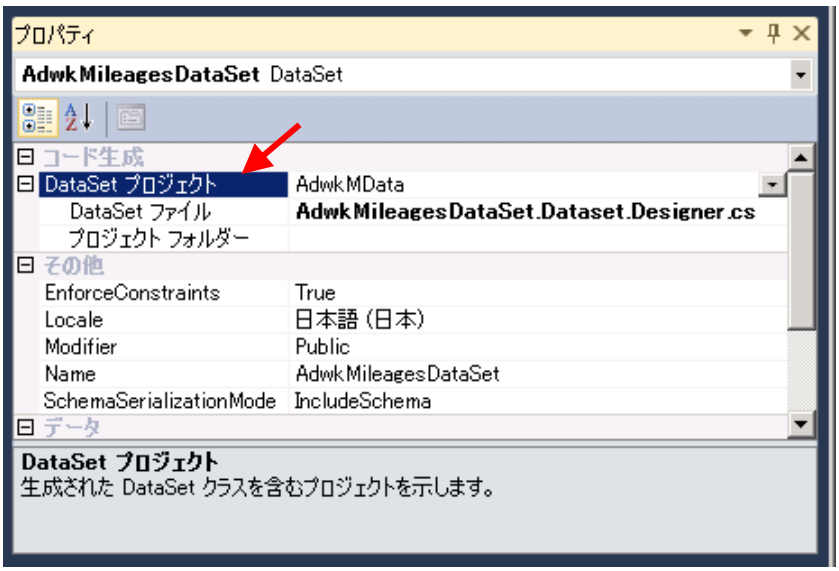
データセット デザイナーを用いた操作自体は、AdwkMDac プロジェクトで行っています。既定構成のままだと、AdwkMDac プロジェクトにソースコードがまとめて生成されます。しかし、このサンプルではテーブル アダプターは AdwkMDac プロジェクトに生成され、型指定されたデータセットは AdwkMData プロジェクトに分離して生成されるように、既に構成されています。

このための構成は、図 58 のデータセット デザイナーを開いた状態で、プロパティ ウィンドウから指定できます。以下の手順で確認することができます。

1. AdwkMService ソリューションに含まれる AdwkMDac プロジェクト内の、AdwkMileagesDataSet.xsd をデータセット デザイナーで開きます。（ソリューション エクスプローラーのツリー上で、AdwkMileagesDataSet.xsd をダブルクリックすると、データセット デザイナーで開きます。）
2. データセット デザイナー内の余白領域をクリックするなどして、データセット デザイナー全体を選択状態にします。

3. プロパティ ウィンドウが開いてなければ、プロパティ ウィンドウを開きます。（［F4］キーを押せば、プロパティ ウィンドウが表示されます。）
4. データセット デザイナー全体が選択されている状態を維持しながら、プロパティ ウィンドウで「DataSet プロジェクト」プロパティを見つけ、このプロパティの先頭の［+］ボタンを展開し、次図のように詳細の設定が表示されるようにします。

図 61. データセットの定義を別のプロジェクトへ分離して出力する設定



このデータセット デザイナーの直接の成果物（AdwkMileagesDataSet.xsd）は、AdwkMDac プロジェクトに属しますが、前図 61 の「DataSet プロジェクト」プロパティには、「AdwkMData」と指定してあるので、この指定によって、データセットの定義は AdwkMData プロジェクトへ分離して出力されます。また、「DataSet プロジェクト」プロパティの配下の「DataSet ファイル」プロパティでは、出力されるソースファイルの名前を指定できます。

WCF サービスなどを用いた分散環境では、クライアント側で、データの受け皿などのデータ型の定義（エンティティの定義）が必要になることがあります。そのような場合、前図 61 の設定を用いることで、テータセットの型定義のみを分離できるので便利です。

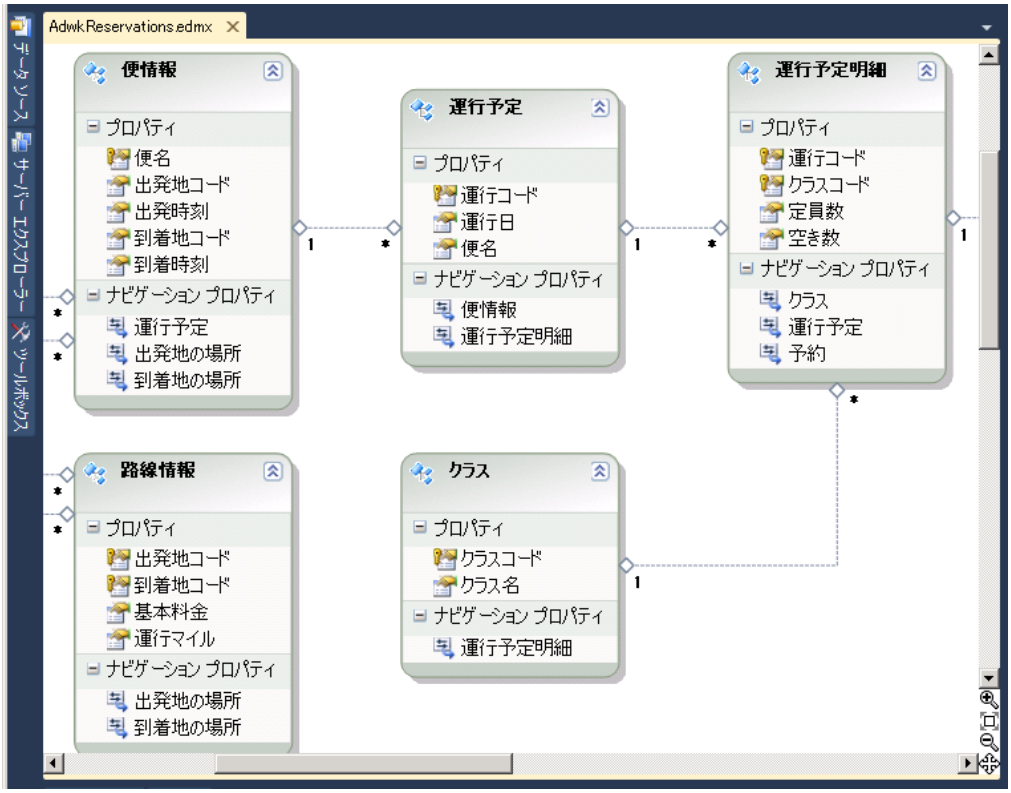
4.4 Entity Data Model の利用

WCF サービスでは、前節で取り上げたデータセットを利用できるだけでなく、ADO.NET Entity Framework のデータ モデルである Entity Data Model（EDM）も扱うことができます。この節では、WCF サービスにおいて、このような EDM の利用方法や注意点について説明します。

EDM に基づく具体的なエンティティの定義（データ型などの定義）は、Visual Studio の EDM デザイナーのビジュアル環境を使用して、対話操作で作成することができます。この AWTS サンプル プログラムでも、予約管理サービスの一部のデータについては、EDM デザイナーを用いてデータ モデルを定義しています。具体的には、AdwkRService ソリューションの中の、AdwkRDac プロジェクトに含まれる AdwkReservations.edmx がモデルの定義ファイルです。ソリューション エクスプローラー上

で、このファイルをダブルクリックすると、次図のように EDM デザイナーが開いて、ビジュアルな環境で編集することができます。

図 62. EDM デザイナー



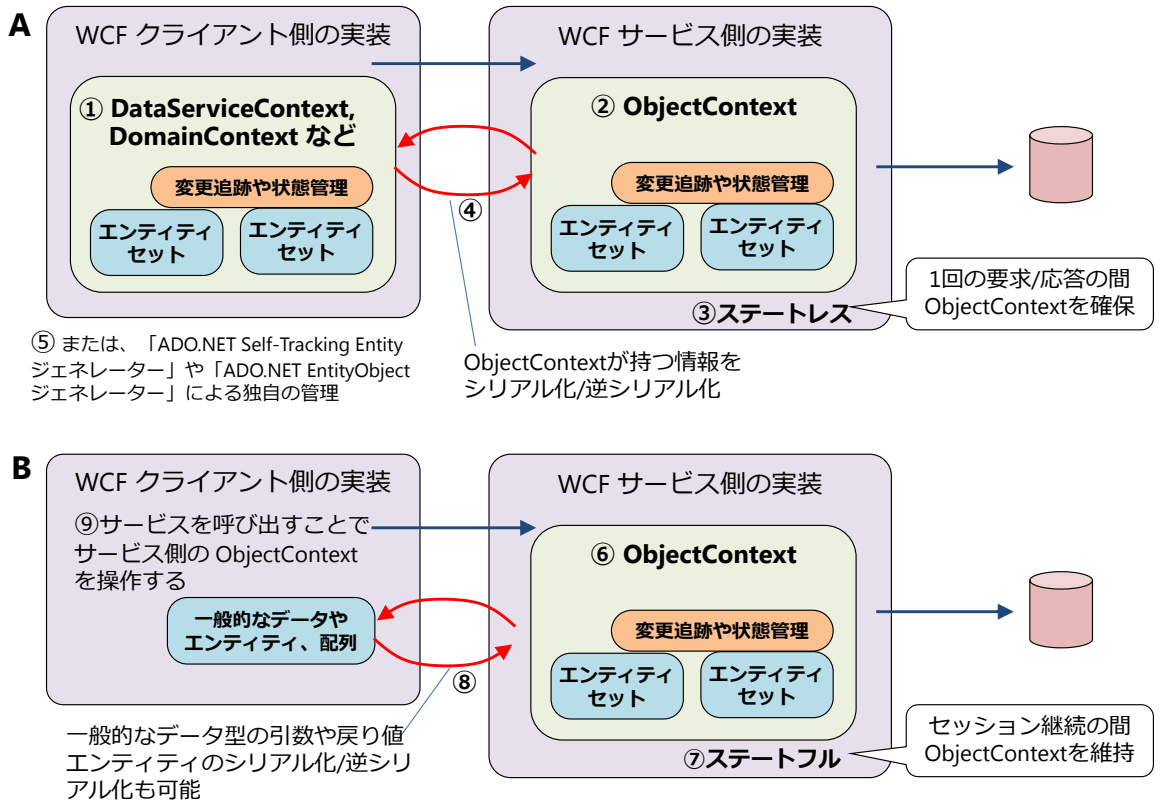
このデザイナーでの対話操作を介して、最終的には、EDM 対応のエンティティ定義（データ型定義）のソースコードが生成されるほか、エンティティの管理や操作を行う「ObjectContext」とよばれるオブジェクトのソースコードが生成されます。このあとは、このようなエンティティや ObjectContext を WCF サービスで使用する方法や注意点について確認します。

> 註: ADO.NET Entity Framework については、次のアドレスから自習書を入手することができます。
> http://msdn.microsoft.com/ja-jp/data/gg615417 データ アクセス自習書
> 「データ アクセス自習書 (Visual Studio 2010/.NET4 版)」の中の
> 「第 3 部 SQL Server データ アクセス手法 - (3) ADO.NET Entity Framework 編」

WCF サービスにおける EDM の利用形態

一般に EDM のデータモデルを使用する典型的な方法としては、EDM デザイナーによって生成されたエンティティの定義や、そのエンティティを管理する ObjectContext（正確には ObjectContext 派生クラス）を用います。特に、WCF サービスを伴う分散環境では、次のような利用形態が考えられます。

図 63. 分散環境における EDM の利用



この図 63 では、上半分の A と下半分の B の 2 つのパターンが挙げられています。それぞれのパターンの中央の WCF サービスで、ObjectContext が使用されています（それぞれ②、⑥）。

ObjectContext は、データベース上のデータを読み込んで、データベースからは切断した非接続の状態で、データをキャッシュできます。ObjectContext の管理下にあるエンティティのオブジェクト インスタンス 1 つ分が、テーブルの 1 行に相当します。テーブル 1 つ分は、エンティティ セットと呼ばれるオブジェクトとして管理されています。

また、②や⑥の欄にも示したように、ObjectContext はエンティティの変更を追跡できるようになっています。これらの変更追跡は、データベースとは非接続の状態で行われ、最終的には、データベースに接続して、まとめて変更内容をデータベースに反映できます。

A と B の大きな違いは、WCF サービスでステートを維持しているか否かの違いです。

A の WCF サービスでは、③に示したようにステートレスであり、クライアントからの 1 回の要求/応答の間だけ、ObjectContext インスタンスが確保されます。そのため、ObjectContext は継続的にデータや変更追跡を保持しません。その代り、クライアントとサービスとの間では、④に示したように、ObjectContext の内容を一旦、シリアル化して、クライアント側に移行し、クライアント側でクライアントによる変更を直接追跡します。

ただし、EDM デザイナーで作成した既定の ObjectContext やエンティティセットには、WCF におけるシリアル化がサポートされていません。つまり、クライアント側へ既定の ObjectContext をそのままシリアル化して送ることができません。代わりに、第 1 部で取り上げた WCF Data Services や WCF RIA Services を使用して、クライアント側では、それぞれ、A の①に示した DataServiceContext や

DomainContext を使用します。①のオブジェクトを使用することで、WCF サービス側の ObjectContext と同様のデータのキャッシュや変更追跡をクライアントで行うことができます。そして、クライアント側のデータの変更は、シリアル化され、サービス側の ObjectContext へ反映され、また逆に、サービス側の ObjectContext の内容もクライアント側に反映できます。

これ以外の方法としては、A の⑤に示したように、Visual Studio の項目テンプレートである「ADO.NET Self-Tracking Entity ジェネレーター」や「ADO.NET EntityObject ジェネレーター」を使用すると、独自の変更追跡やシリアル化を行うよう、エンティティや ObjectContext をカスタマイズすることも可能です。

> **註:** この 2 つの項目テンプレートの詳細は、以下のアドレスを参照してください。
> http://msdn.microsoft.com/ja-jp/library/ff477605.aspx
> ADO.NET EntityObject ジェネレーターテンプレート
> http://msdn.microsoft.com/ja-jp/library/ff477604.aspx
> ADO.NET Self-Tracking Entity Generator テンプレート

また、B のパターンでは、WCF サービス側で、ユーザー セッションが継続する間、⑥の ObjectContext をステートとして維持する方法です。この方法では、変更追跡やデータキャッシュは⑥で継続的に保持できるので、クライアント側へ ObjectContext の内容全体をシリアル化して送る必要はありません。クライアントは、必要に応じて、WCF サービスのメソッドを呼び出して、サービス側の ObjectContext に対して、データの問い合わせや変更などを働きかけます。このとき、クライアントとサービスとの間でやり取りされるデータは、一般的なデータ型（int や string など）でも構いません。

ただし、EDM デザイナーで作成したエンティティ自体に関しては、WCF のシリアル化に対応しているので、引数や戻り値として、単体のエンティティ インスタンスのやり取りは可能です。また、エンティティの集合であるエンティティ セットはシリアル化できませんが、エンティティの配列やコレクション（List<T>など）としてはシリアル化が可能であり、サービスとの間でやり取りできます。

この AWTS サンプル プログラムの予約管理サービス（AdwkRService プロジェクト）では、一部のサービスで B のパターンを使用しており、エンティティの配列を引数としてやり取りするメソッドもあります。

なお、この A と B のパターンを比較した場合、この A のパターンでは、WCF サービス側で ObjectContext を常時キャッシュするわけではなく、変更追跡などの負荷は各クライアントに分散し、サービス側のリソースの節約になります。ただし、クライアントは、ObjectContext に相当する変更追跡などの実装（①）が必要になるので、ある程度、クライアントのシステム要件に制約を受けます。

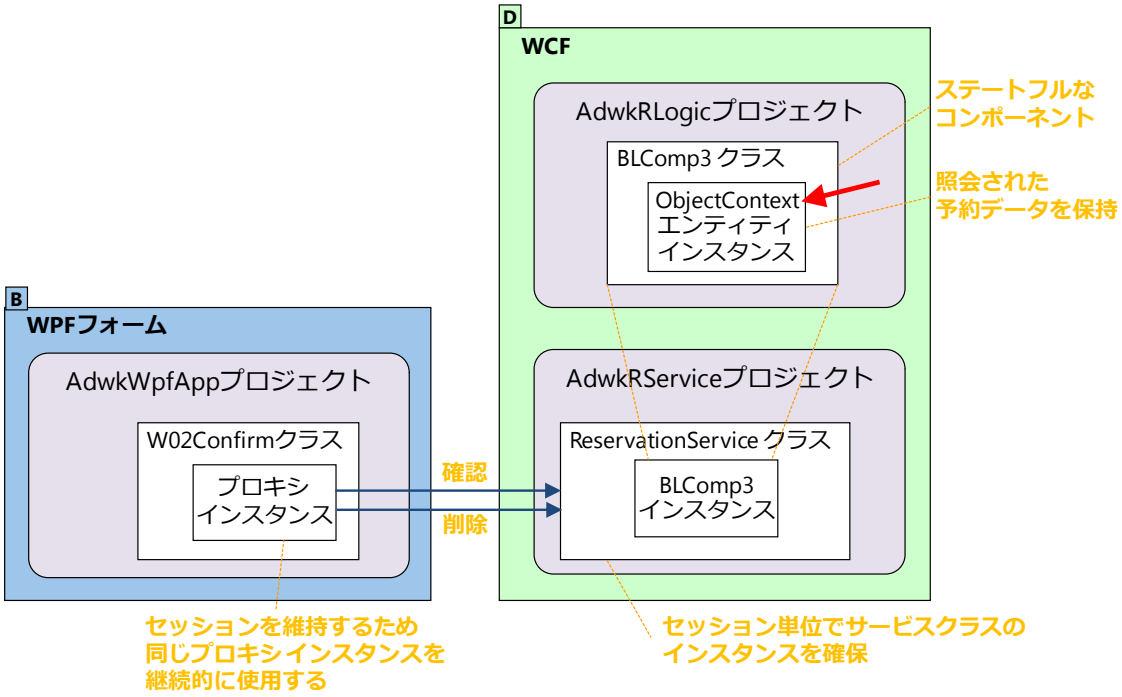
一方、B のパターンでは、ObjectContext を WCF サービス側でキャッシュするので、サービス側の実装が複雑になり、サービス側の負荷も増えます。（クライアントのセッション数だけ、ObjectContext のインスタンスがサービス内に存在します。）しかし、クライアントは軽量になり、環境の制約も少ないので、サービスの相互運用性が向上します。

サンプルに見る ObjectContext およびエンティティの利用

実際の AWTS サンプル プログラムの中で、前述の図 63 の B のパターンの例を確認しましょう。

予約管理サービスでは、セッション単位でステートを維持しており、そのステートの 1 つとして、ObjectContext を維持しています。この構成は、既に図 26 で取り上げています。以下に再掲します。

図 64. （図 26 の再掲）セッション単位のステート管理における ObjectContext の維持



ObjectContext 自身（赤矢印）は、BLComp3 コンポーネントの中に埋め込まれていますが、AdwkRService プロジェクトの ReservationService クラスの中で、BLComp3 インスタンスをセッション単位で維持しています。既に説明したように、ReservationService クラスのサービスでは、照会から削除までの一連の処理を 1 つのセッションの中で行い、照会した時点で ObjectContext にキャッシュされたデータは、クライアントの削除要求によって削除されます。このときは、クライアントとのやり取りの引数には、エンティティのシリアル化などは特に必要はなく、例 55 のように、一般的な引数がコントラクト（IReservation）に使用されいます。

例 55. ReservationService クラスで使用されているコントラクト（抜粋）

IReservation.cs（AdwkRService プロジェクト）

```
[ServiceContract(
    Namespace="http://Adwk.Reservation",
    SessionMode=SessionMode.Required)]
public interface IReservation
{
    [OperationContract]
    bool QuerySchedule(    // ←照会
        DateTime fromDate, DateTime untilDate,
        string deptCode, string arrvCode, string levelCode,
        out ScheduleInfo[] sinfos,
        out string systemMessage);

     （略）

    [OperationContract]
    bool DeleteReservationByRCode(    // ←削除
        string reservationCode,
```

```
        out string systemMessage);

    (略)

}
```

また、同じ AdwkRService プロジェクトには、予約に関わる経路などの基本情報を返す別のサービスとして、RBasicInfoService クラスがあります。このサービスのコントラクトでは、次のように、EDM のエンティティである「場所」や「クラス」が使用されています。既に触れたように、EDM デザイナーによって自動生成されたエンティティ自体は、WCF サービスのシリアル化に対応しているので、改めてデータコントラクトを定義する必要はありません。また、これらは IEnumerable<T>や配列などのコレクションとしてやり取りできることも分かります（List<T>なども利用できます）。

例 56. EDM のエンティティを引数として使用（抜粋）

IBasicInfo.cs（AdwkRService プロジェクト）

```
// 予約管理サービスの基本情報を返すサービス
[ServiceContract(Namespace="http://Adwk.Reservation")]
public interface IRBasicInfo
{
    [OperationContract(Name="GetBasicRouteInfo1")]
    bool GetBasicRouteInfo(
        out IEnumerable<場所> places, out IEnumerable<クラス> levels);

    [OperationContract(Name="GetBasicRouteInfo2")]
    bool GetBasicRouteInfo(
        out 場所 [] places, out クラス [] levels);

    [OperationContract(Name="GetBasicRouteInfo3")]
    bool GetBasicRouteInfo(
        out PlaceInfo[] places, out LevelInfo[] levels);
}
```

なお、このようなコレクションをやり取りするサービスに対して、クライアント側で「サービス参照の追加」を行う際には、コレクションの型を別途指定する必要があります。

というのは、サービス参照の追加では、サービスが提供するメタデータからは、クライアントにコレクションの正確な型は伝わらず、コントラクトに使用するコレクションの型が、配列であろうが、IEnumerable<T>であろうが、List<T>であろうが、既定の構成では、プロキシ側の引数は配列になります。

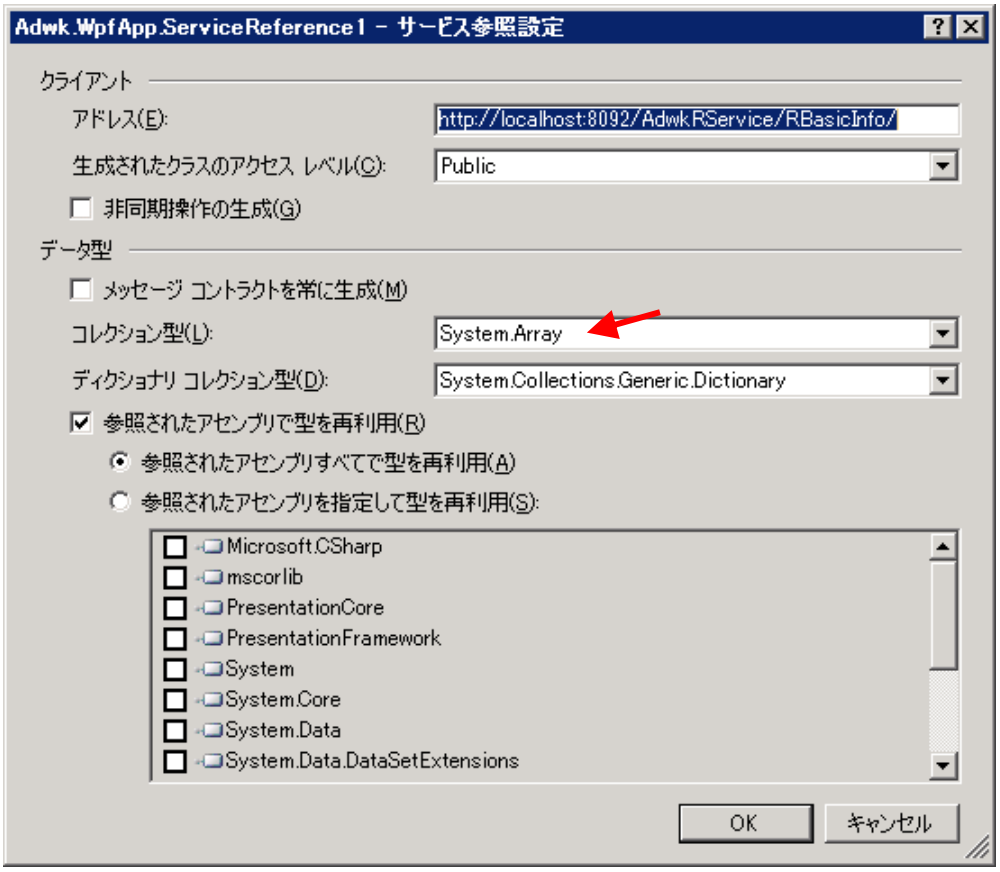
プロキシ側のコレクションの型は、サービス参照の追加の構成画面の中で変更できます。例 56 のコントラクトのサービスを参照する AdwkWpfApp プロジェクトでは、既定の構成でプロキシを作成しているので、プロキシ側では配列として扱っています。この設定は、次の手順で確認や指定ができます。

1. AdwkRService プロジェクトを実行します。（AdwkWpfApp プロジェクトでのサービス参照の追加の際に、メタデータを再取得するために必要です。）
2. AdwkWpfApp プロジェクトを開きます。

3. ソリューション エクスプローラーのツリー上で、［Service References］ノードの配下にある［ServiceReference1］ノードを右クリックして、ショートカットメニューから［サービス参照の構成］をクリックします。

4. すると、次図のように［Adwk.WpfApp.ServiceReferece1 - サービス参照設定］ダイアログボックスが表示されます。このダイアログボックスの中間あたりにある、コレクション型の指定を確認します。必要があれば変更し、［OK］ボタンをクリックします。（この例では変更する必要はありません。）

図 65. プロキシ生成時のコレクションの指定



註: 既存の「サービス参照の追加」の成果物に対して、コレクションの型を変更するのではなく、新規に「サービス参照の追加」を行う際に、コレクションの型を指定するのであれば、［サービス参照の追加］ダイアログボックスの左下部にある［詳細設定］ボタンをクリックしてください。すると、図 65 と同じダイアログボックスが表示されます。

上記の図 65 では、「System.Array」と指定してあるので、プロキシ側ではコレクションは配列として生成されます。このサンプルのプロキシのソースコードには、クライアント側のコントラクトの定義として、次のインターフェイスが生成されています。コレクションの型が、どれも配列であることが分かります。

例 57. プロキシ側のコントラクトでは配列として引数を生成（抜粋）

Reference.cs（非表示ファイル、AdwkWpfApp プロジェクト）

```
public interface IRBasicInfo {
```

```
    bool GetBasicRouteInfo1(out Adwk.WpfApp.ServiceReference1.場所[] places,
        out Adwk.WpfApp.ServiceReference1.クラス[] levels);

    bool GetBasicRouteInfo2(out Adwk.WpfApp.ServiceReference1.場所[] places,
        out Adwk.WpfApp.ServiceReference1.クラス[] levels);

    bool GetBasicRouteInfo3(out Adwk.WpfApp.ServiceReference1.PlaceInfo[] places,
        out Adwk.WpfApp.ServiceReference1.LevelInfo[] levels);
}
```

註: EDM のエンティティを引数や戻り値としてシリアル化する場合、注意すべき点があります。EDM デザイナーでは、エンティティ間の 1 対多などの関連付け（アソシエーション）を設定することができ、設定することによって、関連付いたエンティティを参照するナビゲーション プロパティが既定で生成されます。たとえば、図 62 の「クラス」エンティティの下部には、1 対多に関連付いた相手を参照する「運行予定明細」プロパティがあり、このプロパティを介して、関連付いたエンティティを参照できます。
.NET Framework 4 からは、このようなナビゲーションプロパティでは、参照することによって、自動的にデータベースから ObjectContext へ、関連付いたデータがロードできるように設定できます。このためには、ObjectContext.ContextOptions.LazyLoadingEnabled プロパティを true に設定します。（EDM デザイナーで新規作成したモデルでは、既定で true です。）
しかし、このプロパティを true にしておくと、特定のエンティティをシリアル化する際に、ナビゲーションプロパティに関連付いたエンティティまで一緒にシリアル化されます。さらに、関連付いたエンティティにナビゲーションプロパティがあると、イモづる式に読み込まれます。このため、転送量が必要以上に大きくなります。
今回の「場所」や「クラス」エンティティでは、そのエンティティ自身だけが必要なので、このような関連付いたエンティティの自動的な読み込みとシリアル化は不要です。よって、サンプルではエンティティを返す際に、このプロパティを false に設定しています。たとえば、AdwkRLogic プロジェクトの BLComp1.cs に含まれる GetBasicRouteInfo メソッドの内部では、「場所」や「クラス」のエンティティを返す前に、次のように、このプロパティを false に設定しています。

```
// ナビゲーションプロパティは必要ないので、
// 参照した際に読み込まないようにする
oc.ContextOptions.LazyLoadingEnabled = false;
```

4.5 データ バインディングの利用

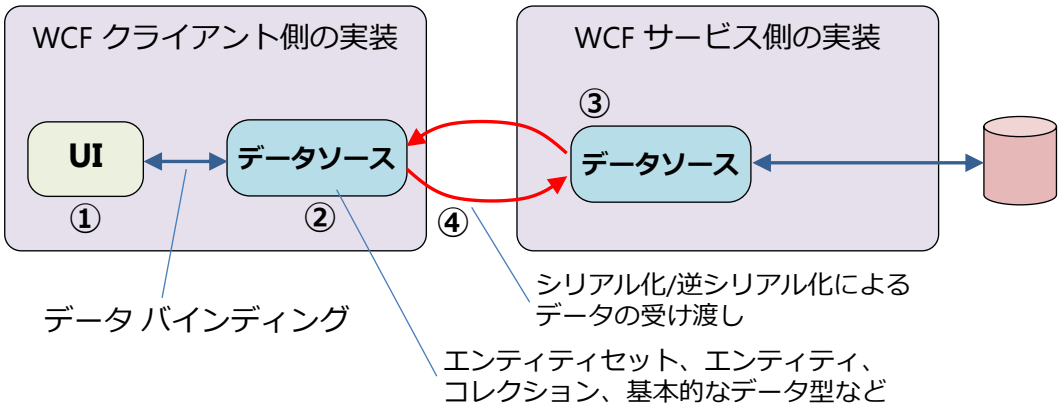
この節では、WCF サービスにおいて、データ バインディングを使用する場合の形態や特徴について確認します。データ バインディング自体は、WCF とは切り離して利用できるテクノロジなので、データ バインディング自体の説明は割愛します。このあとは、WCF における典型的な利用形態を説明したのち、AWTS サンプル プログラムの中での利用例について確認します。

註: データ バインディングの利用方法などの詳細については、以下のアドレスも参照してみてください（使用するユーザーインターフェイスによって、実装方法が異なります）。
http://msdn.microsoft.com/ja-jp/library/ms752347.aspx (WPF)
http://msdn.microsoft.com/ja-jp/library/cc278072(VS.95).aspx (Silverlight)
http://msdn.microsoft.com/ja-jp/library/ef2xyb33.aspx (Windows フォーム)
http://msdn.microsoft.com/ja-jp/library/ms228214.aspx (ASP.NET Web サーバー コントロール)

WCF サービスにおけるデータ バインディングの利用形態

次の図 66 では、WCF サービスとデータ バインディングの観点から、そのシステム構成を表しています。

図 66. WCF サービスとデータ バインディング



データ バインディングは、①のユーザー インターフェイスを伴うクライアント側で行われ、データ バインディングに使用されるデータソースも、②のようにクライアントに存在します。

一般に分散システムでは、このようなデータソースのデータ（②）は、データベースから取り込んだり、逆にデータベースに反映したりする必要があるので、③や④のように WCF サービスを介することとなります。

よって、このような形態でデータバインディングを使用する条件としては、④のように対象のデータは、WCF のシリアル化がサポートされ、クライアントとサービスの間で、やり取りできる必要があり、また、クライアントは②のデータを保持できるような環境でなければなりません。

この図 66 のデータバインディングを利用できるシステム構成としては、図 60 や図 63 の A と B のパターンなど、どれも該当します。

たとえば、図 60 の構成の場合は、図 66 に当てはめると、クライアント側のデータソース（図 66 の②）は、型指定されたデータセットになります。データセットに対して、特別な実装を追加していなくとも、クライアント側のデータ バインディングで利用できます。

図 63 の A のパターンの構成であれば、図 66 に当てはめると、クライアント側のデータソース（図 66 の②）は、DataServiceContext や DomainContext が扱うエンティティセットなどが該当し、WCF Data Services や WCF RIA Services を利用することで実現できます。

また、図 63 の B のパターンの構成であれば、図 66 に当てはめると、クライアント側のデータソースは、一般的なデータ型（または、そのコレクション）、EDM のエンティティ（または、そのコレクション）、そのほか、WCF のシリアル化に対応したデータなどが挙げられます。

サンプルに見るデータ バインディング

データ バインディングの指定方法の詳細は割愛しますが、サンプルのどこで使用されているか、簡単に確認しておきましょう。

図 60 の型指定されたデータセットを用いたデータ バインディングの例としては、図 56 および図 57 で取り上げた ASP.NET Web ページ上の GridView コントロールでのデータ バインディングが該当します。この Web ページ（MileageHistory.aspx）の分離コード ファイル（MileageHistory.aspx.cs）の中で、次のようにデータバインディングを行っています。

例 58. WCF サービスから取り込んだデータセットをデータ バインディングで使用（抜粋）

MileageHistory.aspx.cs（AdwkWebApp プロジェクト）

```
    （略）

// 選択された月に基づいてマイル履歴を構築する
private void MakeupMonthHistory()  //←[1]
{

    （略）

    // マイル管理サービスにアクセスして、マイル履歴を取得
    var proxy = new MileageQuery2Client();
    AdwkMStatus stat;
    string msg;
    AdwkMileagesDataSet.会員マイル履歴DataTable historyTable;
    historyTable = proxy.QueryHistoryByMemberAndPeriod(  //←[2]
        out stat, out msg,
        memberInfo.MemberCode, startDate, untilDate);

    // データバインドを行う
    // 履歴がない場合でも、0件のテーブルとバインドすることでクリアできる。
    HistoryGridView.DataSource = historyTable;     //←[3]
    HistoryGridView.DataBind();                    //←[4]
    Session[SVars.HistoryList] = historyTable; //特定月の履歴を退避

    // メッセージの表示
    string resultMsg = msg;
    if (historyTable.Rows.Count <= 0)
    {
        // データがゼロ件の場合は、その旨のメッセージを追加する
        resultMsg = msgNoData + " " + resultMsg;
    }
    ShowMessage(resultMsg);
}
```

図 57 のマイルの履歴を表示する Web ページでは、初回表示時や、年月のドロップダウン リストから月を選択した際に、上記の例 58 の[1]にある MakeupMonthHistory メソッドが呼び出されます。このメソッドの中で、データ バインディングを行っています。

特に難しいことはありません。[2]で WCF サービスを呼び出して、型指定されたデータテーブル（変数 historyTable）を受け取った後、[3]と[4]のように、通常の ASP.NET のデータ バインディングを使用して、Web ページ上の GridView コントロール（変数 HistoryGridView）に対して、型指定されたデータテーブル（変数 historyTable）をバインドしています。

もう 1 つ例を確認しましょう。

すでに取り上げた図 55 の WPF ベースの［運行予定照会/予約］ウィンドウでは、上部の「経路」ドロップダウンリストや、「クラス」ドロップダウンリストでは、データソースとして配列データを用

いたデータ バインディングを行っています。元になる配列データも、WCF サービスから取り寄せたものです。このパターンは、図 63 の B のパターンに該当します。

この部分の実装コードは、例 53 の中で既に使用していました。例 53 のその部分だけを、以下に再掲します（ソースコード内の番号は割り振り直しています）。

例 59. WCF サービスから取り込んだ配列データをデータ バインディングで使用（抜粋）

W01Reserve.xaml.cs（AdwkWpfApp プロジェクト）

```
    （略）

// 呼び出し完了時のイベント
private void OnEndCall(IAsyncResult ar)  //←[1]
{
    try
    {
        bool result;
        result = proxy.EndGetBasicRouteInfo3(out places, out levels, ar); //←[2]
        this.Dispatcher.Invoke(new Action(PopulateUIAfterCall));
        proxy.Close();
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message);
        proxy.Abort();
    }
}

// 呼び出し完了後にUIを設定する
private void PopulateUIAfterCall()
{
    // 出発地のリストを作成
    deptCombo.DisplayMemberPath = "PlaceName"; //←[3]
    deptCombo.SelectedValuePath = "PlaceCode"; //←[4]
    deptCombo.ItemsSource = places;            //←[5]
    deptCombo.SelectedIndex = 0;

    （略）

}
```

この例では、WCF サービスを非同期で呼び出しているので、サービス呼び出しの完了時のイベントで駆動する[1]の OnEndCall メソッドの中で、[2]のように WCF サービスの EndGetBasicRouteInfo3 メソッドを呼び出し、配列データとして places と levels を取得しています。

そして、[3]から[5]は、通常の WPF でのデータ バインディングの方法を使用して、「経路」の左側のドロップダウンリスト（変数 deptCombo）に、配列 places をバインドしています。

その結果、図 55 の［運行予定照会/予約］ウィンドウのようにウィンドウのドロップダウンリストに経路情報（場所名のリスト）が表示されます。

4.6 非接続でのデータ操作とオプティミスティック同時実行制御

プログラムからデータベースを扱う際に、データベースからデータを一旦メモリに読み込んで、データベースとの接続を断ち、メモリ上のデータをいくつか変更した後、最後にまとめてデータベースに書き込む方法があります。このような方法は「非接続型アプローチ」と呼ばれており、この方法によって、データベース サーバーの負荷を軽減し、スケーラビリティを高めることができます。

特に、WCF サービスなどの分散環境のシステムでは、クライアントのプログラムがデータベースに直接的に接続することがないこともあり、このような非接続型アプローチが、見受けられます。

たとえば、図 60 のデータセット、また、図 63 の EDM を使用した例も、非接続型アプローチです。これらの例で使用されたデータセットや、ObjectContext、DataServiceContext、DomainContext などのオブジェクトは、非接続の状態でデータをメモリ上にキャッシュできるオブジェクトです。どのオブジェクトの場合も、データベースからデータを読み込んでキャッシュしたのち、そのキャッシュに対して変更を加えると、変更も追跡され記憶されます。そして、それらの変更追跡は、最後にまとめてデータベースへ反映されます。

このような非接続型アプローチでは、データの更新の際に注意すべき点があります。

たとえば、2 台の異なるクライアントが、同じ購入予約のデータをそれぞれ読み込んでキャッシュしたとします。そして、一方のクライアントでは、キャッシュした予約データに対してキャンセル処理を行い、もう一方では、キャッシュした予約データに対して発注手続きを行い、発注完了のフラグを設定したとします。このとき、一方のクライアントがキャッシュした更新内容（予約キャンセル済み）をデータベースに書き込んだのち、もう一方のクライアントがキャッシュした変更内容（発注済み）を書き込もうとしたとき、不都合が生じます。この場合、更新されたデータベースの内容（予約キャンセル済み）と、これから更新しようとするキャッシュの内容の、どちらが正しいか、一概には判断できません。

というのも、2 つのクライアントが行った更新内容は、それぞれ、元の状態の予約データが存在していることが前提になっており、2 つの更新が相反するからです。キャンセルしたものは、発注手続きはできませんし、一方で、予約に対して発注完了のものは、予約自体のキャンセルも不自然です（もしキャンセルするなら、予約データではなく、発注処理をキャンセルすべきです）。このような、データに相反する複数の状態が存在する状況を「データの競合」といいます。

このような状況では、どのように対処するかは、システムの要件や状況によっても異なりますが、少なくとも、データの競合を検出する仕組みは必要です。EDM でも、データの競合を検出する仕組みが用意されています。

なお、このような非接続型アプローチにおいて、キャッシュしているデータに対してクライアントが編集している間、データベース上のデータはロックされずに他からの変更が可能な状態にして、両者の更新を制御する形式を「オプティミスティック同時実行制御」と呼んでいます。

このような「データの競合」の問題や「オプティミスティック同時実行制御」は、WCF 固有のテーマではなく、データ アクセス テクノロジ全般に関わる点であり、ADO.NET のデータセット（およびテーブルアダプター）や LINQ-to-SQL などでも、オプティミスティック同時実行制御の仕組みが用意

されています。ここでは、のちほど例の 1 つとして、ObjectContext や EDM を使用した場合での、オプティミスティック同時実行制御について確認していきます。

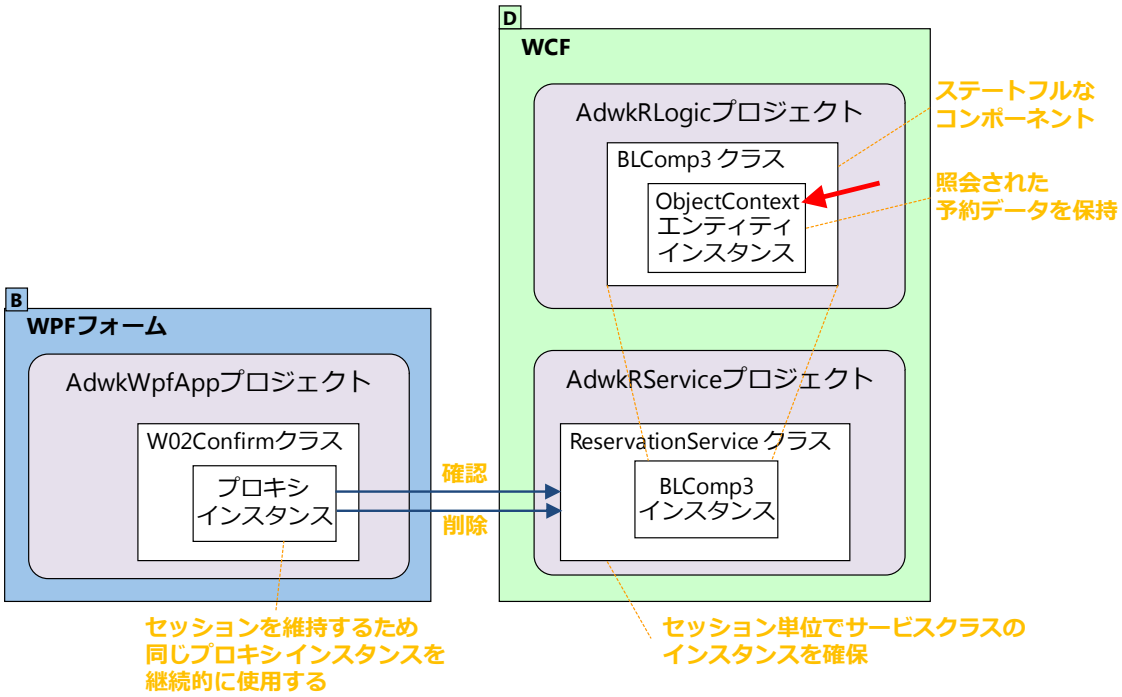
> **註:** その他のデータの競合やオプティミスティック同時実行制御については、データアクセス関連の、以下の自習書も参考にしてみてください。このアドレスのデータアクセス自習書には、各種テクノロジのデータの競合に関する記述もあります。
> http://msdn.microsoft.com/ja-jp/data/gg615417 データ アクセス自習書
> 「データ アクセス自習書 (Visual Studio 2010/.NET4 版)」の中の
> 「第 3 部 SQL Server データ アクセス手法」にある
> (1) ADO.NET 編、(2) LINQ (統合言語クエリ) 編、(3) ADO.NET Entity Framework 編、
> および、(4) WCF Data Services 編

> **註:** WCF サービスにおけるデータアクセスの方法として、非接続型アプローチが唯一の方法ではありません。従来からある接続型アプローチも利用できます。
>
> たとえば、クライアントが WCF サービスのメソッドを呼び出したとき、その 1 回の要求/応答の過程の中でのみ、データベースへの接続を維持し、トランザクションを実行して、在庫の減算や発注処理などの一連のデータ操作を行い、トランザクションを完了して接続を終了のち、応答を返す方法です。
>
> この接続型アプローチでは、非接続型アプローチよりもスケーラビリティの面で劣ることがありますが、一般に接続型アプローチの中で使用される 1 つのトランザクション中で、データがロックされるため、他のユーザーが変更することがなく、データの整合性を維持できます。

サンプルに見るオプティミスティック同時実行制御

図 64 で取り上げたように、AdwkRService プロジェクトでの WCF サービスでは、セッション単位でステートを持ち、そのセッションの中で ObjectContext を保持しています（以下の図 67 に再掲示）。

図 67. （図 64 の再掲）セッション単位のステート管理における ObjectContext の維持



この図の ObjectContext（赤矢印の部分、BLComp3 クラスの中のメンバー変数）では、クライアントが予約データを参照した際に、データベースに接続して、予約データを読み込みでキャッシュします。キャッシュしたのち、データベースとの接続を断ち、非接続の状態になります。

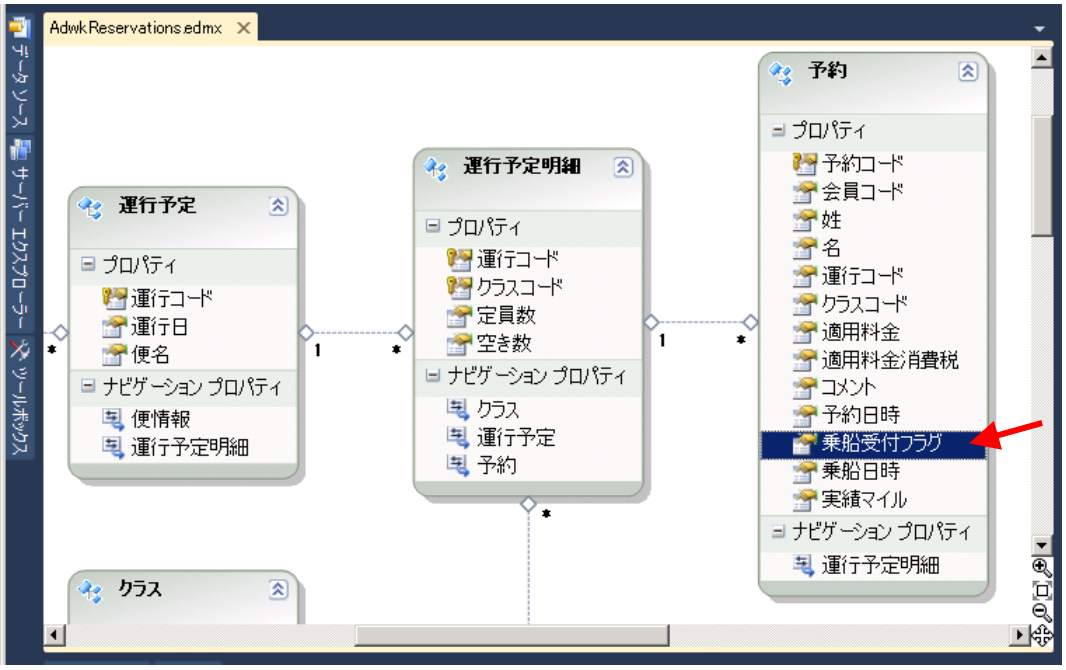
さらにクライアントが予約の削除操作を行った場合には、そのキャッシュしたデータに対して、削除操作を行います。その操作は、ObjectContext の中で追跡され、キャッシュ内に記録されます。そして、データベースに再接続して、この変更の記録をもとに、データベースを更新します。

このサンプルでは、ObjectContext がデータベースを更新する際には、データベース上の元のデータが、他のクライアントによって既に更新されていないか確認して、データの競合を検出するように構成されています。もし、既に他のクライアントによって更新されていた場合、データの競合として ObjectContext が例外を発生するようになっています。

ただし、既定オプションのままだと ObjectContext はデータの競合を検出せずに、常に、後から更新したほうが、その時点のデータベースのデータを上書きします。更新の際に、ObjectContext がデータの競合を検出できるようにするには、予め EDM デザイナーを用いて列単位で指定する必要があります。今回の例では、簡単にするため、最低限の設定をしており、次の方法で確認できます。

1. AdwkRService ソリューションを開きます。
2. そのソリューションに含まれる AdwkRDac プロジェクト内の、AdwkReservations.edmx をダブルクリックして、EDM デザイナーで開きます。
3. 次図のように、EDM デザイナー内の「予約」エンティティの「予約受付フラグ」列をクリックして選択します。

図 68. EDM デザイナーで予約エンティティを確認



4. ここで、プロパティ ウィンドウが表示されていない場合は、［F4］キーを押すなどして、プロパティ ウィンドウを表示させます。

5. プロパティ ウィンドウに表示された「予約」エンティティのプロパティ一覧の中から、「同時実行モード」プロパティを見つけ、その値が「Fixed」であることを確認します。

図 69. 同時実行モード

既定では、この同時実行モードの値が「None」になっており、データの競合を検出せずに、後からデータベースに書き込んだほうがデータベースを上書きします。データの競合を検出するには「Fixed」に設定します。

上記の確認をした場合は、EDM デザイナーを閉じておきましょう。AdwkRService ソリューション自体は、この後も使用します。

次に、データ競合を検出して例外に対処するコードを確認してみましょう。次の例は、BLComp3 クラスの中で、予約の削除を行うメソッドの部分です。

例 60. ObjectContext による予約削除とデータ競合の検出（抜粋）

BLComp3.cs（AdwkRLogic プロジェクト）

```
    public bool DeleteReservationByRCode(  //←[1]
        string reservationCode,
        out AdwkRStatus stat, out string systemMessage)
    {

     (略)
                            //↓[2]
        using (var ts = new TransactionScope(TransactionScopeOption.Required))
        {

     (略)

                // 対象予約を削除としてマーク
                // (簡単にするため、予約データ自体を削除)
                adwkRObjectContext.予約.DeleteObject(currentReservation); //←[3]

                try
                {
                    // 更新
                    adwkRObjectContext.SaveChanges();  //←[4]
                }
                catch (OptimisticConcurrencyException)  //←[5]
```

```
            {
                stat = AdwkRStatus.OptimisticConcurrencyError;
                systemMessage = msgOptimisticConCurrencyError;  //←[6]
                return false;
            }
            catch
            {
                stat = AdwkRStatus.UndefinedError;
                systemMessage = msgDeleteReservationError;
                return false;
            }

            // 正常終了に投票(Vote)
            ts.Complete();  //←[7]
```

この例の[1]にある DeleteReservationByRCode メソッドは、BLComp3 クラスのメソッドです。このクラスのメンバー変数には、メモリ上の ObjectContext を表す adwkRObjectContext が既に用意されており、この DeleteReservationByRCode メソッドの中で、必要に応じて利用しています。

たとえば[3]では、変数 adwkRObjectContext に対して、予約データを削除しています。この時点で、メモリ上の ObjectContext には、予約データの削除操作が記録されます。

そして、[4]の SaveChanges メソッドの呼び出しによって、ObjectContext に記録されたすべての変更が、まとめてデータベースに反映されます。このとき、データの競合がないか検証され、競合が発生した場合には、OptimisticConcurrencyException という例外が発生し、この例では、[5]の catch ブロックでその例外が補足されます。そして、クライアントに返すシステム メッセージとして、[6]のように、定数 msgOptimisticConCurrencyError（"システムエラーです。(0302)"）が設定され、返されます。

なお、[4]の SaveChanges メソッドによる更新は、[2]の TransactionScope からも分かるようにトランザクションに参加できます。ここでは、[6]のようにデータ競合の例外を捕捉した場合は、[7]のトランザクションを正常終了させる Complete メソッドを呼び出さないので、ObjectContext による更新は中止できます。

サンプルでのオプティミスティック同時実行制御の動作確認

それでは、実際に前述のサンプル（BLComp3 クラス）を使用して、オプティミスティック同時実行制御の効果を確認してみましょう。

ここでは、2 つのクライアント（AdwkWpfApp）から同一の予約データに関して、相反する変更を行ってみます。この 2 つをそれぞれ「クライアント A」、「クライアント B」と呼ぶことにします。

まず、以下の手順でクライアント A から予約データを作成します。

1. AdwkRService プロジェクト、および、AdwkMService プロジェクトを実行します。
2. クライアント A として、AdwkWpfApp プロジェクトのアプリケーションを実行します。
3. ［予約管理・乗船手続き］メイン ウィンドウが表示されたら、ウィンドウ左側の中間あたりにある［運行予定照会/予約］ボタンをクリックします。すると、次図のように［運行予定照会/予約］ウィンドウが表示されます。

図 70. ［運行予定照会/予約］ウィンドウ

4. 経路やクラス、照会期間は、前図のように既定のままにして、［運行予定照会］ボタンをクリックし、運行予定データを WCF サービス（AdwkRService）から取得します。

5. すると、次図のように運行予定が一覧表示されるので、先頭行（運航日が 2011/07/01 の 0001 便、クラスは一等）を選択します。そして、姓と名の欄には、それぞれ「山田」、「太郎」、会員コードには「0000100001」と入力して、［予約］ボタンをクリックします。

図 71. 運行予定を選択して予約データを入力

6. すると予約が完了して、次図のように予約番号が表示されるので、その番号を控えておきます。（予約番号は、このサンプルの使用状況によって異なります。）

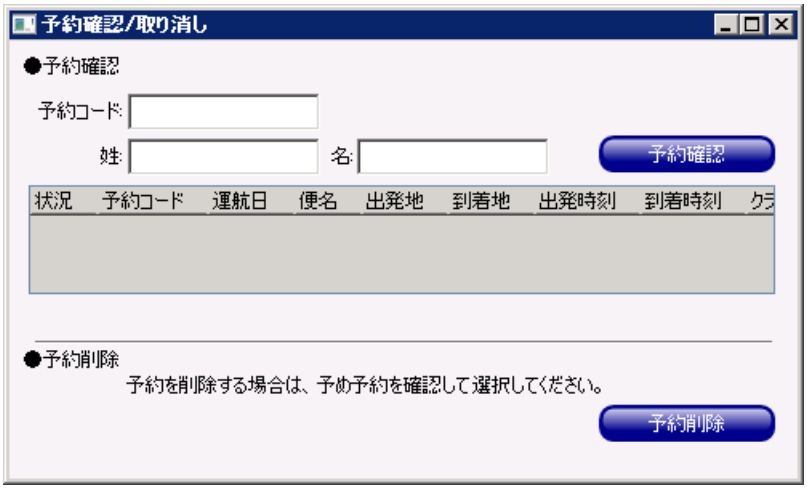
図 72. 予約完了（予約番号を控えること）

7. 予約が済んだら、この［運行予定照会/予約］ウィンドウのみ閉じておきます。（クライアント A としての AdwkWpfApp のアプリケーションは起動したままにしておきます。）

次に以下の手順に従って、クライアント B を起動して、同じ予約データをキャッシュします。

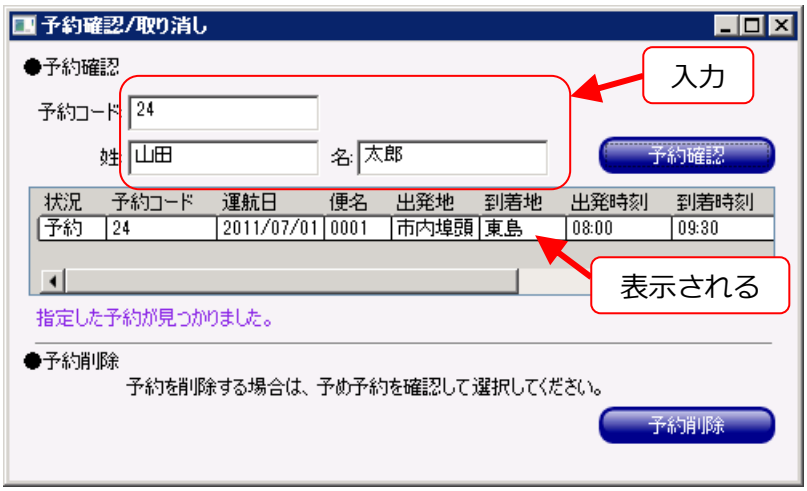
8. 同じ AdwkWpfApp プロジェクトから、もう 1 つ別のアプリケーションをクライアント B として起動します。
9. ［予約管理・乗船手続き］メイン ウィンドウが表示されたら、ウィンドウ右側の中間あたりにある［予約確認/取り消し］ボタンをクリックします。すると、次図のように［運行予定照会/予約］ウィンドウが表示されます。

図 73. 予約確認



10. 次図のように「予約コード」欄には、前の手順 6 で控えた予約番号を入力し、姓に「山田」、名に「太郎」と入力して、［予約確認］ボタンをクリックし、予約データを表示させます。

図 74. 先に作成したものと同じ予約を読み込んで表示



11. クライアント B では、照会した予約データをこのまま表示しておきます。

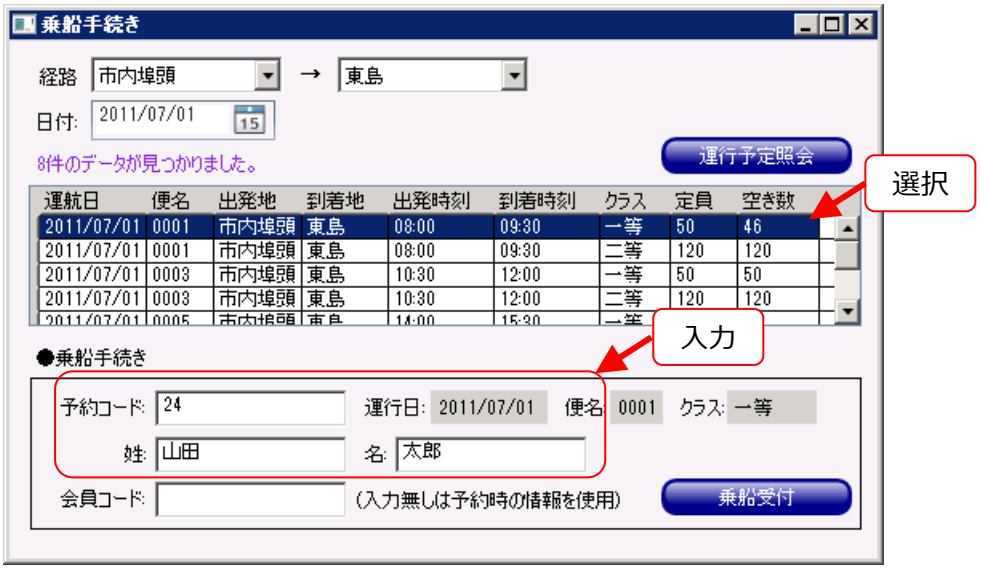
ここまでのところで、クライアント B である AdwkWpfApp アプリケーションから、予約管理サービス（AdwkRService）を呼び出し、その結果、サービス側にある AdwkRLogic プロジェクト内の、BLComp3 クラスのメンバーである ObjectContext の中に、この予約データがキャッシュされました。ここで次の手順に従って、クライアント A から同じ予約データの乗船手続きを行います。

12. クライアント A の［予約管理・乗船手続き］ウィンドウに戻って、左側下部の［乗船手続き］ボタンをクリックします。すると、次図のように［乗船手続き］ウィンドウが表示されます。

図 75. 乗船手続き

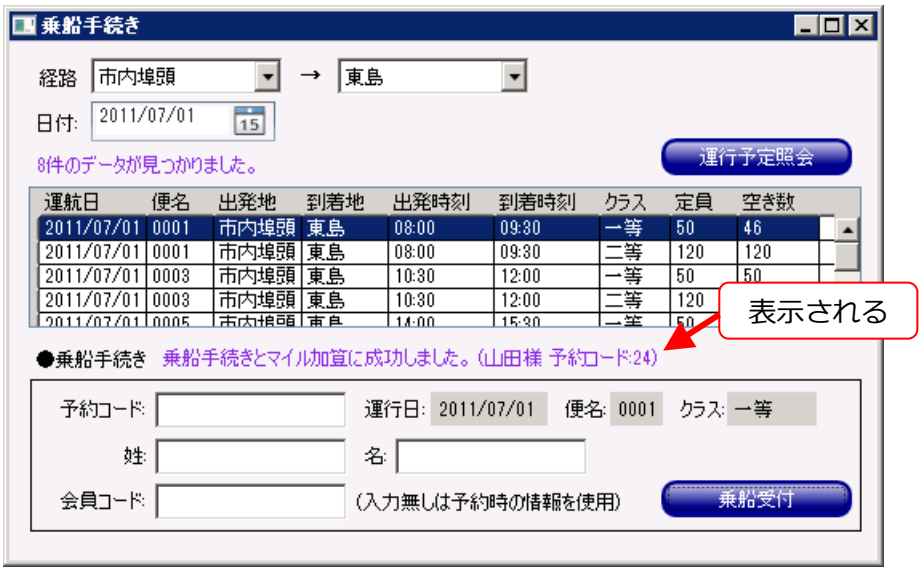
13. 経路や日付は既定のままにして、［運行予定照会］ボタンをクリックします。次図のように運行予定の一覧が表示されたら、先頭の行（運航日 2011/07/01、便名 0001、クラスは一等）を選択した後、予約コードには控えていた同じ予約番号を入力し、姓と名にそれぞれ、「山田」、「太郎」と入力し、［乗船受付］ボタンをクリックします。

図 76. 乗船手続きのデータを入力



14. すると、予約管理サービス（AdwkRService）を呼び出して乗船手続きを行い、次図のように手続きが正常に成功した旨のメッセージが表示されることを確認します。

図 77. 乗船手続き成功

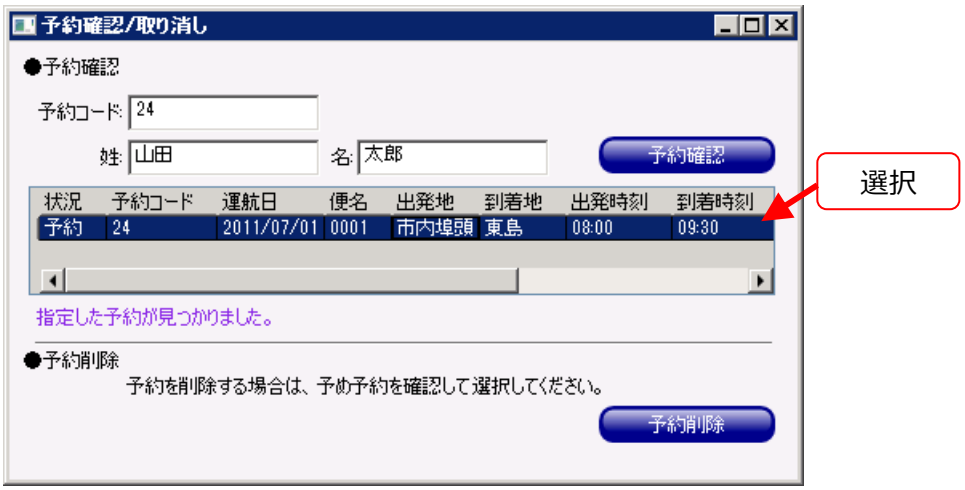


これで、クライアント A は予約管理サービス（AdwkRService）を介して、データベースの予約データの乗船手続きを完了し、「乗船受付フラグ」列のスタータスを完了状態に変更しました。この時点で、クライアント A によって更新されたデータベースの情報（乗船済み）と、クライアント B のキャッシュされた情報（予約の状態）に食い違いが生じています。

ここで次の手順に従って、さらにクライアント B から予約の削除（予約のキャンセル）を行ってみましょう。

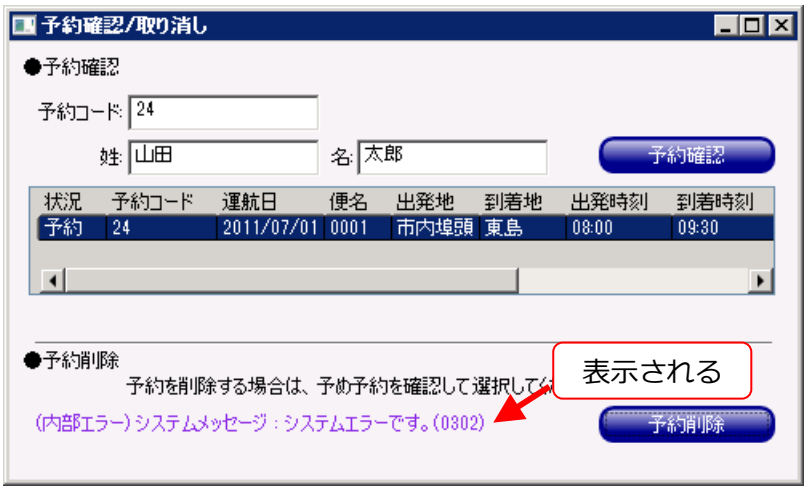
15. クライアント B の［予約確認/取り消し］ウィンドウに戻ります。次図に示すように、現在照会中の予約データを選択して、［予約削除］ボタンをクリックし、削除を行ってみます。

図 78. 予約を選択して削除を試行



16. ［予約削除の確認］メッセージボックスが表示されたら、［はい］をクリックします。
17. すると、クライアント B は予約管理サービスを呼び出し、サービス側でデータの競合を感知します。次のように、クライアント B にはエラーメッセージが表示されることを確認します。

図 79. データの競合を検出



クライアント B（AdwkWpfApp）は、予約管理サービス（AdwkRService）の ReservationService サービスクラスを介して、AdwkRLogic プロジェクトにある BLComp3 の DeleteReservationByRCode メソッド（例 60）を呼び出します。今回は、データの競合が検出され、OptimisticConcurrencyException 例外がスローされて、例 60 の[5]の catch ブロックで捕捉されました。その結果、[6]のエラーメッセージとして、定数 msgOptimisticConCurrencyError（"システムエラーです。(0302)"）が返されました。

この結果、この例では、乗船手続きが完了済みであるとして、予約の削除は行われません。

以上、このサンプルにおけるデータの競合の流れを確認しました。

註: 余力があれば、図 68 の EDM デザイナーで、図 69 に示す「同時実行モード」を「Fixed」から「None」に変更して、前述の手順 1 から 17 までを行てみてください。データの競合を検出できないので、乗船手続き完了の予約データも、削除できてしまいます。

まとめ

以上、第 2 部では WCF の具体的な実装方法や関連する Visual Studio 2010 の使用方法について取り上げました。ここでは、実際のアプリケーションの中で、WCF 関連のテクノロジをどう活用するのかを確認していただくために、主に AWTS サンプルプログラムを題材にして、重要な実装例や、そのコードに至る Visual Studio の操作のポイントなどを確認しました。また、バリエーションとして、一部のサンプルについては、簡単なものを新規作成しました。

ここでは、WCF テクノロジが提供するすべての機能を網羅的に解説したわけではありませんが、ここで取り上げたことから、WCF を使いこなす上での「着眼点」や「指針」、「コツ」など掴み取っていただければ幸いです。