CL-04 サーバ通信機能

■ 概要

TERASOLUNA フレームワークでは、通信基盤として、WCF(Windows Communication Foundation)を採用する。WCF を利用することで、プラットフォームや通信プロトコルによって実装スタイルを変えることなくサーバ通信を実現できる。
本機能は、サーバと接続するためのクライアント通信クラス(WCF クライアント)と、クライアント通信クラスを開発支援するための機能を提供する。

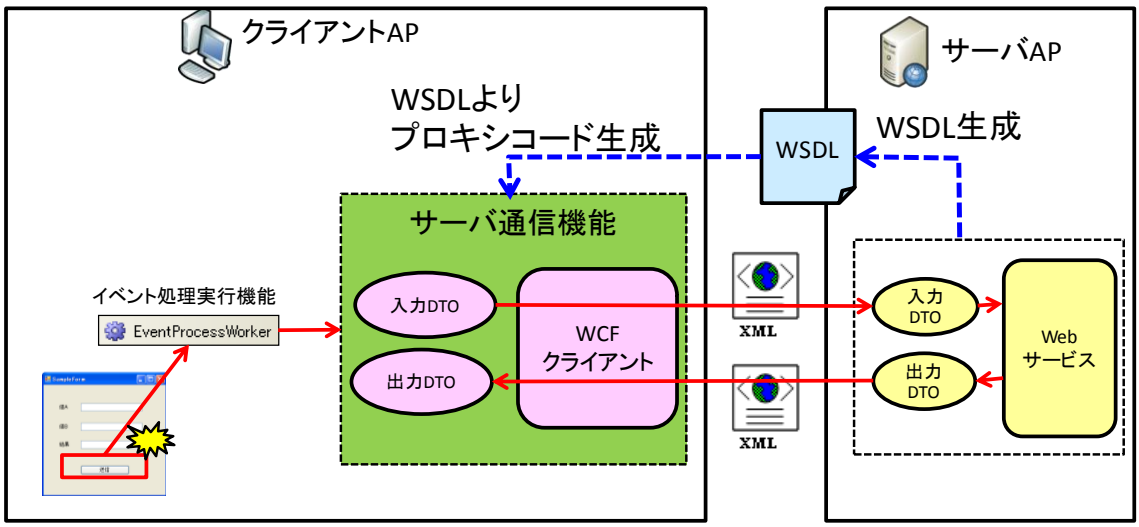


図 1 サーバ通信機能の動作概念図

通常の WCF による開発と同様に、Visual Studio の「サービス参照の追加」(Svcutil.exe)機能を利用して、サーバ側(Web サービス)から取得した WSDL(Web Services Description Language)をもとにプロキシコード(WCF クライアント)を生成する。

- WCF クライアントの呼び出し
  TERASOLUNA フレームワークでは、「CL-03 イベント処理実行機能」の処理フローの「ビジネスロジック実行」フェーズにおいて WCF クライアントを呼び出し、サーバと通信することができる。
  WCF クライアントの呼び出し方法は、以下の2通りである。
  - クライアント処理を伴わないサーバ通信
    - EventProcessWorker に、実行対象の WCF クライアントを設定することで、対象となる Web サービスメソッドを自動的に(追加実装なしで)呼び出す。
  - クライアント処理を伴うサーバ通信
    - 開発者が実装したクライアントビジネスロジック内で WCF クライアントを生成し対象の Web サービスメソッドを明示的に呼び出す。
    - 呼び出し方法は、通常の WCF の実装方法と同様。

- 接続先のサーバアドレスの一括置換
  WCF では、Web サービスの接続先（エンドポイント）の設定をアプリケーション構成ファイル（App.config）に記述する。エンドポイントの設定には、接続ホストの URL 情報も含まれる。また、接続先サーバが一つであっても、複数のサービスクラスやバインディング方式が存在する場合、エンドポイントを複数作成する必要がある。
  TERASOLUNA フレームワークでは、AP 共通構成ファイル(TerasolunaApplication.config)に、URL の置換設定をすることで、接続先のサーバアドレスを一括置換する機能を提供する。これにより、製造工程、試験工程、運用工程などの環境に応じたエンドポイントアドレス（URI）の置換作業を大幅に軽減することができる。

- ファイルアップロード、ダウンロード
  WCF の標準機能を利用し、MTOM(Message Transmission Optimization Mechanism)などのエンコーディング方式によりサーバと送受信する。ファイルデータは、バイト配列やストリームで定義する。

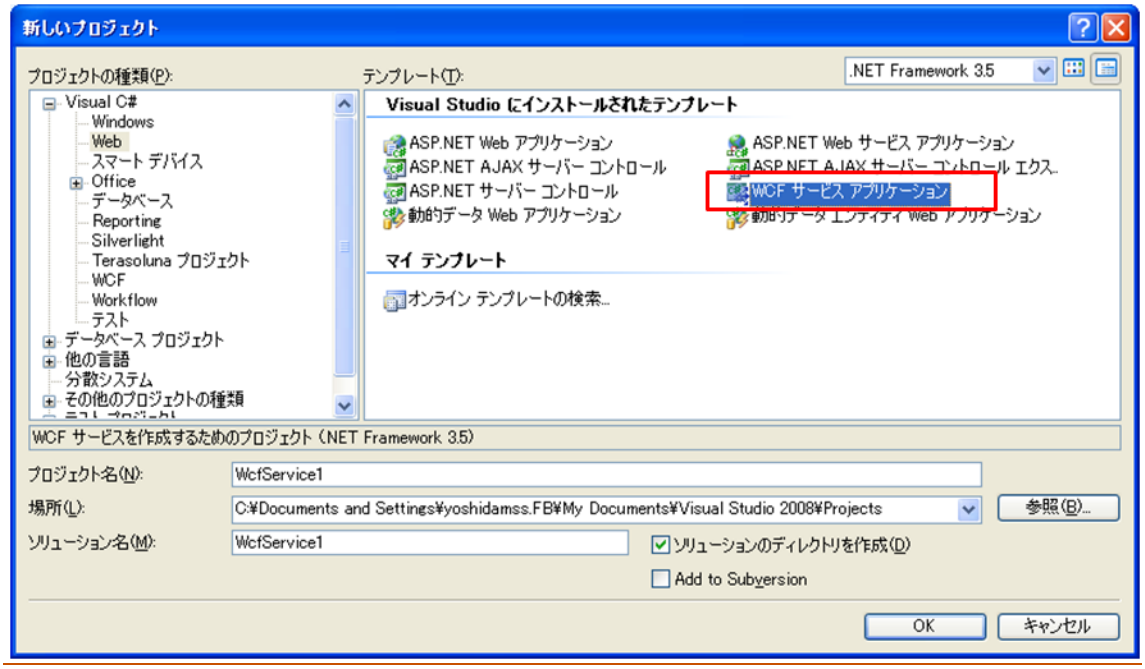
■ 使用方法

◆ Web サービスの作成

(1) サーバ AP が.NET(WCF サービス)の場合

サーバ AP として TERASOLUNA フレームワークを利用する場合の WCF サービス作成方法は、「SV-01 WCF サービス管理機能」の機能説明書を参照すること。

WCF をそのまま利用する場合は、Visual Studio で「WCF サービスアプリケーション」(または「WCF サービスライブラリ」)で WCF サービスを作成する。WCF サービスの作成方法については、MSDN のドキュメント等を参照のこと。



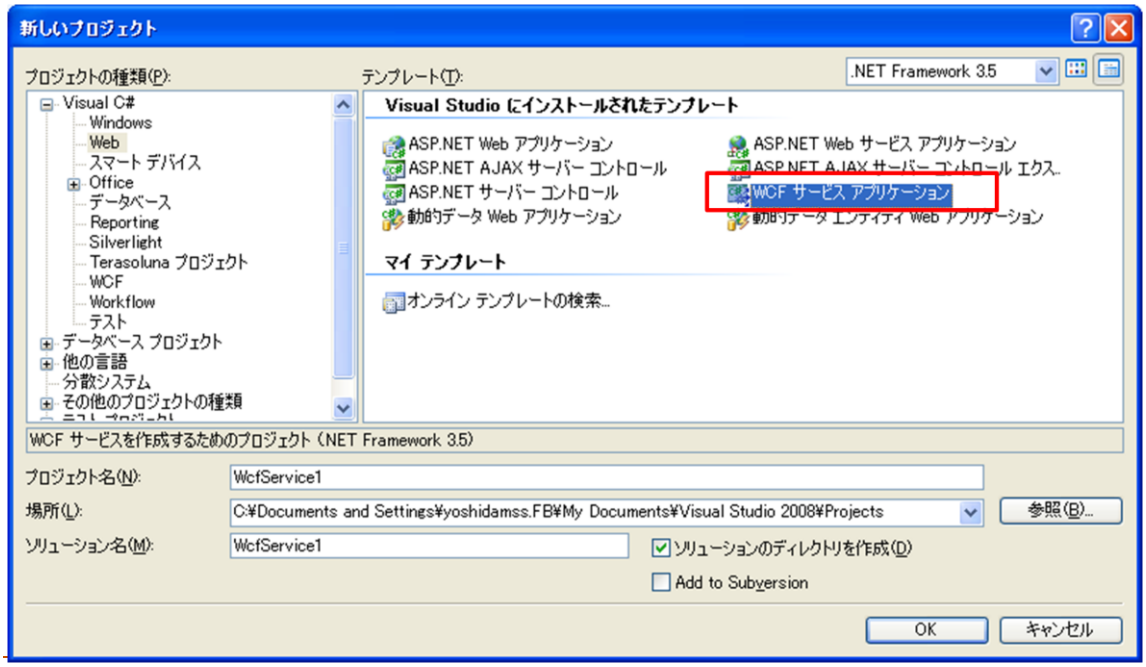


図 2 WCF サービスアプリケーションの作成

以下に、WCF のサービスコントラクトとデータコントラクトの記述例を示す。

```
// サービスコントラクト
 [ServiceContract]
 public interface IA01Serivce
 {
      [OperationContract]
      void ExecuteA01_01_01_S01(A01_01_01_S01InputDto input);

 }

// データコントラクト
 [DataContract]
 public class A01_01_01_S01InputDto
 {
      public string TourCode { get; set; }
      public string TourName { get; set; }
 }

 //サービスの実装クラス
 public class A01Serivce : IA01Serivce
 {

      public void ExecuteA01_01_01_S01(A01_01_01_S01InputDto input)
      {
          // ビジネスロジックの実行
      }
 }
```

図 3 WCF サービスの実装例

また、エンドポイントの設定を Web 構成ファイル(Web.config)に記述する。

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior name="SampleWcfService.A01ServiceBehavior">
        <serviceMetadata httpGetEnabled="true" />
        <serviceDebug includeExceptionDetailInFaults="false" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <services>
    <service behaviorConfiguration="SampleWcfService.A01ServiceBehavior"
      name="SampleWcfService.A01Service">
      <endpoint address="" binding="wsHttpBinding" contract="SampleWcfService.IA01Service">
        <identity>
          <dns value="localhost" />
        </identity>
      </endpoint>
      <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
    </service>
  </services>
</system.serviceModel>
```

リスト 1 サーバ側の Web 構成ファイル(Web.config)の記述例

(2) サーバ AP が Java(JAX-WS)の場合

Web サービスクラスには、@WebService アノテーションを付与する。また、公開するメソッドには、@WebMethod アノテーションを付与する。
詳細は、JAX-WS の仕様書[1]や Metro[2]のドキュメント等を参照のこと。
以下に、Web サービスクラスの実装例を示す。

```
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService(serviceName = "A01Service", targetNamespace =
"http://jp.terasoluna.toursample")
public class A01Service {

    @WebMethod(operationName = "ExecuteA01_01_01_S01")
    public void executeA01_01_01_S01(A01_01_01_S01InputDto input)
            throws SampleSoapFaultException {
        // ビジネスロジックの実行
    }
}
```

リスト 2　JAX-WS による Web サービスの実装例

JAX-WS を使用する場合の注意点として、Web サービスのメソッド引数となる JavaBean(DTO)について、フィールドまたはプロパティ(getter メソッド)に、@XmlElement アノテーションを付与し name 属性に頭文字が大文字の名前を明記すること。
こうすることで、.NET のクライアント側で生成されるプロキシコードの入出力 DTO のプロパティが name 属性で指定したものと同じ(頭文字が大文字の)名前になる。
@XmlElement アノテーションの name 属性を明記しないと、WSDL から生成されるクライアントの入出力 DTO の各プロパティ名の頭文字が小文字になってしまうが、通常、.NET ではパスカル形式(頭文字が大文字)でプロパティを定義するため、画面クラスのプロパティ名とプロキシコードの入出力 DTO の名前が一致せず、「CL-03 イベント処理実行機能」において、画面データクラスと DTO 間でのデータコピーが行われなくなる。
画面データクラスと DTO 間でのデータコピーを確実に実施させるためには、必ず@XmlElement アノテーションの name 属性を明記し、画面クラスのプロパティ名と同名になるようにすること。
以下に、サーバ側 DTO の実装例を示す。

---

[1] JSR 224: JavaTM API for XML-Based Web Services (JAX-WS) 2.0
http://jcp.org/en/jsr/detail?id=224

[2] JAX-WS RI や WSIT(Web Service Interoperability Techonology)等を含むオープンソースの Web サービススタック。
https://metro.dev.java.net/

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "A01_01_01_S01InputDto", namespace = "http://jp.terasoluna.toursample")
public class A01_01_01_S01InputDto {
    @XmlElement(name = "TourCode", namespace = "http://jp.terasoluna.toursample")
    private String tourCode;

    @XmlElement(name = "TourName", namespace = "http://jp.terasoluna.toursample")
    private String tourName;

    public String getTourCode() {
        return tourCode;
    }
    public void setTourCode(String tourCode) {
        this.tourCode = tourCode;
    }

    public String getTourName() {
        return tourName;
    }

    public void setTourName(String tourName) {
        this.tourName = tourName;
    }
}
```

リスト 3 サーバ側 DTO クラスの実装例

また、JAX-WS RI(Metro)の場合、エンドポイントの設定を WEB-INF/sun-jaxws.xml に記述する。以下に記述例を示す。

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'  version='2.0'>
  <endpoint name='CalcService'
    implementation='jp.terasoluna.rich.tutorial.service.calc.service.CalcService'
    url-pattern='/CalcService' />
</endpoints>
```

リスト 4 sun-jax-ws.xml の記述例

◆ WSDL の生成

作成した Web サービスクラスから WSDL を生成する。
簡単な方法としては、WCF サービス(または JAX-WS Web サービス)を起動して、特定の URL(エンドポイントのアドレスに"?wsdl"を付与したもの)を指定することで、WSDL を取得することができる。
また、コマンドラインからも WSDL を生成し取得することができる。(WCF サービスの場合は、"Svcutil.exe"、JAX-WS Web サービスであれば、"wsgen") 詳細は、MSDN や JAX-WS RI(Metro)のドキュメント等を参照のこと。

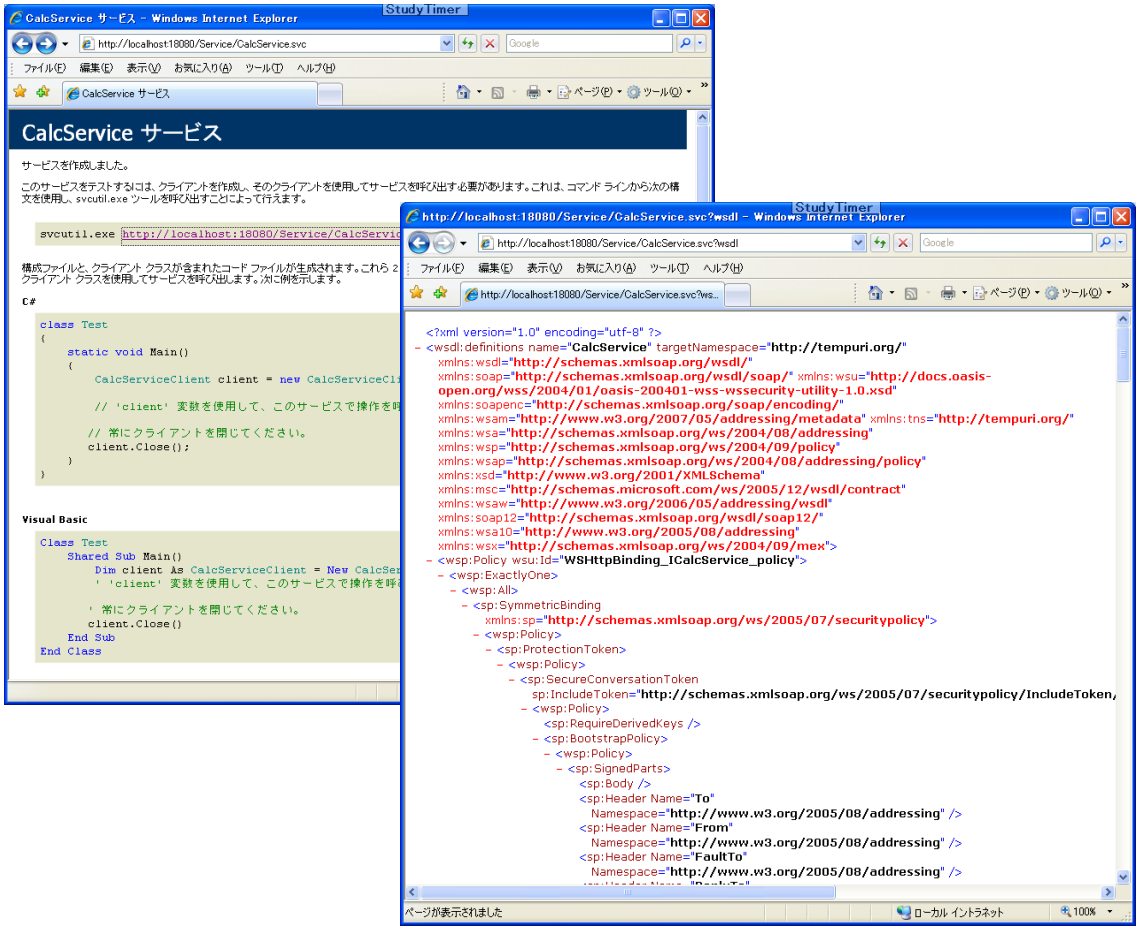


図 4 生成された WSDL(WCF サービス)

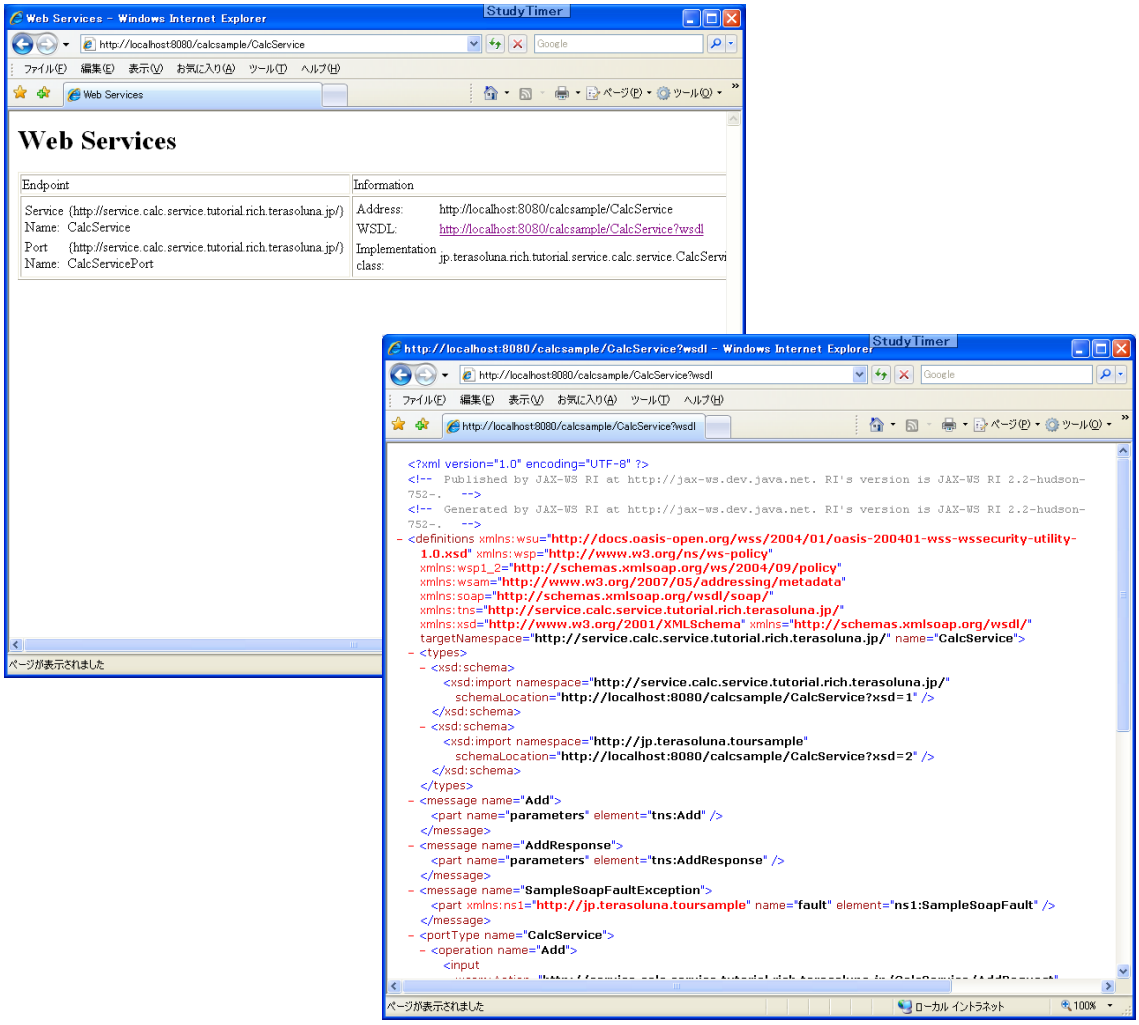


図 5 生成された WSDL(JAX-WS Web サービス)

◆ WCF クライアントの生成

Visual Studio の「サービス参照の追加」により、WSDL の取得先を指定し、WCF クライアントを生成する。また、Svcutil.exe を使ってコマンドラインからも生成することができる。
WCF クライアント生成の詳細な手順は、MSDN のドキュメント[3]等を参照のこと。

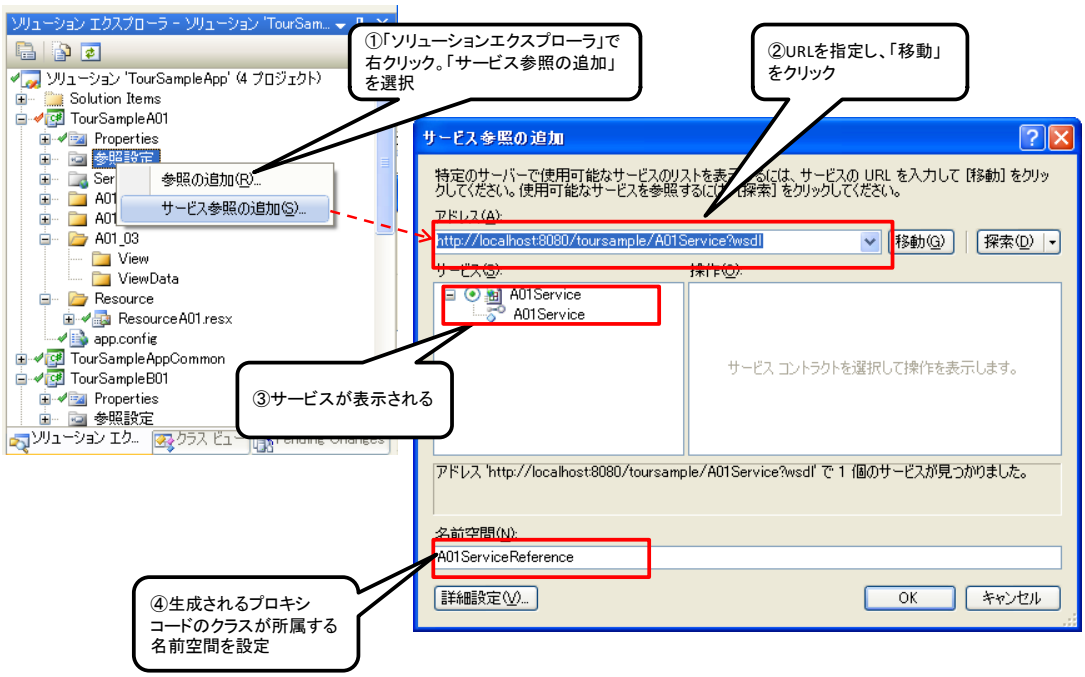


図 6 「サービス参照の追加」の操作イメージ

なお、「サービス参照の追加」を実施したプロジェクトのアプリケーション構成ファイル（App.config）に、エンドポイントの設定が自動生成されるが、実行時はエントリポイントのあるプロジェクトのアプリケーション構成ファイルでの設定が必要となる。
自動生成されたアプリケーション構成ファイルをベースに、適宜プロジェクトの要件に合わせてパラメータ値を変更し、エントリポイントのあるプロジェクトのアプリケーション構成ファイルを記述すること。

[3] Visual Studio での WCF サービスの使用
http://msdn.microsoft.com/ja-jp/library/bb907581.aspx

```
  <system.serviceModel>
    <bindings>
      <wsHttpBinding>
        <binding name="WSHttpBinding_ICalcService" closeTimeout="00:01:00"
            openTimeout="00:01:00" receiveTimeout="00:10:00" sendTimeout="00:01:00"
            bypassProxyOnLocal="false" transactionFlow="false"
                hostNameComparisonMode="StrongWildcard"
            maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
            messageEncoding="Text" textEncoding="utf-8" useDefaultWebProxy="true"
            allowCookies="false">
          <readerQuotas maxDepth="32" maxStringContentLength="8192" maxArrayLength="16384"
              maxBytesPerRead="4096" maxNameTableCharCount="16384" />
          <reliableSession ordered="true" inactivityTimeout="00:10:00"
              enabled="false" />
          <security mode="Message">
            <transport clientCredentialType="Windows" proxyCredentialType="None"
                realm="" />
            <message clientCredentialType="Windows" negotiateServiceCredential="true"
                algorithmSuite="Default" establishSecurityContext="true" />
          </security>
        </binding>
      </wsHttpBinding>
・・・
    </bindings>
    <client>
      <endpoint address="http://localhost:18080/Service/CalcService.svc"
          binding="wsHttpBinding" bindingConfiguration="WSHttpBinding_ICalcService"
          contract="CalcWcfServiceReference.ICalcService" name="WSHttpBinding_ICalcService">
        <identity>
          <dns value="localhost" />
        </identity>
      </endpoint>
・・・
    </client>
  </system.serviceModel>
```

リスト 5 クライアント側のアプリケーション構成ファイル（App.config）の記述例

◆ WCF クライアントの呼び出し

(1) クライアント処理を伴わないサーバ通信

このケースでは、図 7 ~~図 7~~のように、EventProcessWorker の BizLogicExecution プロパティのビジネスロジック情報設定画面で、ビジネスロジック種別に「WcfProxy」を選択し、WCF クライアントのクラス名、定義名(エンドポイント名)、メソッド名、を指定することで、ロジック等の追加実装をすることなく、イベント実行時にサーバ通信処理を実行することができる。なお、この方法で WCF クライアントを呼び出す場合、「接続先サーバアドレスの一括置換」は自動的に有効となる。

EventProcessWorker のプロパティ設定方法の詳細は、「CL-03 イベント処理実行機能」の機能説明書を参照のこと。

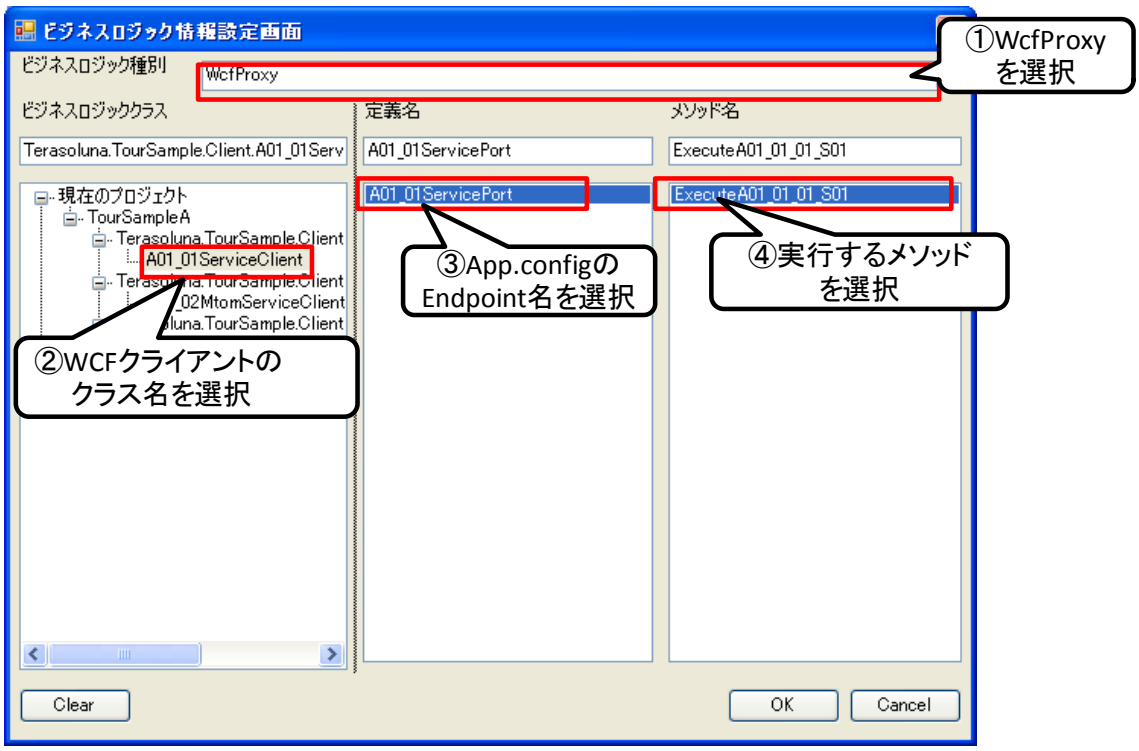


図 7 EventProcessWorker.BizLogicExecution プロパティの設定

(2) クライアント処理を伴うサーバ通信

このケースでは、開発者が作成したビジネスロジッククラスから WCF クライアントを呼び出す。ビジネスロジッククラスは、「CL-03 イベント処理実行機能」により実行されるため、EventProcessWorker の BizLogicExcecution プロパティのビジネスロジック情報の設定画面で、ビジネスロジック種別を「ClientBizLogic」を選択し、ビジネスロジックのクラス名、メソッド名等を指定する。EventProcessWorker のプロパティ設定方法の詳細は、「CL-03 イベント処理実行機能」の機能説明書を参照のこと。

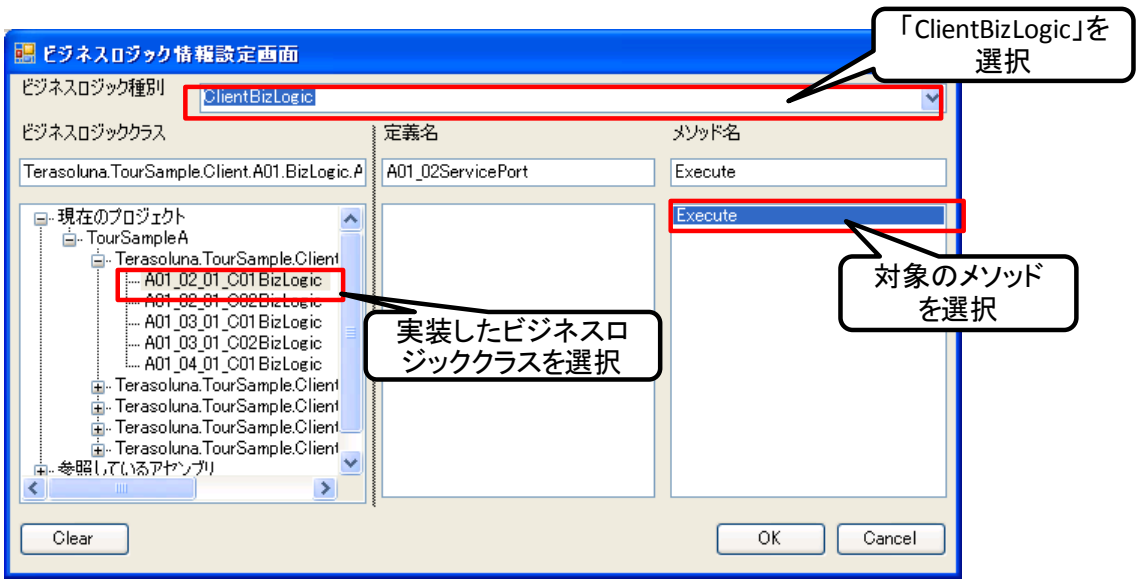


図 8　EventProcessWorker.BizLogicExecution プロパティの設定

クライアントビジネスロジックから WCF クライアントを呼び出す方法は、通常の WCF を使った開発と同様である。ただし、「接続先サーバアドレスの一括置換」を利用する場合は、CommunicationManager クラスを使ってエンドポイントアドレスを取得する。CommunicationManager クラスによる実装手順は以下のとおりである。。

- ビジネスロジッククラスの中で、IUnityContainer インタフェースのプロパティを定義し、Dependency カスタム属性を付与する。
- アプリケーション構成ファイル（App.config）に記述されたエンドポイント名と IUnityContainer オブジェクトを引数として、CommunicationManager.GetReplacedEndpointAddress メソッドを呼び出し、置換後の URI を取得する。
- 置換後の URI をもとに作成した EndpointAddress オブジェクトとエンドポイント名を引数として WCF クライアントを作成し、対象のメソッドを呼び出す。

以下に CommunicationManager の使用例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
・・・
    <client>
      <endpoint address="http://localhost:8080/toursample/A01_02Service"
        binding="basicHttpBinding" bindingConfiguration="A01_02ServicePortBinding"
        contract="A01_02ServiceReference.A01_02Service" name="A01_02ServicePort" />
    </client>
  </system.serviceModel>
</configuration>
```

リスト 6 App.config 上のエンドポイントアドレスの例

```
    public class A01_02_01_C01BizLogic
    {
        /// App.configに記述されたエンドポイント名
        private const string EndpointName = "A01_02ServicePort";

        /// IUnityContainer型のプロパティを定義。Dependency属性を付与。
        [Dependency]
        public IUnityContainer Container { get; set; }

        public A01_02_01_C01OutputDto Execute(A01_02_01_S01InputDto inputDto)
        {
            ・・・
            A01_02_01_S01OutputDto outputDto = null;
            A01_02ServiceClient client = null;
            try
            {
                ///エンドポイントアドレスの取得
                ///WCF接続先変更機能により、接続先サーバアドレスを一律置換してくれる。
                EndpointAddress address =
CommunicationManager.GetReplacedEndpointAddress(Container, EndpointName);
                //WCFサービスの実行
                client = new A01_02ServiceClient(EndpointName, address);
                outputDto = client.ExecuteA01_02_01_S01(inputDto);
                client.Close();
            }
            catch (System.Exception)
            {
                if (client != null)
                {
                    client.Abort();
                }
                throw;
            }
            return AfterCommunicate(outputDto);          }
・・・
    }
```

リスト 7 クライアントビジネスロジッククラスでの WCF クライアント呼び出しの記述例

◆ AP 共通構成ファイル(TerasolunaApplication.config)の記述

本機能を利用するためには、AP 共通構成ファイル(TerasolunaApplication.config)の /configuration/unity/containers/container/extensions/add タグで、以下の UnityContainerExtension 継承クラスを記述する。

- Terasoluna.ServiceModel.CommunicationExtension クラス
  - 本機能を利用するために必要なデフォルトの Extension

その他、「CL-03 イベント処理実行機能」の Extension クラスの設定が必要なため、設定方法について「CL-03 イベント処理実行機能」の機能説明書を併せて参照すること。

また、「接続先サーバアドレスの一括置換」の設定として、
Terasoluna.ServiceModel.IAddressReplacer インタフェースの実装クラスを記述する。
標準では、IAddressReplacer の実装クラスとして、以下のクラスを設定する。

- Terasoluna.ServiceModel.MappingAddressReplacer クラス
  - FromUri プロパティの値を ToUri プロパティに文字列置換する IAddressReplacer 実装クラス
  - FromUri プロパティには、App.config 上に記述された置換前の URI を設定する。
  - ToUri プロパティには、置換後の URI を設定する。
  - Order プロパティの値の順番(0, 1, 2, …)で最初に置換可能な IAddressReplacer 実装クラスを使用した置換処理が適用される。

以下に TerasolunaApplication.config の設定例を示す。
この設定により、App.config のエンドポイントのアドレス(「http://localhost:8080/toursample/TourService」)を、
「http://192.168.1.1/toursample/TourService」に一括置換することができる。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
・・・・
  <unity>
    <typeAliases>
      <!-- エイリアスの設定 -->
      <typeAlias alias="int" type="System.Int32" />
      <typeAlias alias="Uri" type="System.Uri, System, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" />
      <typeAlias alias="IAddressReplacer"
            type="Terasoluna.ServiceModel.IAddressReplacer,
                  Terasoluna" />
      <typeAlias alias="MappingAddressReplacer"
                type="Terasoluna.ServiceModel.MappingAddressReplacer,
                  Terasoluna" />
    </typeAliases>
    <containers>
      <container>
        <types>
          <!-- サーバ通信のアドレス置換設定 -->
          <type name="localhostToDnsName"
                      type="IAddressReplacer" mapTo="MappingAddressReplacer">
            <lifetime type="singleton" />
            <typeConfig>
              <property name="Order" propertyType ="int">
                <value value="0" type="int"/>
              </property>
              <property name="FromUri" propertyType ="Uri">
                <value value="http://localhost:8080/" type="Uri"/>
              </property>
              <property name="ToUri" propertyType ="Uri">
                <value value="http://192.168.1.1/" type="Uri"/>
              </property>
            </typeConfig>
          </type>
        <extensions>
・・・
          <!-- サーバ通信信機能の設定 -->
          <add type="Terasoluna.ServiceModel.CommunicationExtension,
                    Terasoluna" />
        </extensions>
      </container>
    </containers>
  </unity>
```

リスト 8 AP 共通構成ファイル(TerasolunaApplication.config)の記述例

◆ ファイルアップロード・ダウンロード処理の実装

WCF では、大規模データの取扱について Text エンコーディング（Base64 エンコーディング）と MTOM エンコーディングをサポートしている。詳細は MSDN のドキュメント等[4]を参照のこと。また、JAX-WS も MTOM エンコーディングをサポートしている。JAX-WS における MTOM の実装方法は、JAX-WS の仕様書や Metro のドキュメント等を参照のこと。
、サーバ AP に TERASOLUNA フレームワークを利用する場合は、「SV-01 WCF サービス管理機能」の機能説明書も合わせて参照すること。
以下に、MTOM を使ったファイルアップロード・ダウンロードの実装例を示す。

（1）サーバ AP が.NET(WCF サービス)の場合

【ファイルデータをバイト配列で扱う場合】
サーバ側 DTO(DataContract)に byte[]型のプロパティを定義して、DataMember 属性を付与する。以下に、DTO の実装例を示す。

```
[DataContract]
public class TransferedFile
{
    /// ファイル名
    [DataMember]
    public string FileName { get; set; }

    /// ファイルのデータ
    [DataMember]
    public byte[] FileData { get; set; }
}
```

リスト 9　サーバ側 DTO(DataContract)の実装例

次に、上記で作成した DTO を引数または戻り値とするメソッド(OperationContract)を定義した WCF サービス(ServiceContract)を作成する。以下に、WCF サービスの実装例を示す。

```
[ServiceContract]
public interface IMtomService
{
    ///ファイルアップロードの例
    [OperationContract]
    void Upload(TransferedFile trasferdFile);

    ///ファイルダウンロードの例
    [OperationContract]
    TransferedFile Download(DownloadInputDto inputDto);
}
```

リスト 10 WCF サービスインタフェースの実装例

---

[4] 大規模データとストリーミング
http://msdn.microsoft.com/ja-jp/library/ms733742.aspx

MTOM を有効化するには、Web 構成ファイル(Web.config)で、WCF サービスのエンドポイントに対するバインディングの設定を変更する必要があり、binding 要素の messageEncoding 属性の値を”Mtom”に変更する。
また、その他必要に応じて、maxReceivedMessageSize 属性や、readerQuotas 要素の maxArrayLength の値も変更し、受信可能なデータサイズを調整すること。

以下に Web.config の設定例を示す

```
  <system.serviceModel>
    <bindings>
      <wsHttpBinding>
        <!-- messageEncodingにMtomを指定する -->
       <binding name="MtomBinding" messageEncoding="Mtom"
                                ...      maxReceivedMessageSize="10000000">
         <readerQuotas ... maxArrayLength="10000000" />
       </binding>
      </wsHttpBinding>
    </bindings>
...
  </system.serviceModel>
```

リスト 11　Web.config の設定例

WCF サービスの実装が完了したら、クライアント AP 側で、「サービス参照の追加」を実施する。
なお、クライアントのアプリケーション構成ファイルも、binding 要素の messageEncoding 属性を”MTOM”に設定し MTOM を有効化する。(「サービス参照の追加」で自動生成されたものは、通常、”MTOM”が設定されている)また、その他、必要に応じて、maxReceivedMessageSize 属性や、readerQuotas 要素の maxArrayLength の値も変更し、受信可能なデータサイズを調整すること。

【ファイルデータをストリームで扱う場合】

大量のデータを転送する場合など、メッセージ全体をメモリ内で処理する代わりに「ストリーミング転送モード」を使用することができる。ストリーミング転送モードを有効にした状態では、WCF の機能に制約があるため、MSDN のドキュメント等を参照し、プロジェクトの要件に応じて利用可能か十分に検討すること。

ストリーム化されたデータは Stream 型で定義する。ストリーム化されたデータとともに追加情報を転送する場合は、DTO をメッセージコントラクトとして(MessageContract 属性を付与して)作成し、追加情報を保持するプロパティは MessageHeader 属性を付与することでメッセージヘッダーに格納する。また、ストリーム化されたデータは、MessageBodyMember 属性を付与して、メッセージ本体に格納する。なお、ストリーム化されたデータは1つだけ指定することができる。

以下に、DTO の実装例を示す。

```
[MessageContract]
public class TransferedFile : IDisposable
{
    /// <summary>
    ///  ファイル名
    /// </summary>
    [MessageHeader]
    public string FileName { get; set; }

    /// <summary>
    ///  ファイルのサイズ
    /// </summary>
    [MessageHeader]
    public long Length { get; set; }

    /// <summary>
    ///  ファイルのストリーミングデータ
    /// </summary>
    [MessageBodyMember]
    public System.IO.Stream FileData { get; set; }

    /// <summary>
    /// Dispose時にファイルをクローズ
    /// </summary>
    public void Dispose()
    {
        if (FileData != null)
        {
            FileData.Close();
            FileData = null;
        }
    }
}
```

リスト 12 サーバ側 DTO(MessageContract)の実装例

ストリーミング転送モードを有効化するには、Web 構成ファイル(Web.config)で、WCF サービスのエンドポイントに対するバインディングの設定を変更する必要があり、binding 要素の transferMode 属性の値を”Sreamed”(または StreamedRequest、StreamedResponse)に変更する。
その他、必要に応じて、maxReceivedMessageSize 属性や maxBufferSize 属性の値も変更し受信可能なデータサイズを調整すること。
なお、ストリーミング転送モードは、BasicHttpBinding、NetTcpBinding、NetNamedPipeBinding のみで利用可能であり、WsHttpBinding では使用できない。
以下に、BasicHttpBinding を使った、Web.config の設定例を示す。

```
<system.serviceModel>
  <bindings>
    <!-- MTOMでのストリーミング転送モード -->
    <basicHttpBinding>
      <binding name="MtomStreamedBinding" messageEncoding="Mtom" transferMode="Streamed"
               … maxBufferSize="65536" … maxReceivedMessageSize="10000000"/>
    </basicHttpBinding>
  </bindings>
…
</system.serviceModel>
```

リスト 13 Web.config の設定例

WCF サービスの実装が完了したら、クライアント AP 側で、「サービス参照の追加」を実施する。
なお、クライアントのアプリケーション構成ファイルも、binding 要素の transferMode 属性の値を”Streamed”(または StreamedRequest、StreamedResponse)に変更し、ストリーミング転送モードを有効化する。また、その他、必要に応じて、maxReceivedMessageSize 属性や maxBufferSize 属性の値も変更し受信可能なデータサイズを調整すること。

(2) サーバ AP が Java(JAX-WS Web サービス)の場合

【ファイルデータをバイト配列で扱う場合】
サーバ側 DTO に byte[]型の要素を定義する。
以下に、DTO の実装例を示す。

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "FileObjectDto")
public class FileObjectDto {
	@XmlElement(name = "FileName")
	private String fileName;
	@XmlElement(name = "FileData")
	private byte[] fileData;

	public String getFileName() {
		return fileName;
	}

	public void setFileName(String fileName) {
		this.fileName = fileName;
	}

	public byte[] getFileData() {
		return fileData;
	}

	public void setFileData(byte[] fileData) {
		this.fileData = fileData;
	}

}
```

リスト 14 サーバ側 DTO の実装例

次に、上記で作成した DTO を引数または戻り値とするメソッドを定義した Web サービスを作成する。この時、Web サービスメソッドには、@MTOM アノテーションを付与する。
以下に、Web サービスの実装例を示す。

```
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.soap.MTOM;

@MTOM
@WebService(serviceName = "TourMtomService")
public class TourMtomService {

    @WebMethod(operationName = "Upload")
    public void upload(FileObjectDto inputDto) {
        String fileName = inputDto.getFileName();
        byte[] fileData = inputDto.getFileData();
    }

    @WebMethod(operationName = "Download")
    public FileObjectDto download(SampleInputDto inputDto){

        // ビジネスロジックの結果をもとにファイルデータを設定
        FileObjectDto outputDto = new FileObjectDto();
        ByteArrayOutputStream bos =  ・・・
        outputDto.setFileData(bos.toByteArray());
        outputDto.setFileName("出力結果.txt");
        return outputDto;
    }
}
```

リスト 15　JAX-WS Web サービスの実装例

JAX-WS Web サービスの実装が完了したら、クライアント AP 側で、「サービス参照の追加」を実施する。
なお、クライアントのアプリケーション構成ファイルも、binding 要素の messageEncoding 属性を”MTOM”に設定し MTOM を有効化する。(「サービス参照の追加」で自動生成されたものは、通常、”MTOM”になっている。)

【ファイルデータをストリームで扱う場合】

また、「サービス参照の追加」により、WCF クライアント側の DTO を Stream 型で生成させるためには、JAX-WS サーバ側では、以下のような DTO を作成する。

- @XmlType アノテーションの namespace 属性に、以下の名前空間を設定する。
  - http://schemas.microsoft.com/Message
- javax.activation.DataHanlder 型のストリームデータの要素を1つだけ定義する。
  - このとき、クラス内に別の要素を追加定義してしまうと、「サービス参照の追加」により WCF クライアント側の DTO は byte[]で生成されてしまうので注意が必要である。
- ストリームデータの要素に@XmlValue アノテーションを付与する。
- ストリームデータの要素に@XmlMimeType アノテーションを付与し、”application/octet-stream”を指定する。

以下に DTO の実装例を示す。

```
import javax.activation.DataHandler;
import javax.xml.bind.annotation.XmlMimeType;
import javax.xml.bind.annotation.XmlType;
import javax.xml.bind.annotation.XmlValue;

//namespaceに、"http://schemas.microsoft.com/Message"を設定する
@XmlType(name = "StreamBody", namespace = "http://schemas.microsoft.com/Message")
public class StreamBody {
	private DataHandler stream;

	//application/octet-streamを設定
	@XmlValue
	@XmlMimeType("application/octet-stream")
	public DataHandler getStream() {
		return stream;
	}
	public void setStream(DataHandler stream) {
		this.stream = stream;
	}
}
```

リスト 16 サーバ側 DTO の実装例

上記のようなストリームを扱う DTO を Web サービスメソッドの引数または戻り値として定義することで、生成される WCF クライアントでは、Stream 型で取り扱うことができる。

以下に、JAX-WS Web サービスの実装例を示す。

このとき、Web サービスメソッドの引数や戻り値に Stream 型以外のパラメータを追加して渡すと、「サービス参照の追加」時に、WCF クライアント側の DTO は、byte[]で生成されてしまうので注意が必要である。

```
import javax.activation.DataHandler;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.soap.MTOM;

@MTOM
@WebService(serviceName = "TourMtomService")
public class TourMtomService {

    @WebMethod(operationName = "Download")
    public StreamBody download(DownloadInputDto inputDto) {
        // ビジネスロジックの結果をもとにDataHanlderを取得し、Streamデータの作成
        DataHandler dataHandler = ・・・
        StreamBody streamBody = new StreamBody();
        streamBody.setStream(dataHandler);
        return streamBody;
    }

    @WebMethod(operationName = "Upload")
    public void upload(StreamBody streamBody) {
        // DataHanlderを取得しビジネスロジック処理
        DataHandler dataHanlder = streamBody.getStream();
        ・・・
    }
}
```

リスト 17　JAX-WS Web サービスの実装例

JAX-WS Web サービスの実装が完了したら、クライアント AP 側で、「サービス参照の追加」を実施すると、WCF クライアント側の DTO が Stream 型で生成される。
なお、クライアントのアプリケーション構成ファイルも、binding 要素の messageEncoding 属性を”MTOM”に設定し MTOM を有効化する。(「サービス参照の追加」で自動生成されたものは、通常、”MTOM”になっている。)
また、binding 要素の transferMode 属性の値を”Streamed”（または StreamedRequest、StreamedResponse）に変更し、ストリーミング転送モードを有効化する。

- .NET-Java 接続におけるストリームを使ったサーバ通信の制約

.NET クライアント-Java サーバ接続においてストリームを使ったサーバ通信において制約があり、「サービス参照の追加」により、WCF クライアント側の DTO を Stream 型のパラメータで生成させるためには、JAX-WS サーバ側の DTO をストリームデータとともに追加情報を同時に送るように定義することができない。
このため、例えば、ファイルダウンロード処理を実装する際、全体ファイルサイズやファイル名を同時に送信できないため、受信したバイト数と全体ファイルサイズを使用して進捗率を計算したり、クライアントへファイル名を送信したりするといったことができない。ストリームデータとは別のリクエストで送受信するか、同時に送受信可能な別の通信基盤を利用する、といった何らかの回避策を検討する必要がある。

■ TIPS

以下では、本機能を利用する際、開発者がよく直面する問題について対処方法をまとめている。開発時に実装方法に困った場合に参考にするとよい。

◆ TERASOLUNA Server Framework For Java(Rich 版)との接続

TERASOLUNA Server Framework for Java (Rich 版)は、従来、プレゼンテーション層のアーキテクチャとして Spring MVC をベースとしたクライアントとの通信機能を提供しており、本フレームワークの前バージョンである TERASOLUNA Framework for .NET 2.x と接続可能であった。

一方、TERASOLUNA Framework for .NET 3.x との接続では、従来の Spring MVC ベースの通信機能は使用せず、JAX-WS Web サービスを利用する[5]。また、TERASOLUNA Server Framework for Java が利用している Spring Framework と連携するため、Web サービスクラスは、org.springframework.web.context.support.SpringBeanAutowiringSupport クラスを継承して作成する。SpringBeanAutowiringSupport クラスを継承することで、@Autowired アノテーションを付与したフィールドやプロパティに対して、Spring Framework の DI/AOP コンテナが管理するインスタンスを注入することができる。

TERASOLUNA Server Framework for Java を使用したサーバ AP を実装する場合、@Autowired アノテーションを付与したビジネスロジッククラスを表すフィールドを定義し、Web サービスクラスから呼び出すことで、ビジネスロジック以降のレイヤは従来の TERASOLUNA Server Framework for Java と同様のアーキテクチャを利用することができる。

以下に、実装イメージを示す。

```
import javax.jws.WebMethod;
import javax.jws.WebService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.web.context.support.SpringBeanAutowiringSupport;
・・・
@WebService(serviceName = "CalcService")
public class CalcService extends SpringBeanAutowiringSupport {
    // Spring Frameworkが管理するインスタンスを取得
    @Autowired
    private CalcBizLogic bizLogic;

    @WebMethod(operationName = "Add")
    public CalcOutputDto add(CalcInputDto param)  throws SampleSoapFaultException {
        ・・・
            // ビジネスロジック実行
            CalcOutputDto result = bizLogic.Add(param);
        ・・・
    }
}
```

リスト 18 TERASOLUNA Server Framework For Java を使った JAX-WS Web サービスの例

[5] TERASOLUNA Server Framework for Java の SOAP 対応版は、2010 年6月末にリリース予定

◆ .NET-Java 接続における DTO のクラス構造の階層ずれの対処

Java サーバ側で作成した DTO クラスと、WSDL より自動生成された.NET クライアント側の DTO のクラス構造の階層が一致しない場合がある。
このような場合、Java サーバ側で作成した DTO クラスと同じ構造を想定して、画面データクラスのデータの階層構造を設計していると、「CL-03 イベント処理実行機能」により、画面データと DTO 間でのデータコピーに失敗しエラーが発生してしまうため、階層ずれを回避するように実装する必要がある。
階層ずれを起こすケースとして、Java サーバ側で配列や List などのコレクションデータを1つだけ要素として持つ DTO を定義する場合が挙げられる。この場合、.NET クライアント側で自動生成される DTO は、List クラスそのものになってしまい、Java 側の DTO と1階層ずれた関係になってしまう。
このような場合、以下のいずれかの対処が必要である。

- Java サーバ側で、コレクションデータをラップするクラスを定義して、サーバ側 DTO はこれを要素とするように定義する(階層を1つ増やす)。
- Java サーバ側で、コレクションデータの他に、ダミーのデータ要素を1つ追加して、サーバ側 DTO に複数の要素が存在するようにする。
- .NET クライアント側で、イベント処理実行機能による、WCF クライアント自動呼び出しの機構を使わず、WCF による Web サービス呼び出しを伴うクライアントビジネスロジックとして実装する。この時、クライアントビジネスロジックの入力 DTO(または出力 DTO)としてコレクションデータをラップするクラスを実装することで、画面データと DTO のデータ構造(階層)を同じにし、データコピーが正常に処理できるようにする。

以下では、Java サーバ側で、コレクションデータをラップするクラスを定義する例を示す。

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "A01_02_01_S01OutputDto")
public class A01_02_01_S01OutputDto {

    //コレクションデータをラップするクラスでフィールド・プロパティを定義
    @XmlElement(name = "TourList")
    private A01_02_01_S01TourCollection tourList;

    public A01_02_01_S01TourCollection getTourList() {
        return tourList;
    }

    public void setTourList(A01_02_01_S01TourCollection tourList) {
        this.tourList = tourList;
    }
}
```

リスト 19 DTO の実装例

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "A01_02_01_S01TourCollection")
public class A01_02_01_S01TourCollection {
    @XmlElement(name = "Tour")
    private List<A01_02_01_S01TourDto> tourList;

    public A01_02_01_S01TourCollection() {
    }

    public A01_02_01_S01TourCollection(List<A01_02_01_S01TourDto> tourList) {
        this.tourList = tourList;
    }

    public List<A01_02_01_S01TourDto> getTourList() {
        return tourList;
    }

    public void setTourList(List<A01_02_01_S01TourDto> tourList) {
        this.tourList = tourList;
    }
}
```

リスト 20 コレクションデータのラッパークラスの実装例

◆ Web サービスメソッドの入出力 DTO がプリミティブ型の場合の対処

「public int Add(int num1, int num2)」といったメソッドのように、Web サービスのメソッドの入出力 DTO がプリミティブ型の場合、「CL-03 イベント処理実行機能」を使用して Web サービスを呼び出す際に、画面データクラスと DTO 間でデータ構造の階層がずれてしまうため、コピー処理が、正常に処理できなくなる。

このような場合、以下のいずれかの対処が必要である。

- サーバ側で、プリミティブ型データをラップするクラスを DTO とするように実装する。
- クライアント側で、WCF クライアントを直接呼び出す代わりに、Web サービス呼び出しを伴うクライアントビジネスロジックとして実装する。この際、クライアントビジネスロジックの入力 DTO(または出力 DTO)としてプリミティブ型データをラップするクラスを実装することで、画面データと DTO 間のデータコピーが正常に処理できるようにする。

◆ クライアント DTO で生成される「*Specified Flags」[6]の対処

.NET-Java 接続の場合、Java 側の Web サービスメソッドの DTO に、Integer 型、Date 型の要素が含まれていると、WSDL より生成された.NET 側の DTO の要素として、対象プロパティ名に、接尾語として「Specified」が付与された bool 型のプロパティが併せて生成される。
これは「*Specified Flags」と呼ばれるもので、.NET の値型（ ValueType ）における null 表現である。例えば Java 側の Integer 型が null 値の代入を許容する型であるのに対して、対応する.NET 側の int(Int32)型は値型であるため null 値を代入できない。このため、.NET 側で対象プロパティの「*Specified Flags」を false に設定することで、Java 側の Integer 型への null 値代入処理を表現できるようになっている。

```
public DateTime DateTimeValue {get; set;}
public bool DateTimeValueSpecified {get; set;}
```

リスト 21　*Specified Flags の例

ここで、「*Specified Flags」が生成された DTO の対象プロパティでは、「*Specified Flags」を true に設定しないと、サーバ側にデータが送信されなくなる（null 設定される）。「*Specified Flags」はデフォルトでは false に設定されており、そのままデータを送信すると、上述したとおり、サーバ側の対象プロパティには null 値が設定されてしまうので注意が必要である。
つまり、対象プロパティに値が入力されている場合には「*Specified Flags」を true に、対象プロパティの値が未入力の場合は、「*Specified Flags」を false にそれぞれ設定する必要がある。
そこで、TERASOLUNA フレームワークでは、対象プロパティに「*Specified Flags」が存在する場合において、値が入力されている場合にはフラグを true に、未入力の場合にはフラグを false に自動的に設定する機構として、「CM-04 データコピー機能」によりデータコピーを実施する際に「*Specified Flags」を自動的に設定する機能を提供している。
これにより、開発者による「*Specified Flags」の設定は不要となり、設定漏れによるサーバへのデータ送信漏れを防止することができる。

---

[6] Schema Files
http://msdn.microsoft.com/en-us/library/cc500289.aspx

◆ ファイルアップロード・ダウンロード時の進捗状況の表示

ファイルの送受信状況について進捗率を計算し、プログレスバーで表示したい場合は、「CL-03 イベント処理実行機能」を利用して、ビジネスロジックで進捗率を計算しイベントを通知することができる。なお、開発者が進捗率を計算する場合は、EventProcessWorker.ProgressSetName プロパティは”Nothing”(または未指定)にすること。
ビジネスロジックから進捗率を通知するには、
「 InvacationScope.Current.GetContext<EvetProcessContext>().ProgressManager 」を実行して、進捗状況を管理する IEventProcProgressManager オブジェクトを取得する。
次に、IEventProcProgressManager.SetProgress メソッドを使用して、進捗率を通知する。
この時、「CL-03 イベント処理実行機能」で、EventProcessWorker の EventProcessName プロパティを「DialogLock」で指定すると、通知した進捗状況を簡単に表示することができる。
また、プロジェクトの要件に応じた進捗状況の表示も可能である。

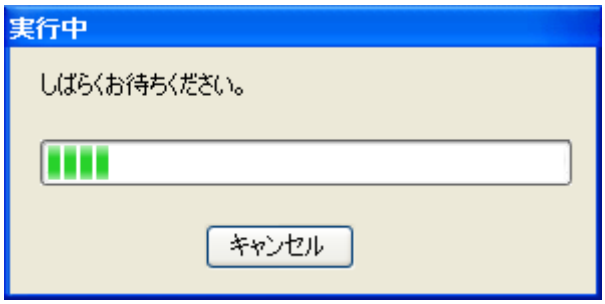


図 9 「CL-03 イベント処理実行機能」の標準機能の進捗ダイアログ

ファイルアップロード・ダウンロードにおける進捗率は、全体ファイルサイズに対する送信(または受信)データのサイズの割合で計算することが多い。
以下に、ファイルアップロード・ダウンロードにおける進捗率を計算し IEventProcProgressManager オブジェクトへ通知するビジネスロジックの実装例を示す。

(1) ファイルアップロードの場合

送信データのストリームデータをラップする Stream 継承クラスを実装して、Stream.Read メソッドをオーバライドして、全体ファイルサイズに対する送信データサイズの割合を計算する。
以下に、進捗率を計算し IEventProcProgressManager オブジェクトへ通知する Stream 継承クラスの実装例を示す。

```
public class NotifyProgressInfoFileStream : Stream
{
    /// ファイル
    private readonly FileStream fileStream;
    /// 進捗率管理クラス
    private IEventProcProgressManager progressManager;
    //Disposeされたか
    private bool disposed;

    public NotifyProgressInfoFileStream(FileStream fileStream)
    {
        this.fileStream = fileStream;
        disposed = false;
        //進捗情報管理機能の取得
        this.progressManager =
                InvocationScope.Current.GetContext<EventProcessContext>().ProgressManager;
    }

    public override bool CanRead
    {
        get { return fileStream.CanRead; }
    }

    public override bool CanSeek
    {
        get { return false; }
    }

    public override bool CanWrite
    {
        get { return false; }
    }

    public override void Flush() {
        fileStream.Flush();
    }

    public override long Length
    {
        get {return fileStream.Length;}
    }

    public override long Position
    {
        get { return fileStream.Position; }
        set { throw new NotSupportedException(); }
    }
・・・
```

```
・・・
        public override int Read(byte[] buffer, int offset, int count)
        {
            if (disposed)
            {
                throw new ObjectDisposedException("すでにファイルはDisposeされています");
            }
            int result = fileStream.Read(buffer, offset, count);
            int percentage = (int)(Position * 100 / Length);
            if (progressManager != null)
            {
                ///進捗状況の通知
                ///送信率を0%～100%で通知する
                progressManager.SetProgress(new ProgressInfo(percentage));
            }
            return result;
        }

        public override long Seek(long offset, SeekOrigin origin)
        {
            throw new NotSupportedException();
        }

        public override void SetLength(long value)
        {
           throw new NotSupportedException();
        }

        public override void Write(byte[] buffer, int offset, int count)
        {
            throw new NotSupportedException();
        }

        /// Disposeパターン
        protected override void Dispose(bool disposing)
        {
            if (!disposed)
            {
                try
                {
                    if (disposing)
                    {
                        fileStream.Dispose();
                    }
                }
                finally
                {
                    base.Dispose(disposing);
                }
                disposed = true;
            }
        }
    }
```

リスト 22 アップロード時の進捗率を計算する Stream 継承クラスの実装例

上記の Stream 継承クラスを使って、ファイルアップロードするクライアントビジネスロジックの実装例を示す。WCF クライアントのサービスメソッドの引数として上記 Stream 継承クラスを渡すことで、Stream.Read メソッドが呼ばれる度に進捗率が計算される。

```
    public class FileUploadBizLogic
    {
        /// エンドポイント名
        private const string EndpointName = "BasicHttpBinding_IMtomService";

        [Dependency]
        public IUnityContainer Container { get; set; }

        public void Upload(FileInfoDto inputDto)
        {
            ///エンドポイントアドレスの取得
            EndpointAddress address =
CommunicationManager.GetReplacedEndpointAddress(Container, EndpointName);
            MtomServiceClient client = new MtomServiceClient(EndpointName, address);

            ///ファイル送信率を通知するStream
            using (NotifyProgressInfoFileStream fs =
                new NotifyProgressInfoFileStream(
                        new FileStream(inputDto.FileName, FileMode.Open)))
            {
                try
                {
                    //ファイルアップロード
                    client.Upload(Path.GetFileName(inputDto.FileName), fs.Length, fs);
                    client.Close();
                }
                catch (System.Exception)
                {
                    if (client != null)
                    {
                        client.Abort();
                    }
                    throw;
                }
            }
        }
    }
```

リスト 23 ファイルアップロードを実施するクライアントビジネスロジックの例

(2) ファイルダウンロードの場合

サーバよりファイルデータ以外に全体ファイルサイズを送信する必要がある。全体ファイルサイズは、リスト 12 ~~リスト 12~~ で示したように SOAP ヘッダに格納する。クライアントビジネスロジックで、Stream 型のデータを Read する際、全体ファイルサイズに対する受信データサイズの割合を計算し、IEventProcProgressManager オブジェクトへ通知する。

以下に、ファイルダウンロードを実施するビジネスロジッククラスの実装例を示す。

```
    public class FileDownloadBizLogic
    {
        /// エンドポイント名
        private const string EndpointName = "BasicHttpBinding_IMtomService";
        private const int BUFFER_SIZE = 1024;
        [Dependency]
        public IUnityContainer Container { get; set; }

        public FileInfoDto Download(FileInfoDto inputDto)
        {
            MtomServiceClient client = null;
            string tmpFileName = "";
            Stream stream = null;
            try
            {
                ///エンドポイントアドレスの取得
                EndpointAddress address =
                    CommunicationManager.GetReplacedEndpointAddress(Container,
EndpointName);
                client = new MtomServiceClient(EndpointName, address);
                long totalLength = 0;
                ///ファイルのダウンロード
                string fileName = inputDto.FileName;
                totalLength = client.Download(ref fileName, out stream);
                if (stream == null || totalLength == 0)
                {
                    return new FileInfoDto(null);
                }
                ///進捗管理クラスの取得
                IEventProcProgressManager progressManager =
                    InvocationScope.Current.GetContext<EventProcessContext>().ProgressManager;
                ///一時ファイルへ書き込み
                tmpFileName = Path.Combine(Path.GetTempPath(), fileName);
                using (FileStream outputFile = File.Create(tmpFileName))
                {
                    byte[] buf = new byte[BUFFER_SIZE];
                    int len = 0;
                    long readLength = 0;
                    while ((len = stream.Read(buf, 0, buf.Length)) != 0)
                    {
                        outputFile.Write(buf, 0, len);
                        readLength += len;
                        progressManager.SetProgress(new ProgressInfo((int) (readLength * 100 /
totalLength)));
                    }
                }
                client.Close();
                return new FileInfoDto(tmpFileName);
・・・
```

```
・・・
            }
            catch (System.Exception)
            {
                if (client != null)
                {
                    client.Abort();
                }
                ///一時ファイルの削除
                if (!string.IsNullOrEmpty(tmpFileName) && File.Exists(tmpFileName))
                {
                    File.Delete(tmpFileName);
                }
                throw;
            }
            finally
            {
                if (stream != null)
                {
                    stream.Close();
                }
            }
        }
    }
```

リスト 24 ファイルダウンロードを実施するクライアントビジネスロジックの例

◆ WCF のトレース機能の利用

通信周りの問題解析時には、System.Diagnostics のトレース機能を使用して、WCF クライアントの処理の追跡や、SOAP メッセージの確認を実施することができる。
詳細は、MSDN のドキュメント[7]等を参照のこと。

トレースの利用には、アプリケーション構成ファイル(App.config)を設定する。
以下に、WCF のトレースの設定例を示す。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <diagnostics>
      <messageLogging logEntireMessage="true"
                      logMalformedMessages="false"
                      logMessagesAtServiceLevel="true"
                      logMessagesAtTransportLevel="false"
                      maxMessagesToLog="3000"
                      maxSizeOfMessageToLog="200000"/>
    </diagnostics>
  </system.serviceModel>
  <system.diagnostics>
    <!-- WCFトレースの設定 -->
      <source name="System.ServiceModel"
        switchValue="Information, ActivityTracing"
        propagateActivity="true">
        <listeners>
          <add name="WCFTraceLog" />
        </listeners>
      </source>
      <source name="System.ServiceModel.MessageLogging">
        <listeners>
          <add name="WCFTraceLog" />
        </listeners>
      </source>
     </sources>
    <sharedListeners>
      <add name="WCFTraceLog" type="System.Diagnostics.XmlWriterTraceListener" initializeData=
".¥WcfClientTrace.svclog" />
    </sharedListeners>
  </system.diagnostics>
</configuration>
```

リスト 25　App.config における WCF トレースの設定例

[7] トレースとメッセージログ
http://msdn.microsoft.com/ja-jp/library/ms751526.aspx

出力されたログは、「サービストレースビューワ」(SvcTraceViewer.exe)で閲覧可能である。
なお、WCF サーバのログを追加で読み込むことで、クライアント-サーバ間の処理を、シーケンシャルに追跡することができる。

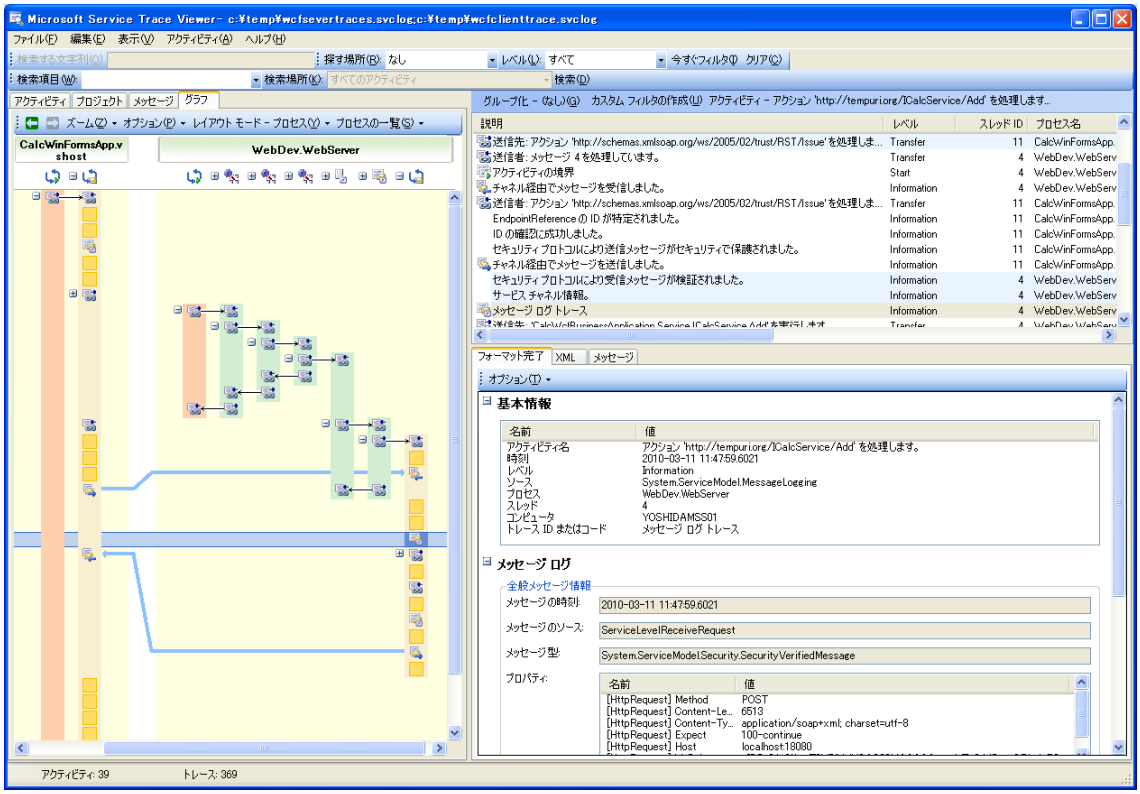


図 10 サービストレースビューワ

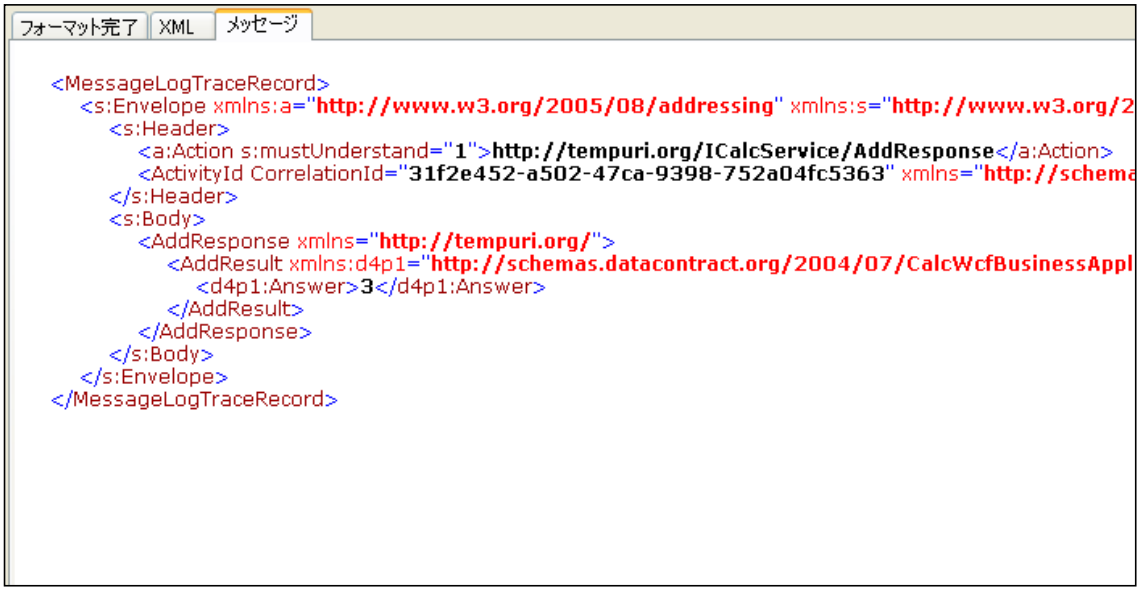


図 11 サービストレースビューワで閲覧した SOAP メッセージ(メッセージログ)の例

■ 内部構成

◆ 構成クラス

本機能を構成するクラスを以下に示す。
「CL-03 イベント処理実行機能」による WCF クライアントの呼び出しなど、他のフレームワーク機能を利用するクラスの記述は省略する。

表 1 構成クラス一覧

項番	クラス名	説明
Terasoluna.ServiceModel 名前空間		
1	CommunicationManager	本機能のエントリクラス。 エンドポイントアドレス情報を管理する。
2	IWcfClientConfigurationManager	WCF クライアントの設定情報を管理するインタフェース。
3	WcfClientConfigurationManager	IWcfClientConfigurationManager のデフォルト実装クラス。
4	IAddressReplaceManager	エンドポイントアドレスの置換処理を管理するインタフェース。
5	AddressReplaceManager	IAddressReplaceManager のデフォルト実装クラス。複数の IAddressReplacer 実装クラスを管理する。
6	CommunicationExtension	本機能を利用するためデフォルトで提供されている UnityContainerExtension 継承クラス。
7	IAddressReplacer	エンドポイントアドレスの置換処理を実施するインタフェース。
8	MappingAddressReplacer	IAddressReplacer のデフォルト実装クラス。
9	WcfUtility	WCF クライアントの呼び出しのためのユーティリティクラス。

■ 拡張ポイント

標準では WCF による通信を前提とした機能を提供している。
その他の通信基盤を使用する場合、クライアントビジネスロジッククラスから対象の通信クラスを呼び出すことで実現可能であるが、EventProcessWorker による自動実行を実現したい場合には、「CL-03 イベント処理実行機能」がもつ「ビジネスロジック実行」のためのコンポーネントを拡張する必要がある。

まず、IBizLogicExecutor インタフェースを実装して、対象の通信基盤を使用してサーバ処理を実施するクラスを実装する。

```
public class HttpClientBizLogicExecutor : IBizLogicExecutor
{
    public const string InstanceName = "Http";

    public object ExecuteBizLogic(BizLogicInfo info, object input)
    {
        //InfoManagerを使用してBizLogicInfoの情報をもとに通信クラスを作成
        object bizLogic = InfoManager.GetBizLogic(info);
        //サーバ処理を呼び出す
        ・・・
    }

    [Dependency(InstanceName)]
    public IBizLogicInfoManager InfoManager { get; set; }

}
```

リスト 26　IBizLogicExecutor インタフェースの実装例

また、BizLogicInfoManagerBase クラス を継承するなどして IBizLogicInfoManager インタフェースを実装し、対象の通信クラス(プロキシクラスなど)を取得する処理を実装する。

```
public class HttpClientBizLogicInfoManager : BizLogicInfoManagerBase
{
    public override object GetBizLogic(BizLogicInfo info)
    {
        ・・・
    }
    public override TBizLogic GetBizLogic<TBizLogic>(string bizLogicName)
    {
        ・・・
    }
    public override string[] GetCandidateBizLogicNames(Type type)
    {
        ・・・
    }
    public override bool IsSupportedBizLogic(Type type, string bizLogicName)
    {
        ・・・
    }
}
```

リスト 27　IBizLogicInfoManager インタフェースの実装イメージ

次に、「ビジネスロジック実行」のためのデフォルトの Extension クラスである ClientBizLogicExtension クラスを継承して、UnityContainerExtension.Initialize メソッドをオーバライドし IBizLogicExecutor インタフェースの実装クラスの DI 設定を追加する。
この時、RegisterType メソッドの引数で、登録名として渡した文字列は、EventProcessWorker.BizLogicExecution プロパティの「ビジネスロジック種別」の表示名に追加される。

```
public class SampleClientBizLogicExtension : ClientBizLogicExtension
{
    protected override void Initialize()
    {
        base.Initialize();
        /// 引数で渡した登録名が
        /// EventProcessWorker.BizLogicExecutionプロパティの
        ///  「ビジネスロジック種別」の表示名になる
        Container.RegisterType<IBizLogicExecutor, HttpClientBizLogicExecutor>(
            HttpClientBizLogicExecutor.InstanceName,
            new ContainerControlledLifetimeManager());
        Container.RegisterType<IBizLogicInfoManager, HttpClientBizLogicInfoManager>(
            HttpClientBizLogicExecutor.InstanceName,
            new ContainerControlledLifetimeManager());
    }
}
```

リスト 28 UnityContainerExtension 継承クラスの実装例

AP 共通構成ファイル（TerasolunaApplicaiton.config）の
/configuration/unity/containers/container/extensions/add タグで
ClientBizLogicExtension クラスの代わりに、新たに作成した Extension クラスを設定する。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <unity>
    <containers>
      <container>
        <extensions>
          <!-- ビジネスロジック実行の設定 -->
          <add type="CalcBusinessApplication.Web.SampleClientBizLogicExtension,
                    CalcBusinessApplication" />
        </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 29 TerasolunaApplication.config の設定例

すると、EventProcessWorker.BizLogicExecution プロパティの設定画面で、先ほど追加した IBizLogicExecutor 実装クラスが「ビジネスロジック種別」に追加され、業務開発者が新しい通信基盤によるサーバ通信を選択できるようになる。

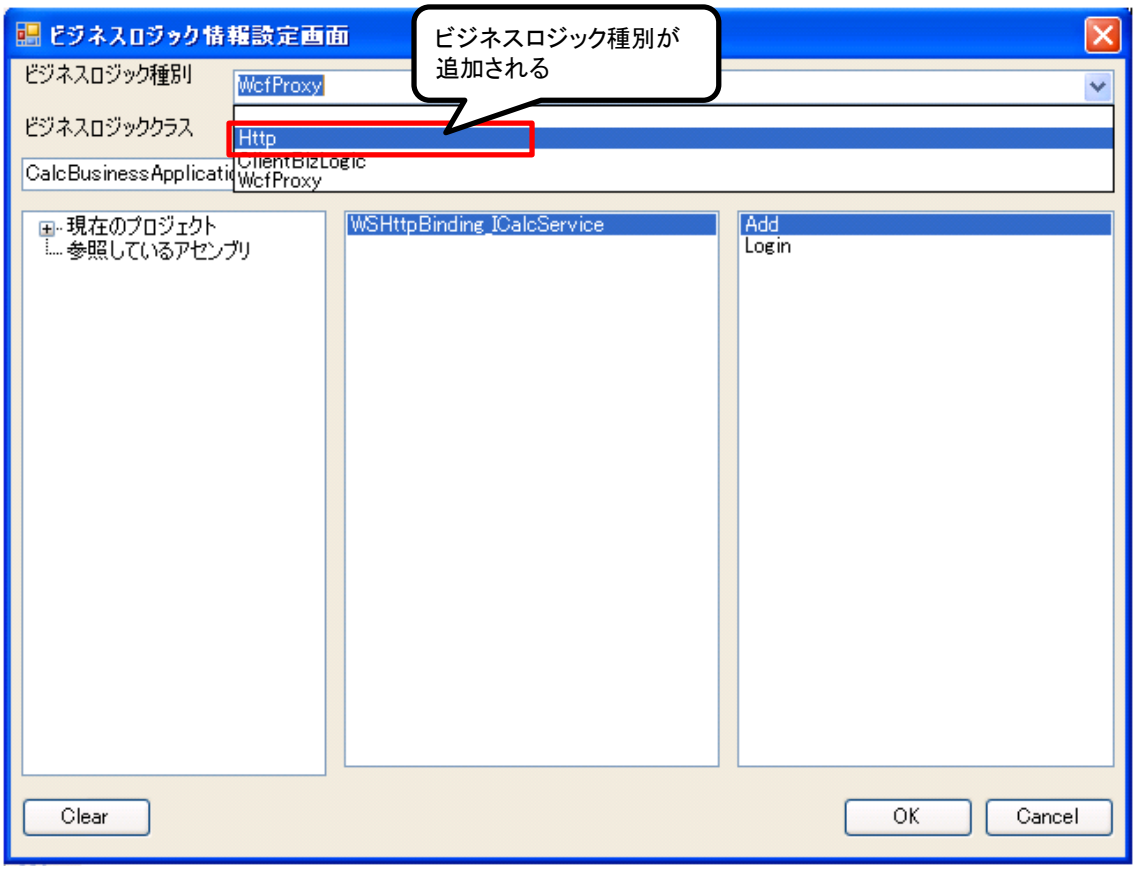


図 12 EventProcessWorker の Communiation プロパティによる通信設定画面

■ 関連機能

- CM-02 インスタンス管理機能
- CL-03 イベント処理実行機能