WebOTX アプリケーション開発ガイド

WebOTX アプリケーション開発ガイド

バージョン: 7.1

版数: 第 10 版

リリース: 2011 年 2 月

Copyright (C) 1998-2011 NEC Corporation. All rights reserved.

目次

2.J2EE共通

本章では、WebOTX Developer の機能を使いこなすための詳細な説明を行います。また、WebOTX が提供する API の利用方法について説明します。

2.1.Webサービス

2.1.1.Webサービス開発のポイント

まずは土台作り

アプリケーションの部品化はシステムの拡張性、柔軟性を高める上で非常に重要です。Web サービスは、その部品をつなぐための道具にすぎません。Web サービスを構築する前に、まずはしっかりと設計し、部品化されたアプリケーションを作ることが大切です。WebOTX は、**Java アプリケーション**、**Web アプリケーション**、**EJB**、**メインフレームに代表されるレガシーなシステム**などの様々なアプリケーションを Web サービス化する手段を提供しています。

Javaアプリケーション・WebアプリケーションのWebサービス化

Java アプリケーションと Web アプリケーションは、形式こそ違いますが、アプリケーションの基礎の部分は **JavaBean** の集合体で作られていることでしょう。JavaBean は、それだけで機能する単純なものでも、JDBC を使ってデータベースにアクセスして何かをするものでも構いません。Web サービス化するのは、そのうちのあるクラスのあるメソッドです。どのメソッドを Web サービス化したいのかということをよく計画してください。

EJBのWebサービス化

EJB を Web サービス化する場合には、まずその EJB が**ステートレスセッション Bean** であるかどうかを確認してください。現在、Web サービスで状態を扱うための標準仕様はなく、事実上ステートレスセッション Bean のみ Web サービス化が可能です。また、JavaBean と同様、どの EJB のどのメソッドを Web サービス化したいのかをよく計画してください。

XMLアプリケーションのWebサービス化

最近は XML をやり取りするシステムを構築するケースが増えてきており、やり取りする XML を SOAP に乗せて Web サービス化することによって XML の利点を活かしたいというニーズが増えています。もちろん、このような XML アプリケーションも Web サービス化することができます。WebOTX では、**SAAJ**、**JAXP**、**StAX といった API** を利用して**サーブレット**によって Web サービス化する方法を提供しています。

部品はWebサービス化する

ビジネスロジックが完成すれば、あとは WebOTX Developer の Web サービス作成ウィザードを利用するなどして Web サービス化を行うのみですので、作業自体はとてもシンプルです。しかし、Web サービス化を行った後、ビジネスロジックに不具合が見つかり置換したい場合があるかもしれません。そういった場合、Web サービス部分に影響を与えずにビジネスロジックのみを置換したいと誰もが思うことでしょう。「**インタフェースが変わらない**」という条件がクリアできれば、ビジネスロジックのみ置換することができます。こうしたことに備えて、ビジネスロジックのインタフェースのうち Web サービス化する部分については特に綿密に設計し、変更のないようにされることをおすすめします。

組み合わせて発展させる

個々の Web サービスは、部品化されたアプリケーションであるべきというポイントを紹介しましたが、これは裏を返せば「1 つの Web サービスだけでは何の意味も持たないこともある」ということが言えます。しかし、Web サービスには「**アグリゲーションサービス**」といって複数の Web サービスを組み合わせるという考え方があります。これこそ、アプリケーション同士がダイナミックに連携する仕掛けなのです。

例えば、今「**C**」という新サービスを立ち上げたいと思っているが、それに相当するアプリケーションは稼動していないとします。しかし、それは「**A 業務アプリケーション**」と「**B 業務アプリケーション**」という 2 つのアプリケーション部品を組み合わせれば実現できるというものであることがよくあります。A 業務アプリケ

ーションと B 業務アプリケーションを Web サービス化すれば、A と B の Web サービスを簡単に組み合わせることができますので、C サービスはすぐに立ち上げることができるのです。さらに、**B を自社で持っていないと**いう場合にも対応することができます。B だけ誰かが提供している Web サービスを利用すればよいのです。WebOTX では、複数のサービスをコーディングによって連結することも可能ですが、オプション製品の **WebOTX Enterprise Service Bus** や **WebOTX Process Conductor** を使えば、自由自在に様々なサービスを呼び出して、思い通りのシステムが作れます。

2.1.2.Webサービス作成ウィザード

WebOTX Developer を使うと、とても簡単に Web サービスを開発することができます。ここでは、WebOTX Developer が提供する Web サービスの開発機能である「Web サービス作成ウィザード」についてと、その関連機能について詳しく説明します。

Webサービス作成ウィザード

Web サービス作成ウィザードを使うと、Java アプリケーション（Web アプリケーションを含む）や EJB を Web サービス化することができます。作成される Web サービスは JAX-RPC 1.1、SAAJ 1.2、WSEE 1.1、WS-I BP 1.0 などを総括する J2EE 1.4 に対応したものになり、完成した WAR/EJB-JAR/EAR のようなアプリケーションアーカイブは WebOTX Application Server のみならず、他の J2EE 1.4 対応のアプリケーションサーバに配備して動作させることもできます。また、ウィザードの後半に集中している Web サービスセキュリティの設定を行うと、Web Services Security 1.0 仕様とそのすべてのプロファイルに対応したセキュアな Web サービスを作ることができます。

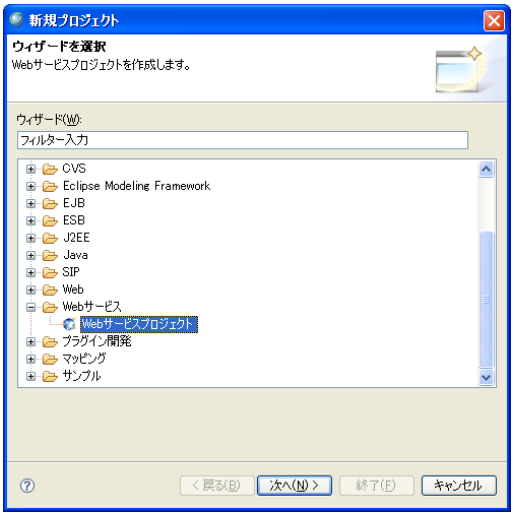
Web サービスセキュリティの機能を使う場合、ウィザードを開始する前に必ず 2.1.10.セキュリティをお読みください。事前設定が必要です。

では、Web サービス作成ウィザードの各画面の設定方法について順に説明します。

新規プロジェクトダイアログ

WebOTX Developer’s Studio のメニューから、**ファイル**｜**新規**｜**プロジェクト**を選択すると、**新規プロジェクト**ダイアログが起動します。

Web サービス作成ウィザードを開始するには、**Web サービス**｜**Web サービスプロジェクト**を選択し、[**次へ**]ボタンを押します。



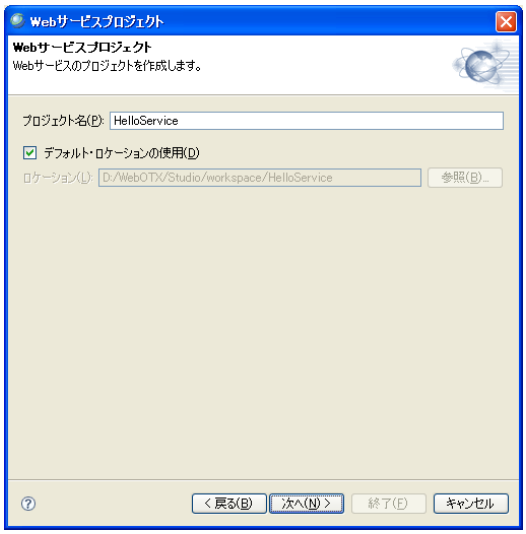
Webサービスプロジェクト

Web サービス作成ウィザードは「**Web サービスプロジェクト**」を作成し、Web サービスは Web サービスプロジェクトに生成されます。**プロジェクト名**に Web サービスプロジェクトの名前を指定します。プロジェクトは、デフォルトでは

<WebOTX_DIR>¥Studio¥workspace¥指定したプロジェクト名

というフォルダに作成されます。もし、別の場所に作成したい場合は、**デフォルトの使用**のチェックをはずし、任意の場所を絶対パスで指定します。

以上の設定を終了したら、[**次へ**]ボタンを押します。



MEMO

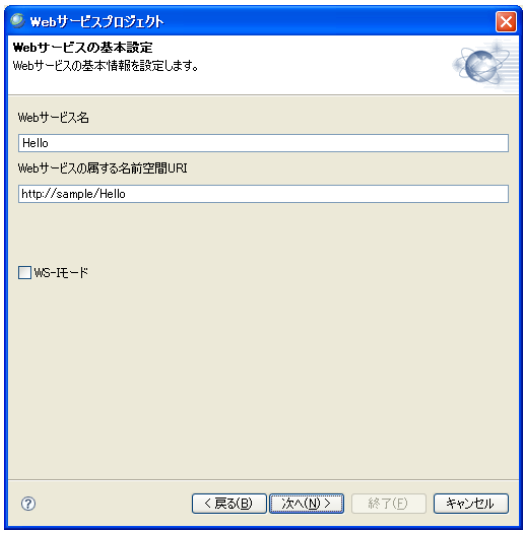
プロジェクト名に含めてはいけない文字を含めると、エラーメッセージが表示されます。

MEMO

<WebOTX_DIR>は WebOTX のインストールルートディレクトリのことです。

Webサービスの基本設定

Web サービスを作成するための最も基本的な情報を指定します。すべて設定が終了したら、[**次へ**]ボタンを押します。



設定項目	設定内容
Web サービス名	Web サービス名を英数字とアンダーバーを使用して指定します。
Web サービスの属する名前空間 URI	Web サービスの存在する名前空間を URI(URL、URN)形式で指定します。http:や urn:などのスキームに続けて任意の名前を指定します。 (入力例) urn:nec-webotx http://www.nec.com/WebOTX
WS-I モード	WS-I BP1.0 に対応したい場合にチェックします。

MEMO

Web サービス名は、Web サービス作成時にクラス名として使用するため、Java の識別子の規則に従っていないとエラーメッセージが表示されます。

MEMO

名前空間は、Web サービスが存在する空間の名前のことです。Java のパッケージ名に相当します。

MEMO

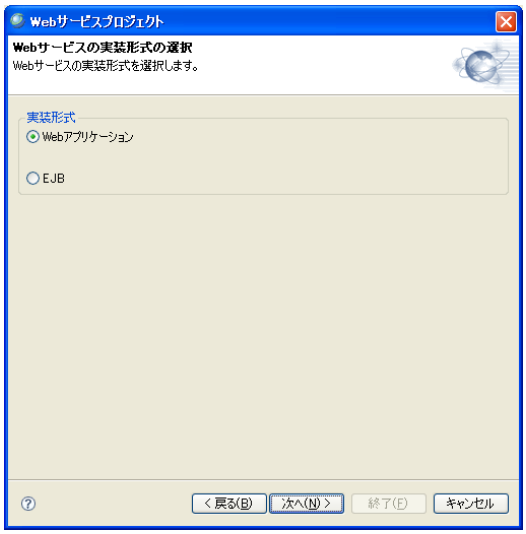
URI は、ウェブ上でリソースを識別する手段で、スキームに続けて名前を指定する形式です。http:、mailto:、urn:などのスキームはよく知られたところでしょう。詳細は、「URI」でウェブ検索してください。

WS-I とは

WS-I とは Web Services Interoperability Organization の略称で、Web サービスの相互接続性を向上させるためにはどうしたらよいかということを様々な角度から検証している団体です。W3C や OASIS などで策定された Web サービスの基本仕様を元にしたガイドライン、サンプルアプリケーション、テストツールなどを提供しています。そのうちの 1 つに、SOAP、WSDL、UDDI のレベルで Web サービスアプリケーションをどのように作れば相互接続性を向上できるかということについて示されたガイドラインとして Basic Profile があり、そのバージョン 1.0 が WS-I BP 1.0 です。WebOTX は WS-I BP 1.0 に対応しており、開発時に WS-I モードを指定することで、Web サービスを WS-I に対応させることができます。

Webサービスの実装形式選択

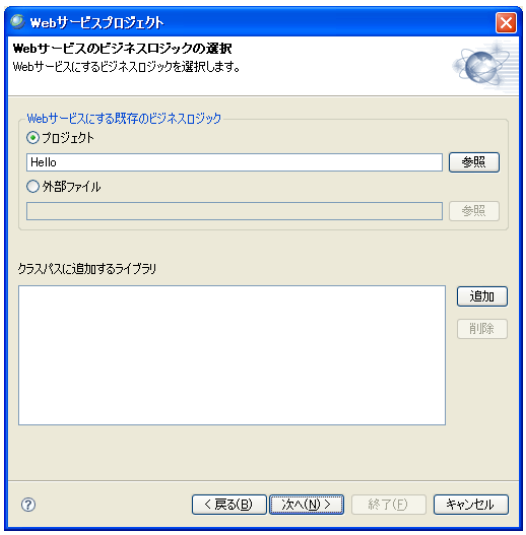
Web サービスを実装する形式を選択します。選択したら、[**次へ**]ボタンを押します。



Webサービスのビジネスロジックの選択

Web サービスにする既存のビジネスロジックに、Web サービス化したいビジネスロジックを含むプロジェクト、またはアプリケーションのアーカイブを指定します。Web サービスの実装形式選択画面で Web アプリケーションを選択した場合は **Java プロジェクト**、**WebOTX Web プロジェクト**、**JAR ファイル**、**WAR ファイルのうちいずれか**を、EJB を選択した場合は **WebOTX EJB プロジェクト**、**EJB-JAR ファイルのうちいずれか**を指定します。

ビジネスロジックが参照しているライブラリがある場合、クラスパスに追加するライブラリの[**追加**]ボタンを押し、ライブラリを指定します。追加できるファイルは、**JAR ファイル**、もしくは **ZIP ファイル**です。追加を取り消したい場合は、そのライブラリをリストで選択し、[**削除**]ボタンを押します。



Web サービスの実装形式	指定するビジネスロジックの形式
Web アプリケーション	・Java プロジェクト ・JAR ファイル ・WebOTX Web プロジェクト ・WAR ファイル ・WTP Web プロジェクト などの Eclipse の Java プロジェクトに準じた全てのプロジェクト、ならびに Java アプリケーションに相当する全てのアプリケーションアーカイブ
EJB	・WebOTX EJB プロジェクト ・EJB-JAR ファイル ・WTP EJB プロジェクト

ビジネスロジックとして指定するプロジェクトは、Eclipse のビルド機能によりビルドが成功しているものでなければなりません。

ビジネスロジックとして指定するプロジェクトが参照するプロジェクトも、すべて Eclipse のビルド機能によりビルドが成功しているものでなければなりません。

MEMO

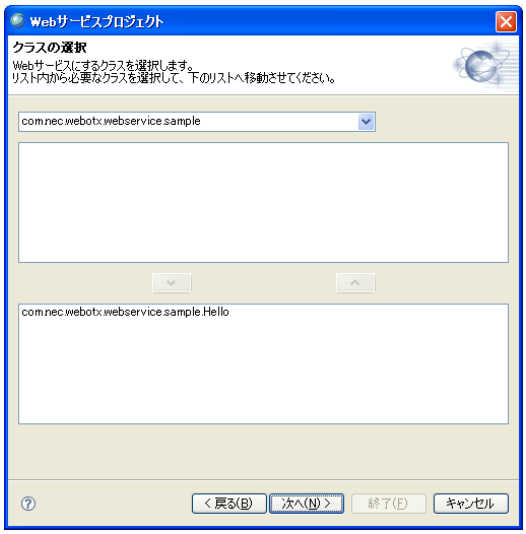
Ant でビルドを行っており、Eclipse のビルド機能ではビルドに失敗するプロジェクトを指定したい場合は、一旦 Ant によるビルドでアーカイブを作成すれば、そのアーカイブを指定して Web サービスを作成することができます。

クラスの選択

Web サービスのビジネスロジックの選択画面で **Java プロジェクト、WebOTX Web プロジェクト、JAR ファイル、WAR ファイルのいずれか**を指定したとき、そのアーカイブに含まれるクラスの中から、Web サービス化したいクラスを選択する画面です。

プルダウンメニューからパッケージ名を選択すると、上のリストにそのパッケージに含まれるクラスの一覧が表示されます。その中から Web サービス化するものを選び、[▼]ボタンを押します。下のリストに追加されれば、選択完了です。

この作業を繰り返して、Web サービス化したいクラスが全て下のリストに表示されるようにしてください。下のリストでクラスを選択して[▲]ボタンを押すと、選択解除することができます。最後に、[**次へ**]ボタンを押します。



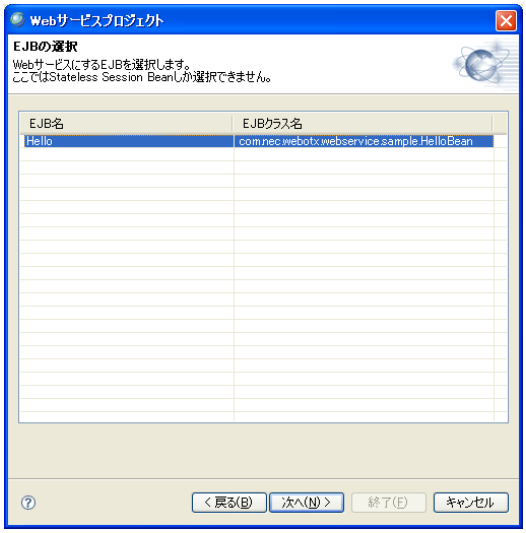
インタフェース、抽象化クラスは選択しないでください。

パッケージ名のついていないクラス（デフォルトパッケージに存在するクラス）は表示されません。

EJBの選択

Web サービスのビジネスロジックの選択画面で **WebOTX EJB プロジェクト、EJB-JAR ファイルのいずれか**を指定したとき、そこに含まれるEJBの中からWebサービス化したいEJBを選択する画面です。

Web サービス化の対象になる EJB の EJB 名と EJB のクラスが一覧表示されます。この中から Web サービス化したい EJB を 1 つ選択し、[**次へ**]ボタンを押してください。



ステートレスセッション Bean 以外の EJB は表示されません。

MEMO

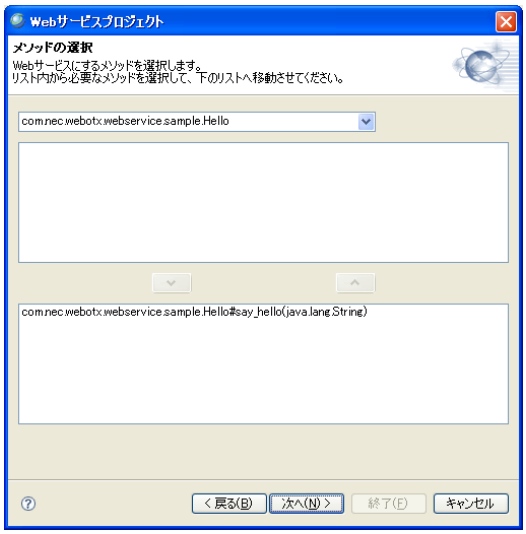
異なるパッケージに存在する複数のクラスを同時に選択することもできます。

メソッドの選択

クラスの選択、またはEJBの選択画面で選択したクラスに含まれるメソッドから、Web サービス化するものを選択する画面です。

プルダウンメニューからクラスを選択すると、上のリストにそのクラスに含まれるメソッドの一覧が表示されます。その中からWeb サービス化するものを選び、[▼]ボタンを押します。下のリストに追加されれば、選択完了です。

この作業を繰り返して、Web サービス化したいメソッドが全て下のリストに表示されるようにしてください。下のリストでメソッドを選択して[▲]ボタンを押すと、選択解除することができます。最後に、[**次へ**]ボタンを押します。



インタフェース、抽象化クラスはプルダウンメニューには表示されません。

1 つのクラス内に同名のメソッドが複数ある場合、それらを同時に複数選択することはできません。

コンストラクタ、小文字で始まっていない名前のメソッドは選択できません。

ビジネスロジックが EJB の場合で、メソッド名に数字がある場合、その直後の文字は大文字でなければなりません。

MEMO

異なるクラスに存在する複数のメソッドを同時に選択することもできます。

Web サービスと型

Web サービスのデータのやり取りは XML を利用します。Java と XML ではデータ型の種類が異なっているため、それぞれの型同士を結び付けて互換性を取る必要があります。そこで、JAX-RPC 仕様では Java の型から XML の型への変換規則、XML の型から Java の型への変換規則を規定しています。WebOTX は JAX-RPC の規定している型変換法則に対応しています。Web サービス作成ウィザードでは、その中で Java の型と XML の型の相互変換をする時に可逆である型の組み合わせを使用しており、実際に利用できる型は次に挙げるものです。ランタイムの種類、ウィザード中で選択するモードによって対応する型に違いがありますので、ビジネスロジックの引数と返り値の型が対応しているかどうかをよくご確認ください。引数や返り値に任意に作成した Bean を使用している場合、そのフィールドに含まれる変数の型についても同様に考慮する必要がありますのでご注意ください。

Java の型	WS-I mode OFF RPC/encoded	WS-I mode ON WS-I mode OFF RPC/literal WS-I mode OFF Document/literal
boolean	xsd:boolean	xsd:boolean
★byte	xsd:byte	xsd:byte
★byte[]	xsd:base64Binary	xsd:base64Binary
short	xsd:short	xsd:short
int	xsd:int	xsd:int
long	xsd:long	xsd:long
float	xsd:float	xsd:float
double	xsd:double	xsd:double
java.lang.String	xsd:string	xsd:string
java.math.BigInteger	xsd:integer	xsd:integer
java.math.BigDecimal	xsd:decimal	xsd:decimal
java.util.Calendar	xsd:dateTime	xsd:dateTime
javax.xml.namespace.QName	xsd:QName	xsd:QName
java.net.URI	xsd:anyURI	xsd:anyURI
java.lang.Boolean	soapenc:boolean	xsd:boolean
java.lang.Byte	soapenc:byte	xsd:byte
java.lang.Short	soapenc:short	xsd:short
java.lang.Integer	soapenc:int	xsd:int
java.lang.Long	soapenc:long	xsd:long
java.lang.Double	soapenc:double	xsd:double
java.lang.Float	soapenc:float	xsd:float

- ★のついたもの以外は、その型の配列型にも対応します。
- xsd の名前空間 URI は「http://www.w3.org/2001/XMLSchema」です。
- soapenc の名前空間 URI は「http://schemas.xmlsoap.org/soap/encoding/」です。

型に関する注意

詳細設定画面で RPC/literal の設定をしたとき、メソッド選択画面で選択した全てのメソッドの引数、返却値について、次の型の組み合わせについては同時に複数使用することができません。

同時に選択できない型の組み合わせ	
java.lang.Boolean[]	boolean[]
java.lang.Float[]	float[]
java.lang.Double[]	double[]
java.lang.Integer[]	int[]
java.lang.Short[]	short[]

※[]は配列の意味です。

添付ファイル

Web サービスで使用する SOAP メッセージには添付ファイルをつけることができます。WS-I モードが OFF で、RPC/encoded のとき、次の型の引数、または返り値が存在するメソッドが選択されると、自動的に添付ファイルとして扱います。なお、添付ファイルがある状態で生成された **Main クラス**は必ずカスタマイズする必要があります。

- javax.activation.DataHandler
- java.awt.Image
- javax.mail.internet.MimeMultipart
- javax.xml.transform.Source

Web サービスと Web サービスクライアントのどちらか片方でも WebOTX の JAX-RPC 以外のランタイムを使う場合、WS-I モードを ON にする場合、RPC/literal や Document/literal を使用する場合は、SOAP メッセージの添付ファイル領域を使うことはできません。その場合、あらかじめビジネスロジック内で byte 配列型に変換するようにしておき、その byte 配列を Web サービス化するメソッドの引数、または返却値に入れるようにしてください。こうすることによって、ファイルの受け渡しを byte 配列の受け渡しとして扱うことができます。ただし、byte 配列はメモリに展開して扱うため、ヒープ領域を多量に消費します。特に送受信するファイルが大容量の場合は不向きです。byte 配列を用いたときにヒープ領域が圧迫され、性能劣化が予測される環境では、SAAJ API を使用して Web サービスを実装するか、HTTP を直接使ってファイル送受信するようにしてください。

ユーザ定義の例外

Web サービス化するビジネスロジックで java.rmi.RemoteException 以外の例外を定義すると、それはユーザ定義の例外として扱われます。ユーザ定義の例外をスローすると、ビジネスロジック内で例外が起こったときに SOAP Fault メッセージを使って、例外が発生した場所やクラス名、例外の詳細な内容についてクライアントに通知することができます。
ユーザ定義の例外は、クライアントに渡す情報量によって作り方がいくつかあります。
1 つ目は、java.lang.Exception を継承するだけのクラスを作成し、引数なしのインスタンスを作成して、例外をスローします。すると、faultstring に発生した例外のクラス名が入ります。
2 つ目は、文字列のメッセージをクライアントに渡す場合です。このときは、java.lang.Exception を継承するクラスを作成します。ビジネスロジックの中で例外をスローするときには、String の引数が 1 つのコンストラクタでインスタンスを作成し、引数にクライアントに渡したい文字列を入れます。

```
public class UserException extends Exception {
    public UserException(String arg0) {
        super(arg0);
    }
}
```

このとき、detail には次のようにメッセージが封入されます。

```
  <detail>
```

```
    <ns1:UserException xsi:type="ns0:UserException">
      <message xsi:type="xsd:string">例外のメッセージ</message>
    </ns1:UserException>
  </detail>
```

3 つ目は、detail 要素を使ってさらにいろいろな情報を渡したい場合です。このときは、JavaBean の規則に沿って private 変数を宣言し、各変数ごとに setter, getter を用意します。このとき、コンストラクタは定義しないか、デフォルトコンストラクタのみを定義しておきます。

```
public class UserException extends Exception {
    private int code;
    private String message;
    public int getCode() {
        return code;
    }
    public void setCode(int code) {
        this.code = code;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
}
```

この状態で Web サービス作成ウィザードを通すと、コンパイルエラーになるクラスが現れるため、次のようにコンストラクタがあることを前提としたコードから引数を取り出し、setter を使うコードに修正します。(逆にユーザ定義例外側に生成されたのと同じ形式のコンストラクタを追加しても構いません。)

RPC-encoded の場合・・・ユーザ定義の例外クラス名_SOAPBuilder 修正前：

```
    public void construct() {
        _instance = new com.nec.webotx.webservice.sample.UserException(code, message);
    }
```

RPC-encoded の場合・・・ユーザ定義の例外クラス名_SOAPBuilder 修正後：

```
    public void construct() {
        _instance = new com.nec.webotx.webservice.sample.UserException();
        _instance.setCode(code);
        _instance.setMessage(message);
    }
```

RPC-encoded の場合・・・ユーザ定義の例外クラス名_SOAPSerializer 修正前：

```
        if (isComplete) {
            instance = new com.nec.webotx.webservice.sample.UserException(((Integer)codeTemp).intValue(),
(java.lang.String)messageTemp);
        } else {
```

RPC-encoded の場合・・・ユーザ定義の例外クラス名_SOAPSerializer 修正後：

```
        if (isComplete) {
            instance = new com.nec.webotx.webservice.sample.UserException();
            instance.setCode(((Integer)codeTemp).intValue());
            instance.setMessage((java.lang.String)messageTemp);
        } else {
```

RPC/Document-literal の場合・・・ユーザ定義の例外クラス名_LiteralSerializer 修正前：

```
    instance = new com.nec.webotx.webservice.sample.UserException(((Integer)codeTemp).intValue(),
      (java.lang.String)messageTemp);
```

RPC/Document-literal の場合・・・ユーザ定義の例外クラス名_LiteralSerializer 修正後：

```
    instance = new com.nec.webotx.webservice.sample.UserException();
    instance.setCode(((Integer)codeTemp).intValue());
    instance.setMessage((java.lang.String)messageTemp);
```

このとき、detail には次のようにメッセージが封入されます。

```
<detail>
  <ns1:UserException xsi:type="ns0:UserException">
    <code xsi:type="xsd:int">012345</code>
    <message xsi:type="xsd:string">例外のメッセージ</message>
  </ns1:UserException>
</detail>
```

※ユーザ定義の例外クラスが継承関係を持っている場合などに、ユーザ定義の例外クラスと同じクラス名のクラスが Web サービスプロジェクトに生成されてしまう場合がありますが、そのクラスは不要ですので手動で削除してください。

※SOAP Fault の返却が成功した場合、JAX-RPC は正常に処理を終了しているため、webservice.log にログは出力されません。ユーザ定義の例外クラスが呼び出された際のログを残す必要がある場合は、アプリケーション側で処理を追加する必要があります。

※Document/literal の時、void（返却値なし）のメソッドにおいてユーザ定義の例外をスローすることはできません。
Document/literal では、void（返却値なし）のメソッドを Web サービス化するには「一方向通信を行う」オプションをつけなければならず、一方向通信においてユーザ定義の例外をスローすることは、本来返るはずのない SOAP Fault メッセージを定義することにあたるためです。

ユーザ定義の配列型について

JAX-RPC に準拠する Web サービスプロジェクトを作成する際、パラメータでデータを入出力するには Holder クラスを使用する必要があります。
Web サービスプロジェクトを作成する際にユーザ定義の Object を保持する Holder クラスをパラメータに持つメソッドを Web サービス化すると、 自動生成されるソースコード内でコンパイルエラーが発生する場合があります。その場合、手動で次のような修正が必要になります。

例:
Web サービスのプロジェクト名:Web
ビジネスロジックのプロジェクト名:LogicBean
ユーザ定義の Object を保持する Holder クラス:test.TestDataHolder
自動生成された Holder クラス:webotx.ws.ds.WebService.holders.TestDataHolder

WebServiceClient.java:
自動生成されたメソッド(getData_LogicBean)で自動生成されたクラス(webotx.ws.ds.WebService.holders.TestDataHolder)にユーザ定義された Holder クラス(test.TestDataHolder)を格納する必要があります。

・修正前

```
public int getData_LogicBean(java.lang.String in0, test.TestDataHolder in1) throws java.rmi.RemoteException {
    return binding.getData_LogicBean(in0, in1);
}
```

・修正後

```
import webotx.ws.ds.WebService.holders.TestDataHolder;
    public int getData_LogicBean(java.lang.String in0, test.TestDataHolder in1) throws java.rmi.RemoteException {
        TestDataHolder td = new TestDataHolder(in1.getValue());
        return binding.getData_LogicBean(in0,td);
    }
```

WebServiceSoapBindingImpl.java:
WebServiceClient.java とは逆のことを行ってください。

・修正前

```
public int getData_LogicBean(javax.xml.rpc.holders.ByteArrayHolder in0, test.TestDataHolder in1)
        throws java.rmi.RemoteException {
    try {
        return new test.TestDelegate().getData(in0, in1);
    } catch (Throwable e) {
        throw new java.rmi.RemoteException("", e);
    }
}
```

・修正後

```
public int getData_LogicBean(java.lang.String in0, webotx.ws.ds.WebService.holders.TestDataHolder in1) throws
java.rmi.RemoteException {
    try {
        test.TestDataHolder td = new test.TestDataHolder(in1.value);
        return new test.TestDelegate().getData(in0, td);
    } catch (Throwable e) {
        throw new java.rmi.RemoteException("", e);
    }
}
```

パッケージ名変更の設定

自動生成される Java クラスのパッケージ名を変更することができます。変更方法は次の通りです。

<workspace>/.metadata/.plugins/org.eclipse.core.runtime/.settings にある **com.nec.webotx.webservice.prefs** ファイルに

common.package=新パッケージ名＋「.」を追加します。

（例）common.package=homo.moge.

※**com.nec.webotx.webservice.prefs** ファイルがなければ、手動で同じ名前のテキストファイルを新規作成する必要があります。

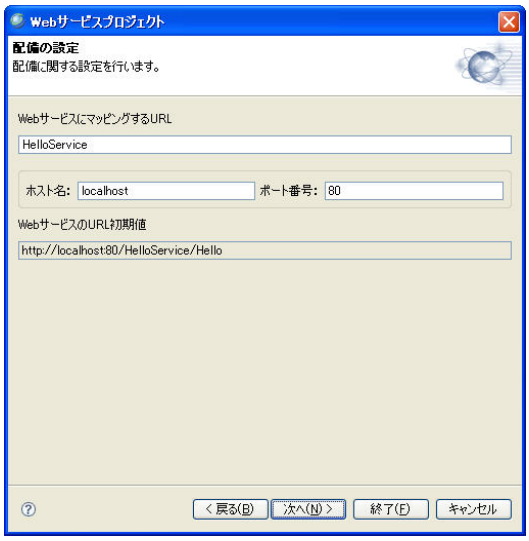
デフォルトのパッケージ名は **webotx.ws.ds.「Web サービス名」**です。

配備の設定

Web サービスを配備する URL を指定する画面です。Web サービスの URL は、

http://ホスト名:ポート番号/Web サービスにマッピングする URL/Web サービス名

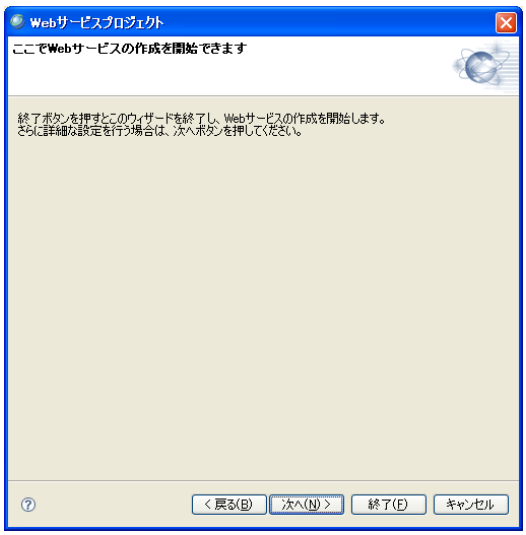
となります。この画面で指定した値が上記の URL の **Web サービスにマッピングする URL、ホスト名とポート番号**部分に反映されます。Web サービスにマッピングする URL の初期値はプロジェクト名が指定されています。ホスト名の初期値は「localhost」、ポート番号の初期値は「80」が指定されています。変更したい場合には書き換えてください。変更後の URL のイメージは随時 Web サービスの URL 初期値に表示されます。設定が終わったら、[**次へ**]ボタンを押します。



指定する文字列の最初と最後に「/」をつける必要はありません。

ここでWebサービスの作成を開始できます

この画面が出たら、Web サービスを作成するための最低限の設定は終わったことを表します。[**終了**]ボタンを押すと、Web サービスの作成を開始します。[**次へ**]ボタンを押すと、詳細設定の画面に移ります。

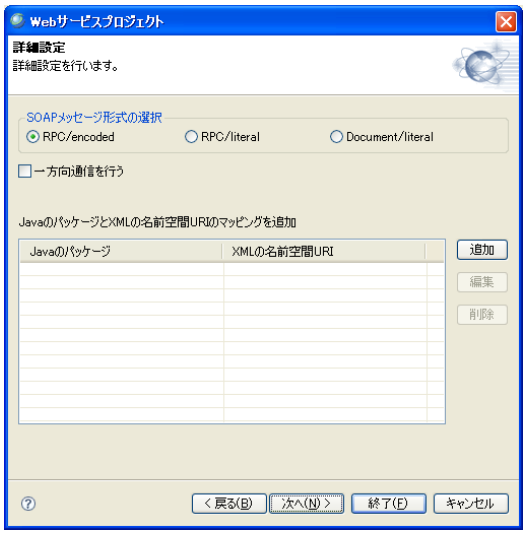


MEMO

「foo/var/service」のような深い階層を設定することもできます。

詳細設定

詳細設定を行う画面です。



SOAP メッセージ形式の選択

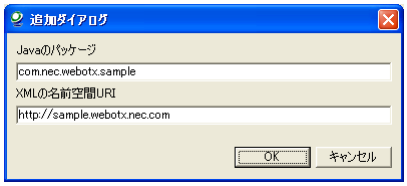
SOAP エンジンが作成する SOAP メッセージの形式を **RPC/encoded**、**RPC/literal**、**Document/literal** から選択します。**RPC/encoded** は SOAP 仕様に定められた RPC のエンコード方法で SOAP メッセージを作成し、SOAP で RPC を実現することを目的とするものです。**RPC/literal** は SOAP 仕様に定められた方法で RPC を実現しますが、XML スキーマで定義されたデータ型を使用することや、WSDL に定義された XML スキーマに従った書式の XML をやりとりすることに主眼が置かれます。**Document/literal** は SOAP Body の中に WSDL で定義された XML スキーマに従った任意の XML 文書を入れて送ることを目的とした形式です。このウィザードで Web サービスを作成する場合、RPC/literal と Document/literal の SOAP メッセージは全く同じ形式になりますが、RPC/literal では XML スキーマで定義されるデータ型を参照した WSDL になるのに対し、Document/literal では「型」の概念が無いため、WSDL 内で SOAP Body 内に入る XML の書式が全て XML スキーマで表現されるという特徴があります。

一方向通信を行う

クライアントからサーバの方向のみ SOAP メッセージを送る場合にチェックします。通常、Web サーバのタイムアウトまでビジネスロジックからの応答を待ち続けますが、ここでチェックすることによって、クライアントからの SOAP メッセージを正常に受け取ったら、ビジネスロジックの応答を待たずに HTTP 202 を返却し、一方的に SOAP 通信を終了させます。ビジネスロジックやそのバックエンドが非同期で実装されている場合など、非同期の Web サービスを実現したい場合にご活用ください。RPC/encoded、RPC/literal の時は void（返却値なし）のメソッドのみで一方向通信が成立します。Document/literal の時は返却値の有無に関わらず強制的に一方向通信になります。また、Document/literal の時は RemoteException 以外のユーザ定義の例外をスローしないでください。

Java のパッケージと XML の名前空間 URI のマッピングを追加

Web サービス化するメソッドの**引数**か**返却値**に Bean 型が含まれている場合に任意で指定することができます。追加するには、[**追加**]ボタンを押すと表示される「**追加ダイアログ**」を使用します。Java のパッケージには Bean が属するパッケージ名を、XML の名前空間 URI には、Bean が Web サービス（XML）の空間で振舞う時に属する名前空間 URI を指定し、[**OK**]ボタンを押します。



MEMO

WS-I モードにチェックしている場合には、RPC/literal か Document/literal しか選択できません。

MEMO

パッケージ名と名前空間 URI のマッピングは、ここで指定しなければ自動で行われます。その際の規則は、「http://パッケージ名のドット区切りで逆順の文字列」になります。

MEMO

WSDL も自動的に作られるので、WSDL に XML スキーマを定義することについて気にする必要はありません。

追加したマッピングを変更する時は、リストで変更したいマッピングを選択し、[**編集**]ボタンを押します。追加ダイアログが表示されるので、修正して[**OK**]ボタンを押します。マッピングを削除する時は、リストで変更したいマッピングを選択し、[**削除**]ボタンを押します。

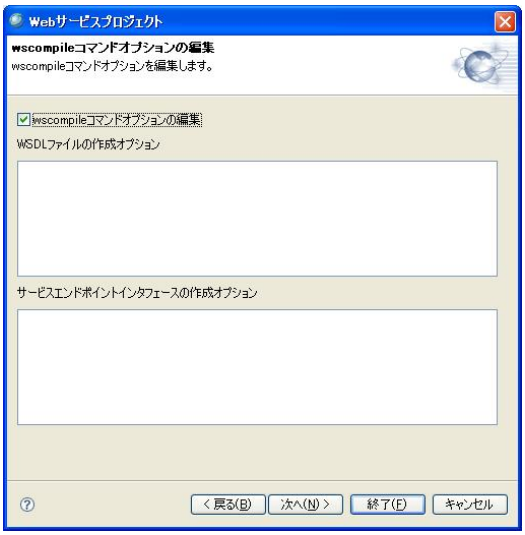
設定が終了したら、[**次へ**]ボタンを押します。

Document/literal を選択したとき、Web サービス化するメソッドに void（返却値なし）のものが含まれる場合、必ず「一方向通信を行う」にチェックしてください。配備に失敗します。

パッケージ名と名前空間 URI のマッピングを設定する場合は、必要な設定をすべて行ってください。自動でマッピングする機能は、ここで設定を 1 つでも行うと働きません。

wscompileコマンドオプションの編集

wscompile コマンドのオプションを編集する画面です。



WSDL ファイルの作成オプション

WSDL ファイルを作成する際に使用するオプションを入力します。

指定可能なオプションは下記表をご参照下さい。

サービスエンドポイントインタフェースの作成オプション

サービスエンドポイントインタフェースを作成する際に使用するオプションを入力します。

指定可能なオプションは下記表をご参照下さい。

WebOTX Developer’s Studio では既定で指定しているオプションがあり、この画面で既定のオプション設定と異なるオプションを入力した場合、画面で指定されたオプションが有効となります。

この画面では下記のオプションを指定できます。

オプション	説明
-define	WSDL を生成する場合に指定します。「サービスエンドポイントインタフェースの作成オプション」 では指定できません。
-classpath <path>	入力クラスファイルの検索場所を指定します。

-f:<features>	JAX-RPC の feature を指定します。指定できる Feature の一覧は下記の Feature 一覧テーブルをご参照ください。 複数個指定する場合は、カンマ「,」で区切って指定して下さい。
-features:<features>	-f:<features> と同様です。
-mapping <file>	マッピングファイルの生成場所を指定します。
-keep	Java ソースファイルを生成します。デフォルトでは、ソースファイル自体が生成されずにコンパイル結果のみが生成されます。
-g	デバッグ情報を生成します。
-model <file>	内部モデルをファイルに出力します。
-O	生成されたコードを最適化します。
-httpproxy:<host>:<port>	HTTP プロキシサーバを指定します。デフォルトのポートは 8080 です。

<features> には以下の値が指定可能です。

Feature	説明	WSDL 作成	SEI 作成
datahandleronly	常に属性をデータハンドラ型にマッピングします。	×	○
donotoverride	すでに生成が終了しているクラスファイルの再生成を行わないようにします。クラスファイルの上書きを禁止します。	○	○
donotunwrap	WS-I モード選択時の document/literal ラッパー要素のアンラッピングを無効にします。	×	○
explicitcontext	サービスコンテキストマッピングを明示的に有効にします。	×	○
jaxbenumtype	匿名列挙をそのベース型にマッピングします。	×	○
noencodedtypes	エンコーディングの型情報を無効にします。	○	○
nomultirefs	複数の参照を無効にします。	○	○
norpcstructures	rpc 構成を出力しません。	×	○
novalidation	インポートした WSDL の検証を行いません。	×	○
resolveidref	xsd:IDREF(ID 参照値という意味のデータ型)を解釈します。	×	○
searchschema	type 要素用に XML スキーマを探します。	×	○
serializeinterfaces	直接インタフェースの型とシリアライズすることを有効にします。	○	○

rpcliteral、wsi、documentliteral、useonewayoperations、nodatabinding の Feature については、ウィザード中の選択に依存するため、この画面では指定できません。

Web Services Securityの設定 1

ユーザネームトークンの暗号化を行うには、事前に暗号化構成の登録が必要です。

MEMO

暗号化構成の登録方法については 2.1.10 セキュリティをご覧ください。

MEMO

認証方式が WebOTX の場合とパスワードタイプがテキストの場合、パスワードが

この画面では、ユーザネームトークンに関する設定を行います。

■ユーザネームトークンによる認証機能を利用する

ユーザネームトークンによる認証を行いたい場合はチェックします。

■認証方式

サーバ側で認証に使うユーザ名とパスワードを取得する方法について選択することができます。ID/Password では、お客様ご自身でユーザ名とパスワードを確認する処理を作成することができます。WebOTX では、WebOTX が管理するユーザ情報を利用します。

■パスワードタイプ

認証方式で ID/Password を指定すると選択することができます。**ダイジェスト**は、パスワードを SOAP メッセージに格納する際、WebOTX 独自の方法でダイジェスト値に変換します。**テキスト**は、プレーンテキストをそのまま格納します。

■暗号化する

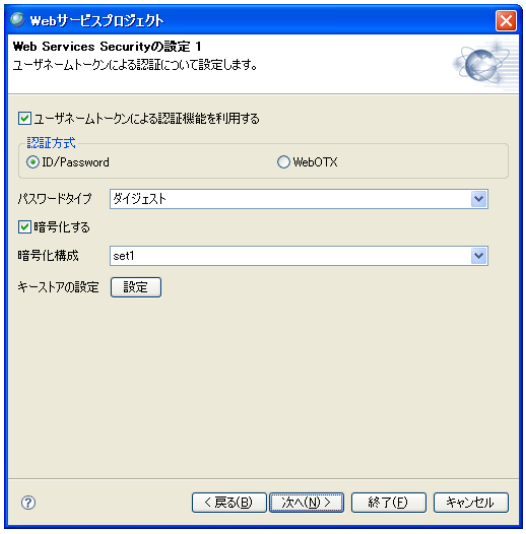
SOAP メッセージに格納するユーザ名、パスワードを暗号化するときにチェックします。

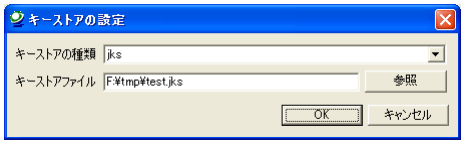
■暗号化構成

暗号化に関する設定を行います。事前に設定しておいた暗号化構成が表示されますので、使用するものを選択します。

■キーストアの設定

暗号化に用いる鍵を格納したキーストアを設定します。[**設定**]ボタンを押し、**キーストアタイプ**を選択し、**キーストアファイル**を絶対パスで指定します。





WebOTX 認証、署名による認証については、どちらかしか使用することができません。

暗号化構成の「暗号化方式」で「TYPE2」または「TYPE3」を選択している時は、「jceks」タイプのキーストアを利用してください。

Web Services Securityの設定 2

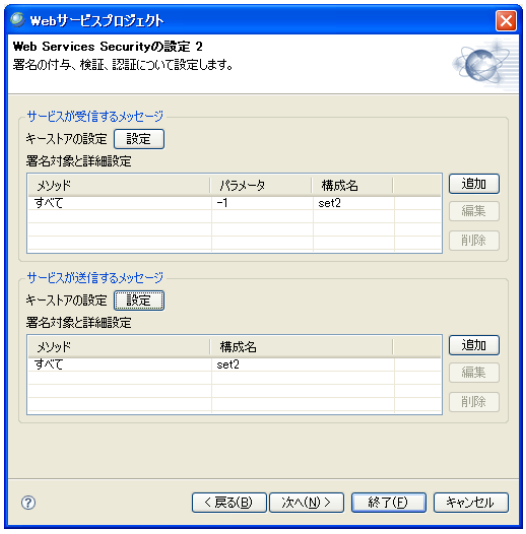
署名を行うには、事前に署名構成の登録が必要です。

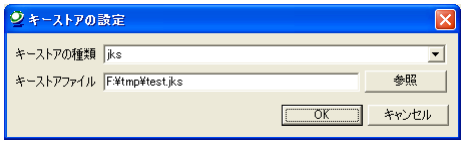
覗き見される危険性があります。「暗号化する」にチェックして暗号化するか、SSL を使用するなどして通信路の安全を確保することをおすすめします。

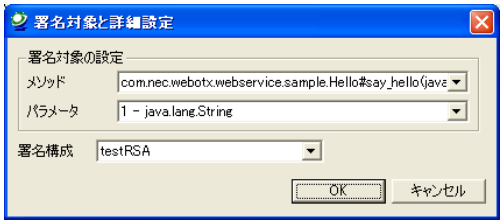
MEMO

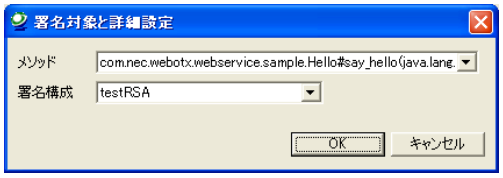
キーストアの設定は、2 画面先の暗号化についての設定（サーバが受信するメッセージ）と同じです。

この画面では、署名に関する設定をおこないます。署名は、今作ろうとしているWebサービスが送受信するメッセージに対してそれぞれ付与、検証することができます。









■キーストアの設定

署名に用いる鍵を格納したキーストアを設定します。[**設定**]ボタンを押し、**キーストアタイプ**を選択し、**キーストアファイル**を絶対パスで指定してください。

■署名対象と詳細設定

署名対象と、その対象を署名するときに使用する構成を指定します。[**追加**]ボタンを押し、署名対象と詳細設定ダイアログで、署名を行いたい場所をメソッド、パラメータで指定します。メソッド、パラメータで選択できるのは、メソッドの選択画面でWebサービス化する指定を行ったものです。また、その署名対象についてあらかじめ設定しておいた署名構成を割り当てることにより、署名の詳細設定を行います。署名対象と詳細設定の一覧表で1つの設定を選択し[**編集**]ボタンを押すと、署名対象と署名構成の設定をやり直すことができます。また、[**削除**]ボタンを押すとその設定を削除することができます。

サービスが受信するメッセージの場合

メソッドで「すべて」を選ぶとSOAP Bodyを署名します。特定のメソッドを指定するとパラメータが選択できるようになります。パラメータで「すべて」を選ぶと、対応するオペレーション要素を署名します。パラメータのコンボボックスには何番目の引数かということと、その型を表示します。特定のパラメータを指定すると、そのパラメータの要素を署名します。署名構成は、認証を行う設定を含んだものを選択することができます。

サービスが送信するメッセージの場合

メソッドで「すべて」を選ぶとSOAP Bodyを署名します。特定のメソッドを指定すると、対応するオペレーション要素を署名します。署名構成は、認証を行う設定を含んだものを選択してはいけません。

認証は、WebOTX認証、署名による認証のどちらかしか使用することができません。

認証を行う設定を持つ複数の署名構成を同時に指定することはできません。

認証を行う設定を持つ署名構成と、そうでない署名構成を同時に指定することはできません。

複数のメソッドに対してそれぞれ別々の署名構成を割り当てることはできません。

Web Services Securityの設定 3

MEMO

SOAP メッセージ形

暗号化を行うには、事前に暗号化構成の登録が必要です。

この画面では、暗号化に関する設定をおこないます。暗号化は、今作ろうとしているWebサービスが送受信するメッセージに対してそれぞれ行うことができます。

■キーストアの設定

暗号化に用いる鍵を格納したキーストアを設定します。[**設定**]ボタンを押し、**キーストアタイプ**を選択し、**キーストアファイル**を絶対パスで指定してください。

■暗号化対象と詳細設定

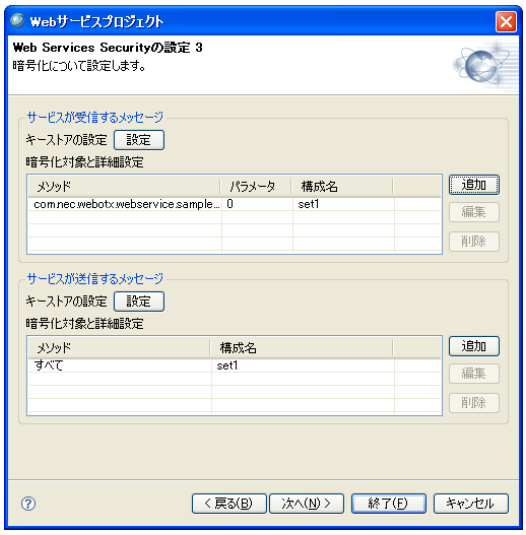
暗号化対象と、その対象を暗号化するときに使用する構成を指定します。[**追加**]ボタンを押し、暗号化対象と詳細設定ダイアログで、署名を行いたい場所をメソッド、パラメータで指定します。メソッド、パラメータで選択できるのは、メソッドの選択画面でWebサービス化する指定を行ったものです。また、その暗号化対象についてあらかじめ設定しておいた暗号化構成を割り当てることにより、暗号化の詳細設定を行います。暗号化対象と詳細設定の一覧表で1つの設定を選択し[**編集**]ボタンを押すと、暗号化対象と暗号化構成の設定をやり直すことができます。また、[**削除**]ボタンを押すとその設定を削除することができます。

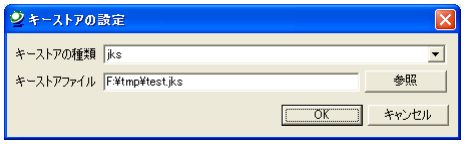
サービスが受信するメッセージの場合

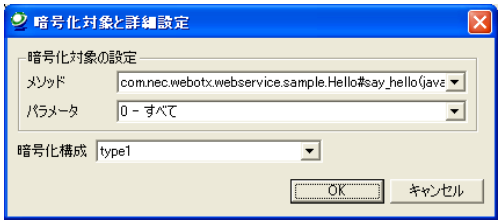
メソッドを指定すると、パラメータが選択できるようになります。パラメータで「すべて」を選ぶと、対応するオペレーション要素の内容を暗号化します。パラメータのコンボボックスには何番目の引数かということと、その型を表示します。特定のパラメータを指定すると、そのパラメータの要素の内容を暗号化します。

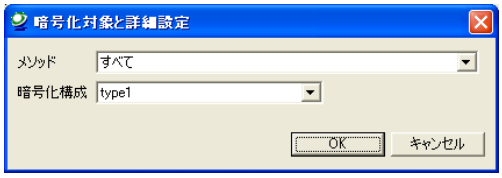
サービスが送信するメッセージの場合

メソッドを指定すると、対応するオペレーション要素の内容を暗号化します。









暗号化構成の「暗号化方式」で「TYPE2」または「TYPE3」を選択している時は、「jceks」タイプのキーストアを利用してください。

WebOTX Standard/Enterprise Edition ではオペレーション名を判断する必要があるため、SOAP Body 要素の内容全体を暗号化する設定はできません。

Web Services Securityの設定 4

タイムスタンプの署名を行うには、事前に署名構成の登録が必要です。

式が encoded で、配列や JavaBean の受け渡しをし、その部分を暗号化する場合、オペレーション要素の外に実際に受け渡しする値が書かれるため、本当に暗号化したい部分が暗号化されるかどうかについては保証しません。配列や JavaBean の中にプリミティブ型以外の値を持つ構造のパラメータを含むメソッドの暗号化には十分ご注意ください。
なお、SOAP メッセージ形式が literal の場合は、完全に暗号化されることを保証します。暗号化を利用する場合は literal（WS-I モードを含む）の使用を推奨します。

この画面では、タイムスタンプに関する設定をおこないます。タイムスタンプは、今作ろうとしている Web サービスが送受信するメッセージに対してそれぞれ付与・検証することができます。

■メッセージの有効時間

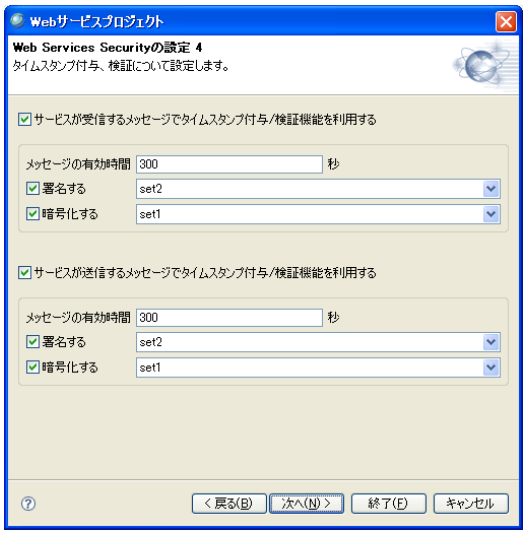
メッセージの有効時間を秒単位で指定します。初期値は 300 秒です。この有効時間を越すと、たとえ正しいメッセージが送られてきても不正なメッセージとして判断されます。HTTP のレイヤーでセッションタイムアウトを設定する場合、それよりも短い時間にする必要があります。

■署名する

タイムスタンプの値に署名する場合、チェックボックスにチェックを入れ、あらかじめ登録してある署名構成を選択します。

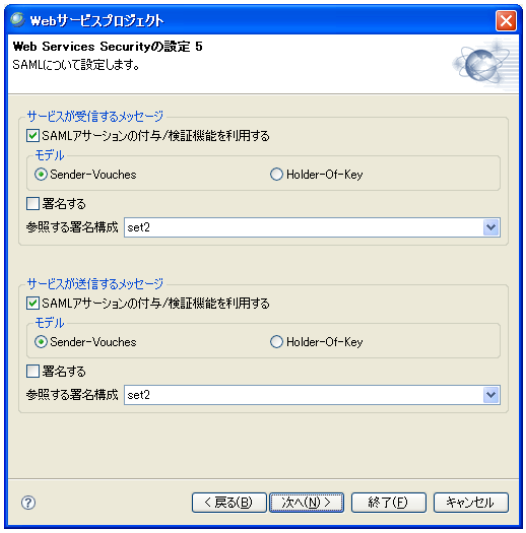
■暗号化する

タイムスタンプの値を暗号化する場合、チェックボックスにチェックを入れ、あらかじめ登録してある暗号化構成を選択します。



Web Services Securityの設定 5

SAML アサーションを署名する、または Holder-Of-Key モデルを使用するには、事前に署名構成の登録が必要です。



この画面では、SAML アサーションの付与・検証についての設定を行います。SAML は、今作ろうとしている Web サービスが送受信するメッセージに対してそれぞれ付与・検証することができます。

■モデル

この機能の処理モデルを選択します。

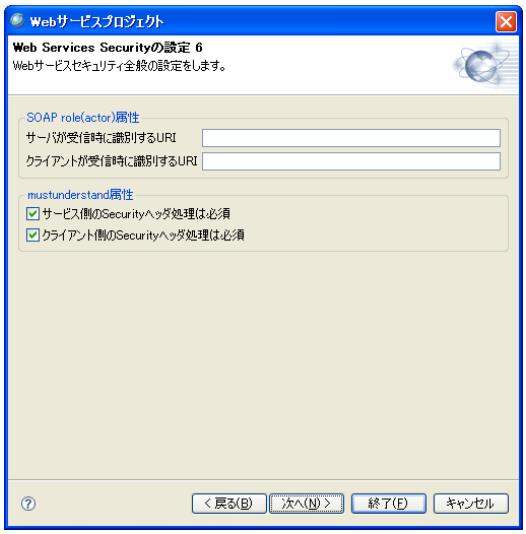
■署名する

SAML アサーションに署名する場合、チェックボックスにチェックします。

■参照する署名構成

Holder-Of-Key モデルを選択した場合、署名する場合に、あらかじめ登録してある署名構成を選択します。署名構成は、「署名を使用して認証を行う」にチェックし、トラストアンカリストを指定しているものでなければなりません。また、Holer-Of-Key モデルかつ署名する場合、署名構成では「鍵または証明書の参照方法」で「SKIKeyIdentifier」を指定していなければなりません。

Web Services Securityの設定 6



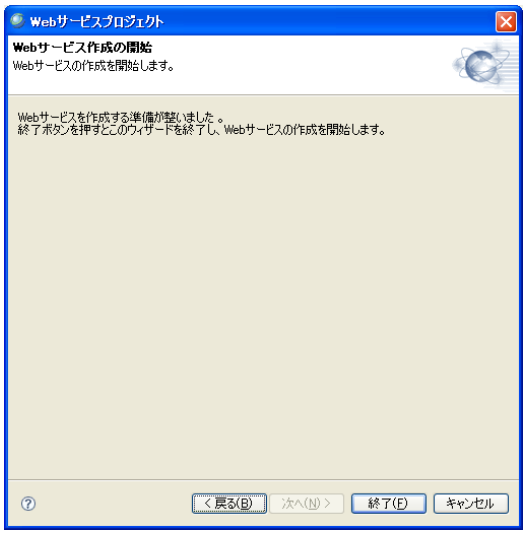
この画面では、SOAP Role（SOAP actor）、mustunderstand の値を設定します。

SOAP Role（SOAP actor）を設定すると、SOAP メッセージの受信側がセキュリティヘッダを処理すべきかどうかの判断をすることができます。アグリゲーションサービスのように、SOAP メッセージを次のサービスへ持ちまわり、ある特定のセキュリティ処理をある特定のサービスが受け持つような場合、設定しておく必要があります。特に設定する必要がない場合は、空にしておいて構いません。

mustunderstand 属性は、セキュリティヘッダの処理が必須かどうかを設定することができます。例えば、アグリゲーションサービスのように、SOAP メッセージを次のサービスへ持ちまわり、ある特定のセキュリティ処理をある特定のサービスが受け持つような場合、経由するサービスは必ずしもセキュリティヘッダを処理しなければならないというわけではありません。この場合ならば、経由するサービスを構築するときにはチェックをはずす必要があります。

Webサービス作成の開始

Web サービスを作成するための設定はすべて終了しました。[**終了**]ボタンを押して、Web サービスの作成を開始します。



Webサービスプロジェクト

ここでは、Web サービスプロジェクトの内容について説明します。

■binフォルダ

src フォルダにあるソースコードをビルドした結果が格納されます。

■srcフォルダ

Web サービスの動作に必要な各種ソースコードや、その Web サービスをテストするためのクライアントコードなどが生成されます。主なクラス、プロパティ構成は次の通りです。

クラス名	役割
Web サービス名_Service	サービスインタフェースです。Web サービスを仮想化したインタフェースで、ポートを内封します。
Web サービス名_Service_Impl	サービスインタフェースの実装クラスです。クライアント側の初期化をになっており、JAX-RPC Handler の初期化パラメータも書かれます。
Web サービス名_PortType	サービスエンドポイントインタフェースです。
Web サービス名_PortType_Tie	タイクラスです。サービス側で SOAP メッセージの受け口になるクラスです。JAX-RPC Handler の初期化パラメータも書かれます。
Web サービス名_PortType_Stub	クライアントが利用するスタブクラスです。
Web サービス名 SoapBindingImpl	ウィザードで指定したビジネスロジックを呼び出す Web サービスの実装クラスです。Web サービスの実装形式が Web アプリケーションの時のみ生成されます。
Web サービス名 Client	テスト用のクライアントです。スタブクラスを利用しながら Web サービスの呼び出しを行います。
Main	テスト用のクライアントを呼び出して動作させるためのクラスです。クライアントを作成する時は、このクラスをカスタマイズすると簡単です。
ClientSenderCallbackHandler	クライアントが送信するメッセージについて必要なユーザ名（またはエイリアス）、パスワードを設定するためのクラスです。WS-Security のユーザネームトークン/署名/暗号化機能を利用する場合に生成されます。このクラスは必ずカスタマイズする必要があります。カスタマイズ方法は 2.1.10 セキュリティをご覧ください。
ClientReceiverCallbackHandler	クライアントが受信するメッセージについて必要なパスワードを設定するためのクラスです。WS-Security の署名/暗号化機能を利用する場合に生成されます。このクラスは必ずカスタマイズする必要があります。カスタマイズ方法は 2.1.10 セキュリティをご覧ください。
ServerSenderCallbackHandler	サーバが送信するメッセージについて必要なユーザ名（またはエイリアス）、パスワードを設定するためのクラスです。WS-Security の署名/暗号化機能を利用する場合に生成されます。このクラスは必ずカスタマイズする必要があります。カスタマイズ方法は 2.1.10 セキュリティをご覧ください。
ServerReceiverCallbackHandler	サーバが受信するメッセージについて必要なパスワードを設定するためのクラスです。WS-Security のユーザネームトークン/署名/暗号化機能を利用する場合に生成されます。このクラスは必ずカスタマイズする必要があります。カスタマイズ方法は 2.1.10 セキュリティをご覧ください。
KeystoreCallbackHandler	キーストアのパスワードを設定するためのクラスです。WS-Security の署名/暗号化機能を利用する場

	合に生成されます。このクラスは必ずカスタマイズする必要があります。カスタマイズ方法は 2.1.10 セキュリティをご覧ください。

■baseフォルダ

Web サービス作成ウィザード中で、WAR ファイル、または EJB-JAR ファイルをビジネスロジックとして指定した場合に、それを展開した内容が置かれます。Web サービスプロジェクト作成後、ビジネスロジックの WAR または EJB-JAR ファイルの中身を置換または追加したい場合は、このフォルダ内のファイルを修正します。また、Web サービスの実装形式を Web アプリケーションとする場合、HTML や JSP のような任意のファイルやフォルダを base フォルダ直下に置けば、WAR ファイルに含めることができます。

ビジネスロジックを置換するとき、Web サービス化したメソッドのインタフェースを変えることはできません。

■configフォルダ

Web サービスの実装形式を Web アプリケーションにした場合、ビジネスロジックのパッケージ、クラスに関する情報を Web サービスのオペレーション名に自動的に付加します。こうすることによって、Web サービスのオペレーションと、ビジネスロジックのメソッドを一対一で結び付けています。この動作によってオペレーション名は次のように命名されます。

<**メソッド名**>_<**ビジネスロジックのクラス名**><**パッケージに対応した数字**>

パッケージに対応した数字は、ビジネスロジックとして複数のパッケージから同時にメソッドを選択した時に自動的に割り振られます。どのパッケージがどの番号が対応しているかについて、config フォルダの中に「web サービス名.properties」という名前で生成されるプロパティファイルに書かれます。

■libフォルダ

Web サービス作成ウィザード中で、JAR ファイルをビジネスロジックとして指定した場合、その JAR ファイルはこのフォルダに自動でコピーされます。また、別途ビジネスロジックや Web サービスから参照する JAR ファイルを作成した場合は、その JAR ファイルをこのフォルダに手動でコピーすると、ここに置かれた JAR ファイルは、WAR ファイルの「/WEB-INF/lib/」配下に封入されます。

■xmlフォルダ

WSDL や各種配備記述子などの XML ファイルがここに生成されます。アーカイブ時、base フォルダにある配備記述子よりも、ここにあるファイルが優先的に封入されますので、配備記述子を編集したい場合は xml フォルダ配下のファイルを編集してください。

■keystoreフォルダ

Web サービス作成ウィザード中で、署名、暗号化など、キーストアを利用する設定を行った場合、指定したキーストアファイルがこのフォルダにコピーされます。

JAX-RPCサービスエンドポイントのカスタマイズ

ここでは、Web サービスの実装形式が Web アプリケーションのときについて、エンドポイントをカスタマイズする方法を説明します。

実装クラスのカスタマイズ

サービスエンドポイントの実装クラスをカスタマイズすると、Web サービスのエンドポイントのインスタンスが作られる、または削除されるタイミングでいろいろな処理を追加することができます。また、コンテキストをビジネスロジックに引き渡して処理したり、コンテキストから取得できる情報を利用してビジネスロジックの呼び出しをコントロールすることもできます。これからカスタマイズするサービスエンドポイントの実装クラスは、ウィザードにより生成された「～SoapBindingImpl」クラスです。次のようにカスタマイズします。

- javax.xml.rpc.server.ServiceLifecycle を implements に加える。
- init、destroy メソッドを実装する。
- init メソッドの引数を javax.xml.rpc.server.ServletEndpointContext でキャストし、コンテキスト情報などを取り出す。

ServletEndpointContext を取得することで、HTTP セッションの取得、SOAP や Servlet のコンテキストの取得、Principal の取得、SOAP Role（SOAP actor）属性のチェックを行うことができるようになります。これらをビジネスロジックを呼び出しているメソッドの中で利用します。

コーディング例

```
import javax.xml.rpc.server.ServiceLifecycle;
import javax.xml.rpc.server.ServletEndpointContext;

public class TestSoapBindingImpl implements
com.nec.webotx.webservice.director.hello.hello_PortType, ServiceLifecycle {

    private ServletEndpointContext jaxrpcContext;

    public void init(Object context) throws ServiceException {
        jaxrpcContext = (ServletEndpointContext) context;
    }

    public void destroy() {
        jaxrpcContext = null;
    }

    public void getMessageContext() {
        jaxrpcContext.getMessageContext();
    }

    ～
}
```

Basic認証を使用する設定の追加

HTTP の Basic 認証を使用したいとき、次の手順でユーザ登録（ユーザ名、パスワード）を登録し、Basic 認証機能を有効にします。

- 運用管理コマンド（otxadmin）でユーザ登録を行う

まず、ログインします。ここでは、あらかじめ登録されているドメイン管理ユーザでログインする例を示します。

```
login -user admin -password adminadmin -port 6212
```

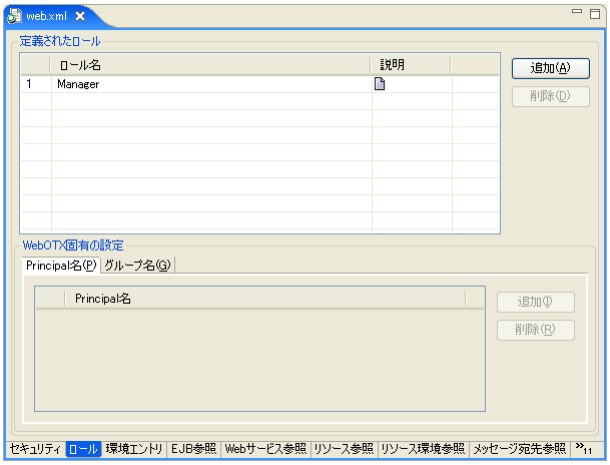
create-file-user コマンドでユーザを追加します。ここでは、sample_user という名前のユーザをパスワード sample で登録する例を示します。コマンドオプションの詳細については、運用管理コマンドリファレンスマニュアルをご参照ください。

```
create-file-user --userpassword sample --groups sampleGroup sample_user
```

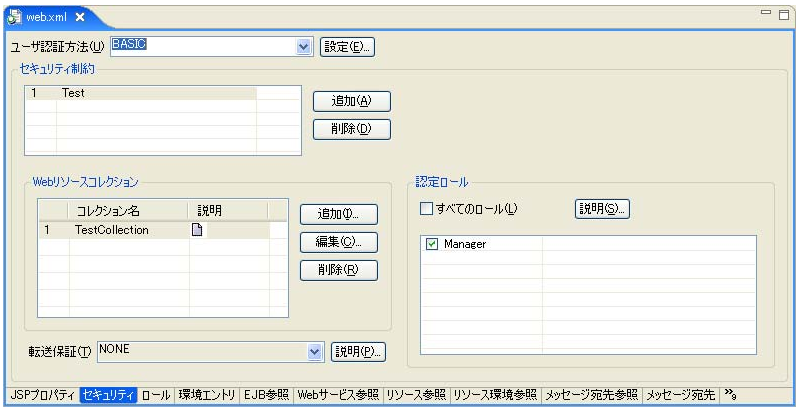
- web.xml に追記する

Web サービスプロジェクトの xml フォルダにある「web.xml」ファイルを配備記述子エディタで開き、次のように設定を追加します。

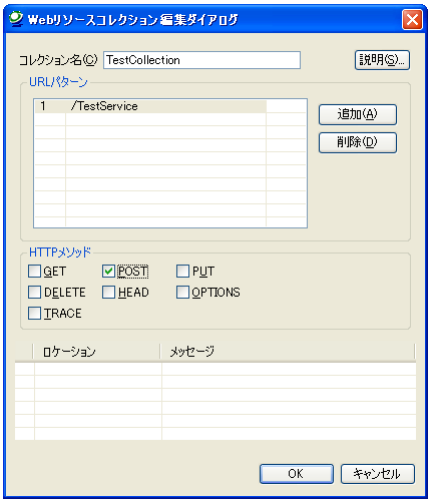
ロールタブを開き、**追加**ボタンを押し、ロール名を指定します。



セキュリティタブを開き、ユーザ認証方法で「**BASIC**」を選択します。セキュリティ制約の**追加**ボタンを押し、名前をつけます。設定ロールに一覧表示されたロールから設定対象のロールにチェックします。Web リソースコレクションで**追加**ボタンを押します。



表示されたダイアログで、コレクション名、URL パターンを設定し、HTTP メソッドの「POST」にチェックし、**OK**ボタンを押します。なお、WSDL の公開についても認証したい場合は、「GET」にもチェックを入れます。



最後に、配備記述子エディタで web.xml を保存します。

- nec-web.xml に追記する

Web サービスプロジェクトの xml フォルダにある「nec-web.xml」ファイルをテキストエディタで開き、次のように **security-role-mapping 要素**を追加します。role-name 要素は web.xml で設定した認証ロール、principal-name 要素はユーザ登録時に指定したユーザ名、group-name 要素はユーザ登録時に指定したグループ名を指定します。principal-name 要素と group-name 要素はどちらか片方だけの指定だけでも構いません。例えば、複数のユーザにアクセス権限を与える時、group-name 要素で対象のグループ

を指定します。

```
<?xml version="1.0" encoding="UTF-8"?>
<nec-web-app xmlns="http://java.sun.com/xml/ns/j2ee">
  <context-root>/sample</context-root>
  <class-loader delegate="true"/>
  <security-role-mapping>
    <role-name>Manager</role-name>
    <principal-name>sample_user</principal-name>
    <group-name>sampleGroup</group-name>
  </security-role-mapping>
</nec-web-app>
```

EJBサービスエンドポイントのカスタマイズ

ここでは、Web サービスの実装形式が EJB のときについて、エンドポイントをカスタマイズする方法を説明します。

実装クラスのカスタマイズ

サービスエンドポイントの実装クラスをカスタマイズすると、クライアントから受け取る SOAP メッセージのコンテキストを取得することができます。こうすることで、取得したコンテキストをビジネスロジックの中で利用することができます。これからカスタマイズするサービスエンドポイントの実装クラスは、ビジネスロジックの EJB です。次のようにカスタマイズします。

- javax.ejb.SessionContext を implements に加える。
- getMessageContext()を実装し、コンテキストを取得する。

Basic認証を使用する設定の追加

HTTP の Basic 認証を使用したいとき、次の手順でユーザ登録(ユーザ名、パスワード)を登録し、Basic 認証機能を有効にします。

- 運用管理コマンド(otxadmin)でユーザ登録を行う

まず、ログインします。ここでは、あらかじめ登録されているドメイン管理ユーザでログインする例を示します。

```
login -user admin -password adminadmin -port 6212
```

create-file-user コマンドでユーザを追加します。ここでは、sample_user という名前のユーザをパスワード sample で登録する例を示します。コマンドオプションの詳細については、運用管理コマンドリファレンスマニュアルをご参照ください。

```
create-file-user --userpassword sample --groups sampleGroup sample_user
```

- ejb-jar.xml に追記する

Web サービスプロジェクトの xml フォルダにある「ejb-jar.xml」ファイルをテキストエディタで開き、ejb-jar 要素の子要素として次のようにして **assembly-descriptor 要素**を追加します。この要素は enterprise-beans 要素と、relationships 要素(省略される場合があります)の次に記述します。

```
<ejb-jar ・・・>
  <enterprise-beans>
～省略～
  </enterprise-beans>
  <relationships>
～省略～
  </relationships>
  <assembly-descriptor>
    <security-role>
      <role-name>ROLE_NAME</role-name>
    </security-role>
    <method-permission>
      <role-name>ROLE_NAME</role-name>
```

```
        <method>
          ～ここの書き方は後述します～
        </method>
      </method-permission>
    </assembly-descriptor>
～省略～
</ejb-jar>
```

2 つある role-name 要素にはロール名を記述します。ロール名は半角英数字を使います。security-role 要素と method-permission 要素はセットなので、必ず 2 つとも同じロール名にします。ロールを複数定義する時は、他のロール名と同じにならないように注意します。method 要素には Basic 認証の対象となるメソッドを指定するところですが、次のようにいろいろな記述方法があります。

(1) EJB に含まれる全てのメソッドへのアクセスを対象にする

ejb-name 要素には、Web サービス化した EJB の EJB 名を指定します。method-name 要素には「*」を指定します。

```
        <method>
          <ejb-name>EJB_NAME</ejb-name>
          <method-name>*</method-name>
        </method>
```

(2) EJB の特定のメソッド名のメソッドへのアクセスのみ対象にする

オーバーロードしているものも含めて、ある特定のメソッド名を持つメソッドへのアクセスを対象にする方法です。ejb-name 要素には、Web サービス化した EJB の EJB 名を指定します。method-name 要素には対象にしたいメソッド名を指定します。

```
        <method>
          <ejb-name>EJB_NAME</ejb-name>
          <method-name>METHOD_NAME</method-name>
        </method>
```

(3) EJB の特定のメソッドへのアクセスのみ対象にする

引数まで指定し、ある特定のメソッドのみを対象にする方法です。ejb-name 要素には、Web サービス化した EJB の EJB 名を指定します。method-name 要素には対象にしたいメソッド名を指定します。method-param 要素は第 1 引数から順番に引数の型を指定します。配列を指定したい場合には、型名の最後に「[]」を追加します。

```
        <method>
          <ejb-name>EJB_NAME</ejb-name>
          <method-name>METHOD_NAME</method-name>
          <method-params>
            <method-param>int</method-param>
            <method-param>java.lang.String</method-param>
            <method-param>foo.Bar[]</method-param>
          </method-params>
        </method>
```

(4) Web サービスへのアクセスの場合のみ対象にする

1 から 3 の方法では、Web サービスのアクセスに限らず、EJB に対する全てのアクセスが認証の対象となります。認証の対象を Web サービスだけにしたい場合、method-intf 要素を追加します。method-intf 要素の値は「ServiceEndpoint」にします。method-intf 要素は、1 から 3 のどの方法についても追加することができます。

```
        <method>
          <ejb-name>EJB_NAME</ejb-name>
          <method-intf>ServiceEndpoint</method-intf>
          <method-name>METHOD_NAME</method-name>
          <method-params>
            <method-param>int</method-param>
            <method-param>java.lang.String</method-param>
            <method-param>foo.Bar[]</method-param>
          </method-params>
        </method>
```

- nec-ejb-jar.xml に追記する

Web サービスプロジェクトの xml フォルダにある「nec-ejb-jar.xml」ファイルをテキストエディタで開き、nec-ejb-jar 要素の子要素として **security-role-mapping 要素**を、webservice-endpoint 要素の子要素として **login-config 要素**を次のようにして追加します。

```
<nec-ejb-jar>
～省略～
  <security-role-mapping>
    <role-name>ROLE_NAME</role-name>
    <principal-name>sample_user</principal-name>
  </security-role-mapping>
～省略～
</nec-ejb-jar>
```

security-role-mapping 要素は nec-ejb-jar 要素直下に記述します。その子要素である role-name 要素には、ejb-jar.xml で指定したロール名を記述します。principal-name 要素には、ユーザ登録時に指定したユーザ名を指定します。principal-name 要素は複数指定することもできます。

```
<nec-ejb-jar>
～省略～
  <enterprise-beans>
    <ejb>
      <ejb-name>EJB_Name</ejb-name>
      <webservice-endpoint>
        <port-component-name>・・・</port-component-name>
        <endpoint-address-uri>・・・</endpoint-address-uri>
        <login-config>
          <auth-method>BASIC</auth-method>
        </login-config>
        ～省略～
      </webservice-endpoint>
    </ejb>
  </enterprise-beans>
～省略～
</nec-ejb-jar>
```

login-config 要素は、上記ような固定の内容を記述します。記述する場所は webservice-endpoint 要素の子要素で port-component-name 要素、endpoint-address-uri 要素の次に記述します。なお、endpoint-address-uri 要素は省略される場合がありますので、その場合は port-component-name 要素の次に記述します。

クライアントのカスタマイズ

Web サービス作成ウィザードが生成するクライアントプログラムは、空の値を渡して返却値をコンソールに表示するというテストを目的とした内容となっています。実際に Web サービスクライアントを運用するには「Main クラス」をカスタマイズします。Main クラスでは、クライアントクラスをインスタンス化して Web サービスのオペレーションを呼び出しているだけです。クライアントクラスのコンストラクタの引数で、Web サービスの URL を指定できます。ここで URL を指定することにより、スタブ内でハードコーディングされている URL を無効にすることができます。

添付ファイルを使用する場合

Web サービス化したメソッドの引数に、添付ファイルに対応する型が含まれている場合、型によっては必ず Main クラスをカスタマイズしなければならない場合があります。封入する添付ファイルのパスを指定するコードがある場合、初期値の「"filename"」が入っているためにそのままでは動作しないケースです。その場合、「"filename"」を実際のファイルパスへ置き換えてください。

Basic認証を使用する設定の追加

Stub クラスでプロパティをセットすることにより、Web サーバの Basic 認証に使用するユーザ名、パスワードの設定ができます。～_Stub クラスのコンストラクタに次のようなコードを追加します。

```
_setProperty(javax.xml.rpc.Stub.USERNAME_PROPERTY, “ユーザ名”);
_setProperty(javax.xml.rpc.Stub.PASSWORD_PROPERTY, “パスワード”);
```

ビジネスロジックの置換

Web サービスプロジェクトを作成した後でも、Web サービス化したメソッドの**インタフェースが変わらなければ**ビジネスロジックだけ置換することができます。ここでは、ビジネスロジックの置換手順について説明します。

インタフェースが変更になる場合は、Web サービス作成ウィザードをやり直す必要があります。

ビジネスロジックがプロジェクトの場合

ビジネスロジックが含まれるプロジェクトを直接修正することで、置換することができます。

ビジネスロジックがJARファイルの場合

ウィザード中で指定したビジネスロジックの JAR ファイルは、Web サービスプロジェクトの lib フォルダにコピーされているので、そのファイルを置換すればビジネスロジックを置換することができます。

ビジネスロジックがWAR・EJB-JARファイルの場合

ウィザード中で指定したビジネスロジックの WAR ファイルや EJB-JAR ファイルは、Web サービスプロジェクトの base フォルダに展開されているので、そのクラスファイル、もしくは JAR ファイルを置換すればビジネスロジックを置換することができます。

クラスパスに追加するライブラリに指定したJAR・ZIPファイルの置換

Web サービスプロジェクトからは、ウィザード中で指定したファイルを絶対パスで参照していますので、指定したファイルをそのまま置換してください。

HTML・JSPファイルなどの追加・置換

Web サービスの実装方式を Web アプリケーションにするとき、HTML や JSP などのファイルや任意のフォルダを追加することができます。また、ビジネスロジックとして WAR ファイルを指定したとき、その中に含まれる HTML や JSP などのファイルや任意のフォルダを置換することもできます。

いずれの場合も、Web サービスプロジェクトの base フォルダ配下にファイル、またはフォルダを置いてください。ただし、配備記述子については base フォルダ配下に置いても反映されません。

Webサービスアプリケーションのアーカイブ

完成した Web サービスアプリケーションをアーカイブし、サーバへの配備、クライアントへの配布ができるようにします。

WARファイル

メニューから、**ファイル**｜**エクスポート**を選択し、エクスポートダイアログを起動します。「**WAR ファイル（WebOTX）**」を選択し、[**次へ**]ボタンを押します。アーカイブを行いたい Web サービスプロジェクトを選択し、WAR ファイルの出力先を指定し、[**OK**]ボタンを押します。

EJB-JARファイル

メニューから、**ファイル**｜**エクスポート**を選択し、エクスポートダイアログを起動します。「**EJB-JAR ファイル**」を選択し、[**次へ**]ボタンを押します。アーカイブを行いたい Web サービスプロジェクトを選択し、EJB-JAR ファイルの出力先を指定し、[**OK**]ボタンを押します。

JARファイル（クライアント配布用）

クライアントに Web サービスクライアントのアプリケーションを配布する場合、JAR ファイルにアーカイブするのが一般的です。メニューから、**ファイル**｜**エクスポート**を選択し、エクスポートダイアログを起動します。「**JAR ファイル**」を選択し、[**次へ**]ボタンを押します。アーカイブに含めたい Web サービスクライアントを含むプロジェクトの **src** フォルダのみにチェックを入れ、JAR ファイルの**出力先**を指定し、[**OK**]ボタンを押します。

なお、Web サービスクライアント実行のためには、ここで作成した JAR ファイル以外に次のものが必要ですので、一緒に配布してください。

MEMO

SOAP 通信高速化設定については、運用編（チューニング）の Web サービスのチューニングを参照してください。

- jaxrpc-impl.jar
- saaj-impl.jar
- j2ee.jar
- resolver.jar
- serializer.jar
- xalan.jar
- xercesImpl.jar
- xml-apis.jar
- jsr173api.jar（SOAP 通信高速化設定追加時に必要）
- woparser.jar（SOAP 通信高速化設定追加時に必要）

Webサービスアプリケーションの配備

アーカイブした Web サービスアプリケーションの配備方法について説明します。

統合運用管理ツール

Web サービスアプリケーションの配備には「統合運用管理ツール」を使用することができます。WebOTX Developer のパースペクティブとして組み込まれます。統合運用管理ツールの使用方法についての詳細は、運用編を参照してください。

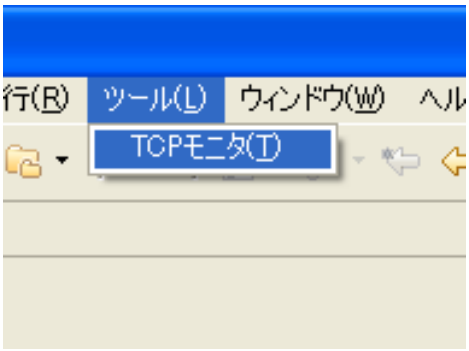
運用管理コマンド

Web サービスアプリケーションの配備には「運用管理コマンド」を使用することができます。運用管理コマンドの使用方法についての詳細は、運用編を参照してください。

TCPモニタ

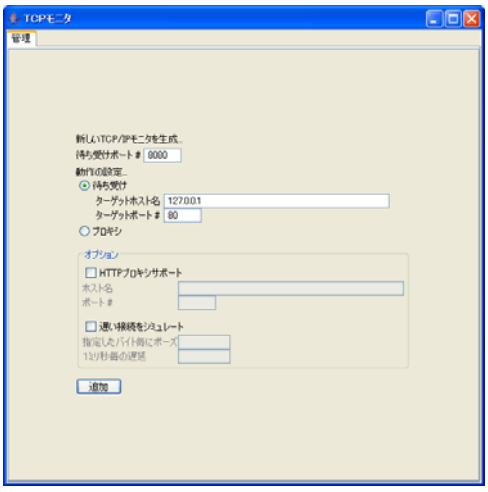
HTTP メッセージを中継することにより、サーバとクライアント間で交換されている SOAP メッセージを覗くためのツールが「TCP モニタ」です。ここでは、TCP モニタの使い方を説明します。

TCP モニタは、メニューから**ツール | TCP モニタ**を選択して起動します。



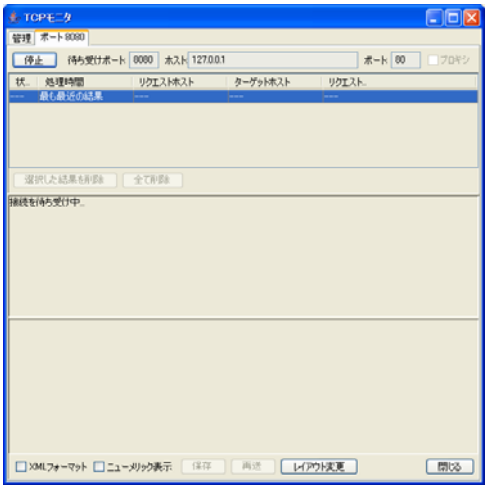
待ち受けポートにクライアントからリクエストを受け付ける**ポート番号**を指定します。ターゲットホスト名はサーバの**ホスト名**、ターゲットポートにはサーバの**ポート番号**を指定します。

設定が終わったら、[**追加**]ボタンを押します。



クライアントのポート番号でタブが作成されますので、そのタブを選択します。タブが作成された段階で、すでにこのポートのモニタリングは始まっていますので、この時点でクライアント・サーバ間を流れる SOAP メッセージを表示することができます。

XML フォーマットをチェックしておくと、SOAP メッセージに改行とインデントが入り、見やすくなります。



TCP モニタはテスト用、メッセージ確認用としてお使いください。実運用においてメッセージログを取る目的で稼動できる設計にはなっていません。

添付ファイルつき SOAP メッセージをモニタリングするときは、**XML フォーマット**のチェックをはずしてください。

メッセージダンプ機能

●クライアント側での送受信メッセージのダンプ

静的スタブ形式での呼び出し時に以下の Java オプションを加えることでクライアントが送受信する際のダンプメッセージを出力できます。本機能を利用することで TCP モニタを利用せずにメッセージを確認することができます。

```
-Dcom.nec.webotx.webservice.http.dump=true
```

静的スタブ形式以外でのダンプメッセージの取得はできません。他の形式のクライアントのダンプメッセージを取得するには WebOTX Developer の TCP モニタを利用するか、他の HTTP プロトコル解析ツールをご利用ください。

ダンプメッセージの出力先は System.out になります。通常はコマンドプロンプトやシェルスクリプトのコンソール画面などに出力されます。

出力例：Hello World!を返す Web サービスの出力

```
******************
**** Request   ****
Content-Type: text/xml; charset="utf-8"
Content-Length: 422
SOAPAction: ""
User-Agent: "WebOTX WebService/x.xx.xx.xx"

<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns0="http://com.nec.webotx.test">
<env:Body>
<ns0:echoString_TestClass>
<String_1>Hello World!</String_1>
</ns0:echoString_TestClass>
</env:Body>
</env:Envelope>
```

```
**** Response ****
HTTP/1.1 200 OK
Date: Mon, 10 Dec 2007 02:03:29 GMT
Server: WebOTX_Web_Server/2.0.61 (Win32) Webserver_Plugin/1.2.21
X-Powered-By: Servlet/2.4
SOAPAction: ""
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/xml;charset=utf-8

<env:Envelope xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns0="http://com.nec.webotx.test"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<env:Body>
<ns0:echoString_TestClassResponse>
<result>Hello World!</result>
</ns0:echoString_TestClassResponse>
</env:Body>
</env:Envelope>
******************

Hello World!
```

※マニュアルのレイアウトにあわせるため、文中で適宜改行を入れています。

●サーバ側での送受信メッセージのダンプ

サーバ側のダンプメッセージはサーバ内のログレベルを変更することでログファイル内に出力させることが可能です。ログは webotx_agent.log に出力されます。

ダンプメッセージを出力させるには、出力したいドメインのログコンポーネント「javax」を CONFIG から DEBUG に変更してください。

ログレベルを変更するには統合運用管理ツール(Administrator)を利用するか、otxadmin コマンドを利用する、または WebOTX ドメインの config ディレクトリにある log4otx.xml に対して直接編集を行って下さい。

log4otx.xml での javax の設定内容

```
<logger name="javax">
        <level class="com.nec.webotx.logging.OTXLogLevel" value="CONFIG" />
        <appender-ref ref="FILELOG" />
</logger>
```

ログレベルの変更の手順については WebOTX マニュアルの運用編にある
「WebOTX 運用編(ロギング)」-「2. ロギングについて」-「2.3.1. ログの設定方法」
を参照してください。

サーバ側ダンプメッセージは Web アプリケーション形式でのみ出力ができます。EJB アプリケーション形式では出力できません。

サーバ側ダンプメッセージはドメイン全体で変更されます。配備したアプリケーションごと、などの切り替えはできませんので注意してください。

ログが正常に出力されると、ファイル内で以下のように出力されます。

出力例：Hello World!を返す Web サービスの出力

```
2007-12-10 11:03:29,871 DEBUG javax.enterprise.resource.webservices.rpc.server.http -
SOAP request message
*************************
```

```
content-type: text/xml; charset="utf-8"
content-length: 422
SOAPAction: ""
user-agent: "WebOTX WebService/x.xx.xx.xx"
host: localhost
accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
connection: keep-alive
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns0="http://com.nec.webotx.test"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<env:Body>
<ns0:echoString_TestClass>
<String_1>Hello World!</String_1>
</ns0:echoString_TestClass>
</env:Body>
</env:Envelope>
**************************
 [TP-Processor3]

・・・中略・・・・

2007-12-10 11:03:29,871 DEBUG javax.enterprise.resource.webservices.rpc.server.http -
SOAP response message
**************************
SOAPAction: ""
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns0="http://com.nec.webotx.test">
<env:Body>
<ns0:echoString_TestClassResponse>
<result>Hello World!</result>
</ns0:echoString_TestClassResponse>
</env:Body>
</env:Envelope>
**************************
```

※マニュアルのレイアウトにあわせるため、文中で適宜改行を入れています。

リクエスト、レスポンスそれぞれについて出力がされます。サーバ側ログはログのフォーマットにより、出力の日時も表示されます。

本機能は開発時や障害時のメッセージ確認用の機能です。機能を有効にしたままシステムを稼動させると大量のログが出力され、システムのパフォーマンスに影響が出ることがありますので必要なとき以外は有効にしないでください。

2.1.3.WSDLからWebサービスを作成する

JAX-RPC の Web サービスを WSDL から wscompile コマンドを使用して作成する方法を説明します。

準備

任意の空のフォルダ（ディレクトリ）を作成して WSDL を置き、環境変数に次の変数を追加します。

PATH 変数に追加する

- <J2SE_DIR>¥jre¥bin

CLASSPATH 変数に追加する

- <WebOTX_DIR>¥lib¥j2ee.jar
- <WebOTX_DIR>¥lib¥jaxrpc-impl.jar
- <WebOTX_DIR>¥lib¥saaj-impl.jar
- <WebOTX_DIR>¥lib¥endorsed¥resolver.jar
- <WebOTX_DIR>¥lib¥endorsed¥serializer.jar
- <WebOTX_DIR>¥lib¥endorsed¥xalan.jar
- <WebOTX_DIR>¥lib¥endorsed¥xml-apis.jar
- <WebOTX_DIR>¥lib¥endorsed¥xercesImpl.jar
- <J2SE_DIR>¥lib¥tools.jar

MEMO

「¥」は Windows のパス区切り文字です。UNIX では「/」に読み替えてください。

MEMO

<WebOTX_DIR>は WebOTX のインストールルートフォルダ（ディレクトリ）です。

MEMO

<J2SE_DIR>は J2SE SDK のインストールルートフォルダ（ディレクトリ）です。

設定ファイルの作成

先に作ったプロジェクトのルートフォルダ、または任意のフォルダに次の内容の XML ファイルを作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location="WSDL ファイル名" packageName="生成するされるクラスのパッケージ名" />
</configuration>
```

wscompileコマンドの実行

次のように wscompile コマンドを実行し、WSDL から Web サービス作成に必要なクラスを生成します。wscompile コマンドオプションの詳細については「コマンドリファレンス」を参照してください。

```
> java com.nec.webotx.webservice.xml.rpc.tools.wscompile.Main -gen:server -d クラスファイル出力先 -s ソースコード出力先 -keep -mapping WSEE マッピングファイル名 設定ファイル名
```

ビジネスロジック/EJBの作成

Web アプリケーションとして Web サービスを実装するときのビジネスロジックは、サービスエンドポイントインタフェースを implements することで作成します。サービスエンドポイントインタフェースは、wscompile コマンドにより生成されたクラスに含まれており、java.rmi.Remote を継承したインタフェースクラスです。

EJB として Web サービスを実装するときのビジネスロジックは、EJB です。EJB は「javax.ejb.SessionBean」を implements したクラスです。EJB の中にサービスエンドポイントインタフェースで定義されたメソッドを実装することで、Web サービスのビジネスロジックは完成します。

Webアプリケーションとして実装する場合の配備記述子

web.xml を作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" version="2.4">
<servlet>
  <servlet-name>任意のサーブレット名</servlet-name>
  <servlet-class>ビジネスロジックのクラス名（パッケージ名つき）</servlet-class>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>任意のサーブレット名</servlet-name>
  <url-pattern>/ビジネスロジックに関連付ける URL</url-pattern>
</servlet-mapping>
</web-app>
```

servlet-name 要素では任意のサーブレット名を記述します。servlet 要素と servlet-mapping 要素にそれぞれ servlet-name 要素がありますが、サーブレット名は同じにしなければなりません。

url-pattern 要素では、WSDL の address 要素の location 属性に指定した URL の一部を記述します。必ず「/」から書き始めることに注意してください。また、コンテキストルートではないことに注意してください。「/*」を指定することは他のサーブレットとの共存とセキュリティの観点から推奨されません。

（例 1）http://host/a/b の場合「/b」と記述します。

（例 2）http://host/a/b/c の場合「/b/c」と記述します。

次に、webservices.xml を作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<webservicesxmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd" version="1.1">
<webservice-description>
  <webservice-description-name>Web サービス名（WSDL を参照）</webservice-description-name>
  <wsdl-file>WSDL ファイルの URL（相対パス）</wsdl-file>
  <jaxrpc-mapping-file>WSEE マッピングファイルの URL（相対パス）</jaxrpc-mapping-file>
  <port-component>
    <port-component-name>任意のポートコンポーネント名</port-component-name>
    <wsdl-port xmlns:ns="ポートの名前空間 URI">ns:ポート名</wsdl-port>
    <service-endpoint-interface>サービスエンドポイントクラス名</service-endpoint-interface>
    <service-impl-bean>
      <servlet-link>任意のサーブレット名</servlet-link>
    </service-impl-bean>
  </port-component>
</webservice-description>
</webservices>
```

webservice-description-name 要素には WSDL で指定した Web サービス名を記述します。1 つの WSDL に複数の Web サービスが定義されたとき、webservices 要素には複数の webservice-description 要素を記述することができます。このとき、webservice-description-name 要素の値はユニークでなければなりません。一般的には、間違いを防止するために、webservice-description-name 要素には WSDL の service 要素の name 属性の値を指定します。

wsdl-file 要素には参照する WSDL のロケーションを WAR ファイルのルートから始まる相対パスで「WEB-INF/wsdl/sample.wsdl」のように指定します。一般的には、WSDL は他の配備記述子と同じ扱いをするため、WEB-INF フォルダの配下に置きます。

jaxrpc-mapping-file 要素には参照する WSEE マッピングファイルのロケーションを WAR ファイルのルートから始まる相対パスで「WEB-INF/mapping.xml」のように指定します。

port-component 要素には、1 つのポートについて、WSDL に指定された port に対するサービスエンドポイントインタフェースとビジネスロジックを関連付けます。port-component-name 要素には任意の「port 名」を指定します。1 つの WSDL に複数の port が定義されたとき、webservice-description 要素には複数の port-component 要素を記述することができます。このとき、この値がユニークになるように注意しなければなりません。一般的には、間違いを防止するために、WSDL の port 要素の name 属性の値を指定します。

wsdl-port 要素には、WSDL で指定されている port を名前空間 URI とローカル名で示します。xmlns: ns 属性の値に名前空間 URI を指定します。この値は、WSDL の targetNamespace に定義されています。また、ローカル名を「ns:ローカル名」のように指定します。この値は、WSDL の port 要素の name 属性の値として定義されています。

service-endpoint-interface 要素にはサービスエンドポイントインタフェースのクラス名（パッケージ名つき）を指定します。

service-impl-bean 要素では Web サービスのビジネスロジックを関連付けます。ここではサーブレットに関連付けるために、「servlet-link」という要素を使い、servlet-link 要素に、web.xml の servlet-name 要素で

指定したサーブレット名を記述します。

EJBとして実装する場合の配備記述子

ejb-jar.xml を作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd" version="2.1">
  <description xml:lang="ja">任意の EJB アーカイブ名</description>
  <enterprise-beans>
    <session>
      <ejb-name>任意の EJB 名</ejb-name>
      <service-endpoint>サービスエンドポイントインタフェースクラス名</service-endpoint>
      <ejb-class>EJB クラス名</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <security-identity>
        <use-caller-identity/>
      </security-identity>
    </session>
  </enterprise-beans>
</ejb-jar>
```

service-encpoint 要素には、サービスエンドポイントインタフェースのクラス名をパッケージ名つきで記述します。

ejb-class 要素には、EJB クラス名をパッケージ名つきで記述します。

次に、webservices.xml を作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<webservices xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd" version="1.1">
<webservice-description>
  <webservice-description-name>Web サービス名（WSDL を参照）</webservice-description-name>
  <wsdl-file>WSDL ファイルの URL（相対パス）</wsdl-file>
  <jaxrpc-mapping-file>WSEE マッピングファイルの URL（相対パス）</jaxrpc-mapping-file>
  <port-component>
    <port-component-name>任意のポートコンポーネント名</port-component-name>
    <wsdl-port xmlns:ns="ポートの名前空間 URI">ns:ポート名</wsdl-port>
    <service-endpoint-interface>サービスエンドポイントクラス名</service-endpoint-interface>
    <service-impl-bean>
      <ejb-link>任意の EJB 名</ejb-link>
    </service-impl-bean>
    </port-component>
  </webservice-description>
</webservices>
```

webservice-description-name 要素には WSDL で指定した Web サービス名を記述します。1 つの WSDL に複数の Web サービスが定義されたとき、webservices 要素には複数の webservice-description 要素を記述することができます。このとき、webservice-description-name 要素の値はユニークでなければなりません。一般的には、間違いを防止するために、webservice-description-name 要素には WSDL の service 要素の name 属性の値を指定します。

wsdl-file 要素には参照する WSDL のロケーションを WAR ファイルのルートから始まる相対パスで「META-INF/wsdl/sample.wsdl」のように指定します。一般的には、WSDL は他の配備記述子と同じ扱いをするため、META -INF フォルダの配下に置きます。

jaxrpc-mapping-file 要素には参照する WSEE マッピングファイルのロケーションを EJB-JAR ファイルのルートから始まる相対パスで「mapping.xml」のように指定します。

port-component 要素には、1 つのポートについて、WSDL に指定された port に対するサービスエンドポイ

ントインタフェースとビジネスロジックを関連付けます。port-component-name 要素には任意の「port 名」を指定します。1 つの WSDL に複数の port が定義されたとき、webservice-description 要素には複数の port-component 要素を記述することができます。このとき、この値がユニークになるように注意しなければなりません。一般的には、間違いを防止するために、WSDL の port 要素の name 属性の値を指定します。

wsdl-port 要素には、WSDL で指定されている port を名前空間 URI とローカル名で示します。xmlns: ns 属性の値に名前空間 URI を指定します。この値は、WSDL の targetNamespace に定義されています。また、ローカル名を「ns:ローカル名」のように指定します。この値は、WSDL の port 要素の name 属性の値として定義されています。

service-endpoint-interface 要素にはサービスエンドポイントインタフェースのクラス名（パッケージ名つき）を指定します。

service-impl-bean 要素では Web サービスのビジネスロジックを関連付けます。ここでは EJB に関連付けるために、「ejb-link」という要素を使い、ejb-link 要素に、ejb-jar.xml の ejb-name 要素で指定した EJB 名を記述します。

アーカイブの作成

JDK の jar コマンドを使用して WAR ファイル、または EJB-JAR ファイルを作成します。アーカイブ内部のイメージは次のようになります。

WAR ファイル

/WEB-INF/classes/<wscompile コマンドで生成したクラス、ビジネスロジック>
/WEB-INF/wsdl/<WSDL ファイル>
/WEB-INF/web.xml
/WEB-INF/webservices.xml
/WEB-INF/mapping.xml

EJB-JAR ファイル

/<wscompile コマンドで生成したクラス、EJB>
/META-INF/wsdl/<WSDL ファイル>
/META-INF/ejb-jar.xml
/META-INF/webservices.xml
/mapping.xml

配備

配備には、統合運用管理ツール、Web 版運用管理ツール、運用管理パースペクティブ、otxadmin コマンドが使えます。各ツールの詳細は、運用編の各ツールの説明を参照してください。

なお、配備の際、コンテキストルート（WSDL の address 要素の location 属性に指定した URL の一部）を指定する必要があります。「/」から書き始めても、そうでなくてもかまいません。

（例 1）http://host/a/b の場合「a」と指定します。

（例 2）http://host/a/b/c の場合「a」と指定します。

2.1.4.SOAPメッセージを直接処理するWebサービスの開発

SOAP メッセージには任意の XML 文書を封入したり、添付ファイルをつけることもできます。ここでは、SOAP メッセージを直接処理するサーブレットを作ることで、任意の XML 文書を処理する Web サービスを実現する方法について説明します。文中で「SAAJ」サンプルを参照する箇所がありますので、あわせて「SAAJ」サンプルをご覧ください。

インタフェースを決める

まず、交換する XML 文書の書式と Web サービスのインタフェースを決めます。

XMLスキーマの作成

XML スキーマを使って交換する XML 文書の書式を定義します。Web サービスが受け取る XML と返却する XML の書式が別々の場合には、それぞれについて定義します。

WSDLの作成

Web サービスのインタフェースである **WSDL** を作成します。先に作成しておいた XML スキーマを **types 要素**の中で **import** して使用します。

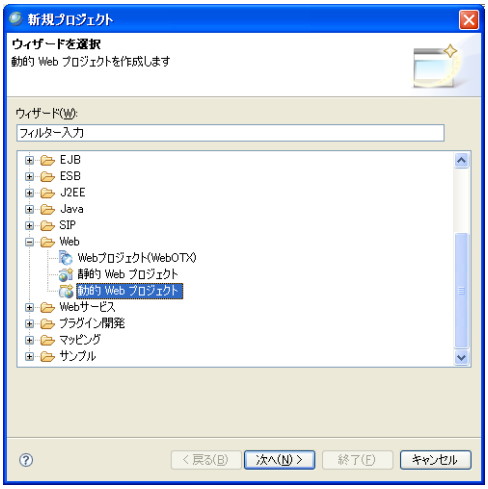
プロジェクトを作成する

HTTP+SOAP メッセージを処理するサーブレットを作成するために、Web プロジェクトを作成します。

新規プロジェクトダイアログ

WebOTX Developer のメニューから、**ファイル｜新規｜プロジェクト**を選択すると、**新規プロジェクト**ダイアログが起動します。

Web｜動的 Web プロジェクトを選択し、[**次へ**]ボタンを押します。



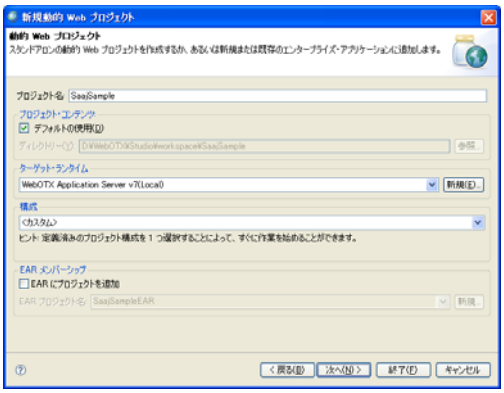
新規動的Webプロジェクト

「**プロジェクト名**」に動的 Web プロジェクトの名前を指定します。プロジェクトは、デフォルトでは

<WebOTX_DIR>¥Studio¥workspace¥指定したプロジェクト名

というフォルダに作成されます。もし、別の場所に作成したい場合は、プロジェクトコンテンツの「**デフォルトの使用**」のチェックをはずし、任意の場所を絶対パスで指定します。

ターゲットランタイムは **WebOTX Application Serverv7(local)**に設定します。まだ、サーバの登録をしていない場合は、[新規]ボタンを押し、「新規サーバランタイム」ダイアログに移ります。ここで **WebOTX Application Serverv7(local)**を選択し、[次へ]ボタンを押します。



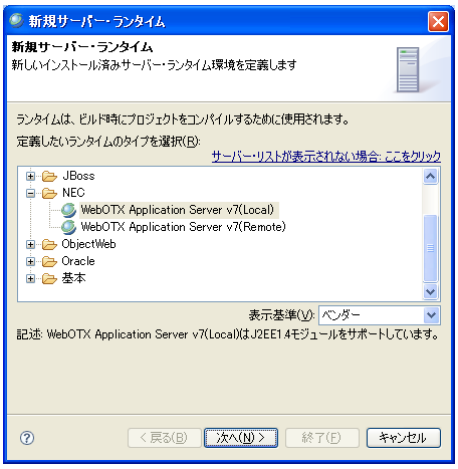
すると WebOTX サーバのインストールパスが空欄になっているので、設定し、[**終了**]ボタン

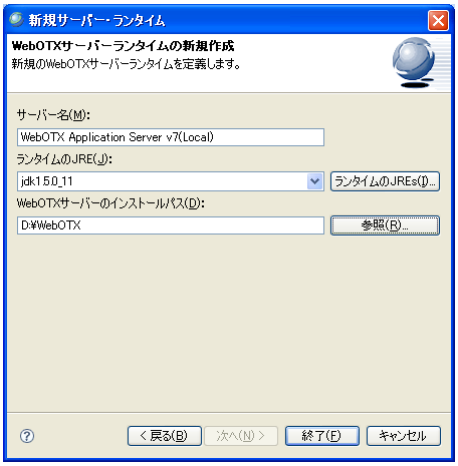
を押します。

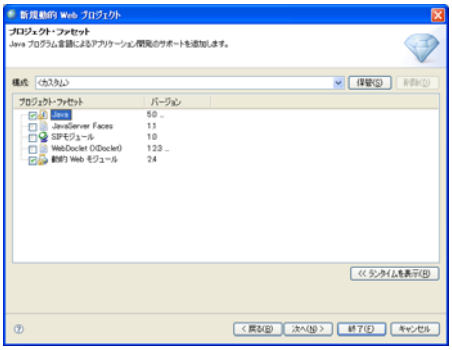
プロジェクトファセットは特に設定する必要はありません。[**次へ**]ボタンを押してください。

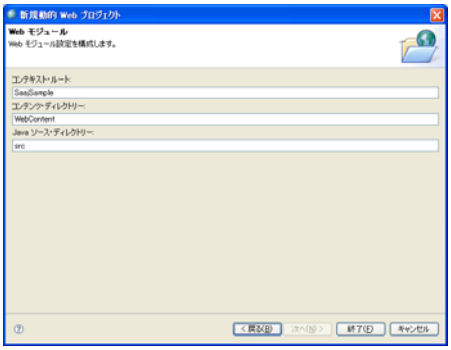
Web モジュール画面も特に設定する必要はありませんが、コンテキストルートなどが設定できます。最後に[**終了**]ボタンを押してください。

これで動的 Web プロジェクトが作成されます。









Webプロジェクトの設定をする

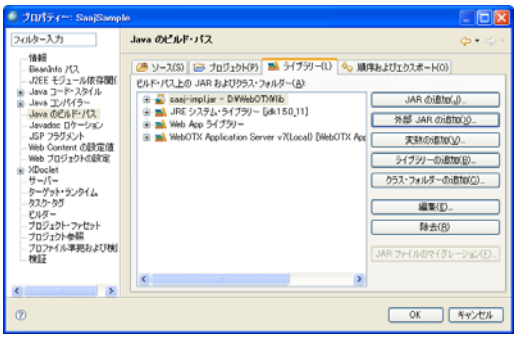
サーブレットの中で SOAP メッセージを処理するために、SAAJ の API を使用します。そのため、Web プロジェクトの設定をして、SAAJ のライブラリをクラスパスに追加します。

プロパティ – Javaのビルドパス

パッケージエクスプローラで作成したプロジェクト上で右クリックし、**プロパティ**を選択します。プロジェクトのプロパティ画面が表示されたら、左のリストから **Java のビルドパス**を選択します。次に、右側の**ライブラリー**タブを選択し、[**外部 JAR の追加**]ボタンを押します。次のファイルを指定します。

<WebOTX_DIR>¥lib¥saaj-impl.jar

ビルドパス上の JAR およびクラス・フォルダのリストに **saaj-impl.jar** が追加されたのを確認し、[**OK**]ボタンを押します。



サーブレットを作成する

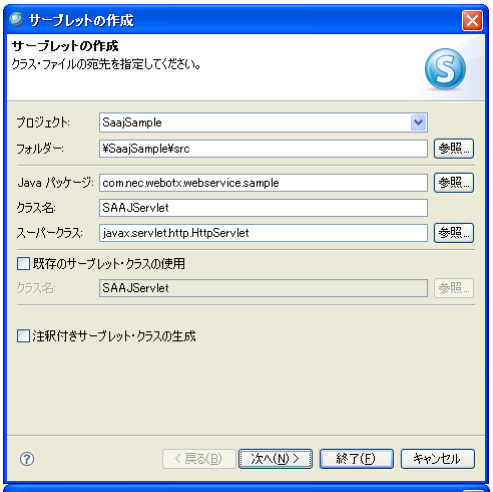
サーブレットウィザードを使って Web プロジェクトに HTTP+SOAP メッセージを処理するサーブレットを追加します。

サーブレットの作成ウィザード

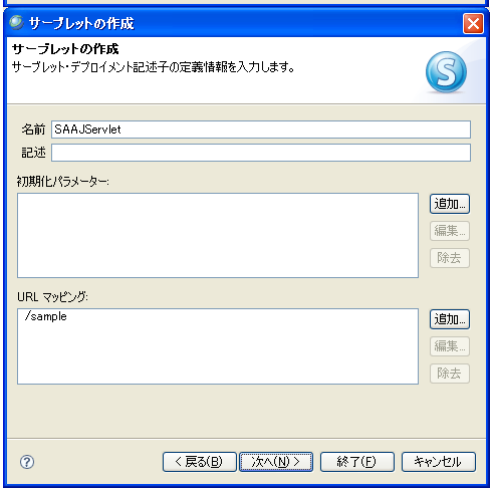
作成した動的 Web プロジェクトで右クリックし、、**新規 | サーブレット**を実行します。

プロジェクト名、**Java パッケージ名**、**クラス名**を設定します。ここでは順に、「SaajSample」、「com.nec.webotx.webservice.sample」、「SAAJServlet」を設定しています。

[**次へ**]ボタンを押します。

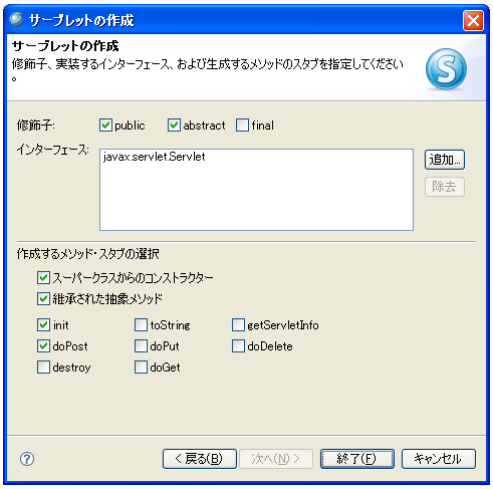


「名前」はサーブレット名です。「SAAJService」に変更します。URL マッピングは「sample」に変更します。URL マッピングの[**編集**]ボタンを押して変更します。



public と **abstract** を選択します。

また、メソッドは **init** と **doPost** を選択し、[終了] ボタンを押します。



生成されたサーブレットクラスの編集

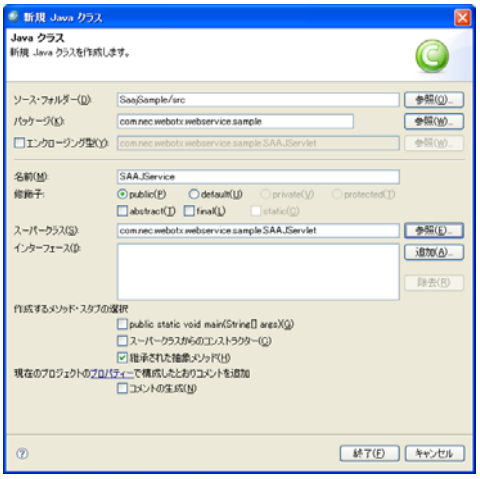
生成されたサーブレットクラスにソースコードを追加します。追加内容は、SAAJ サンプルの「**SAAJServlet**」クラスを参照してください。doPost メソッドでリクエストメッセージから SAAJ の SOAPMessage オブジェクトを生成したり、SOAPMessage オブジェクトからレスポンスメッセージを生成する処理を行っている汎用のクラスです。

SOAPメッセージ処理クラスの追加

サーブレットクラスの中で、doPost メソッド内で取り出した SOAPMessage を処理するメソッドとして **onMessage** メソッドを用意しています。この onMessage メソッドを実装することで、SOAP メッセージに封入された XML 文書を処理することができます。onMessage メソッドを実装するには、作っておいたサーブレットクラスを継承したクラスを作成し、onMessage メソッドをオーバライドします。サーブレットクラスを継承したクラスは次のように作成します。

メニューから、**ファイル**｜**新規**｜**クラス**を選択し、**名前**、**スーパークラス**を指定し、[**終了**]ボタンを押します。ソース・フォルダやパッケージは SaajSample プロジェクトを選択した状態でこのダイアログを開くと自動的に設定されていますが、変更もできます。なお、スーパークラスは、あらかじめ作っておいたサーブレットクラスを指定します。

※ここで作成したものに値するサンプルのクラスは「SAAJService」になります。



WARファイルを作成する

作成したサーブレットを WAR 形式でアーカイブし、サーバに配備できる状態にします。

Webプロジェクトへのファイル追加

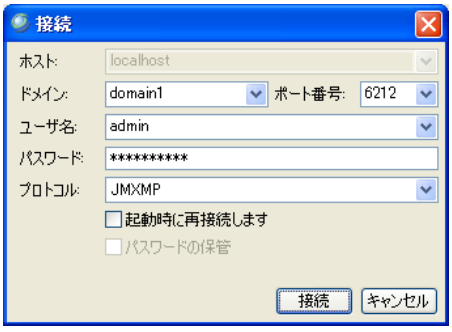
Web サービスのインタフェースを公開する場合、WSDL と XML スキーマを Web プロジェクトのコンテンツディレクトリに追加しておくと便利です。

エクスポート

メニューから、**ファイル**｜**エクスポート**を選び、選択画面で「**Web**｜**WAR ファイル（WebOTX）**」を選択し、[**次へ**]ボタンを押します。プロジェクトはここで作った Web プロジェクトを、出力ファイルに出力先を指定し、[**終了**]ボタンを押します。

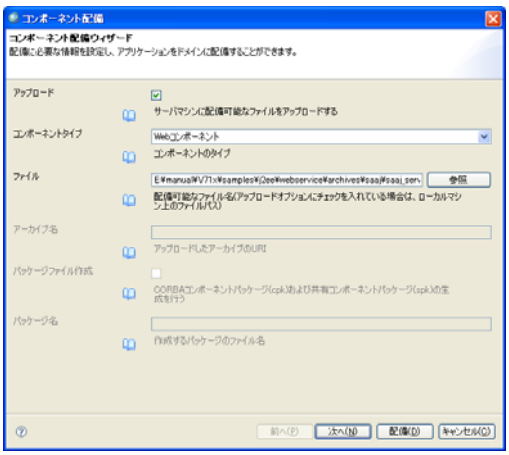
WARファイルを配備する

ウィンドウ｜**パースペクティブを開く**｜**その他**を実行し、リストから「**WebOTX 運用管理ツール**」を指定し[**OK**]を押します。接続ダイアログで設定をし、ドメインに接続します。



ドメインのツリーのアプリケーションで右クリックし、「**コンポーネントの配備**」を実行します。

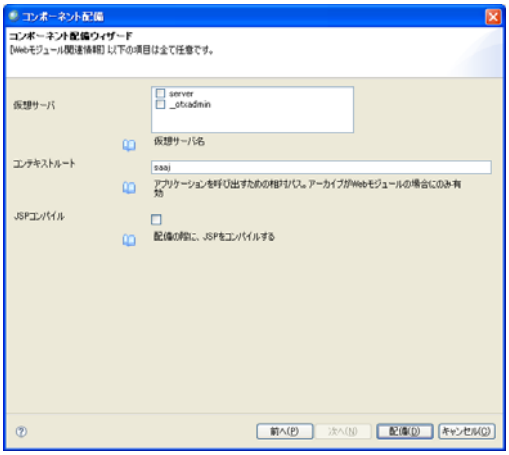
コンポーネントタイプで「Web コンポーネント」を選択し、ファイルに作成した WAR ファイルを指定し、[**次へ**]ボタンを **2 回**押します。



コンテキストルートを指定します。コンテキストルートは Web サービスの URL に次のように影響します。

http://<ホスト名>:<ポート番号>/<**コンテキストルート**>/<**サーブレット作成時に指定した URL**>

最後に[**配備**]ボタンを押し、配備を行います。



2.1.5.Webサービスの公開

WSDLを公開したり、レジストリサービスにWebサービスを登録することによって、Webサービスを公開する方法について説明します。

WSDLを公開する

Webサービスを公開する上でまず初めにやることは、インタフェースの公開、つまりWSDLの公開です。

ランタイムのWSDL公開機能

Webサービス作成ウィザードを利用してRPCのWebサービスを作成した場合、Webサービスの配備を行うと自動的にWSDLが公開されるようになっています。これは、JAX-RPC仕様に定められた機能であり、ランタイムにあらかじめ組み込まれている機能によって実現されています。このWSDLのURLは次のようになります。

http://Webサービスの URL/?wsdl

ファイルを公開する

SOAPメッセージを直接処理するWebサービスを作成した場合や、相手によってWSDLを使い分けるなどの目的で公開するWSDLを厳密に管理したい場合は、WSDLファイルをWebサーバやWebコンテナなどに置くことによって公開してください。

レジストリサービスにWebサービスを登録する

UDDIレジストリなどのWebサービスが登録されているレジストリサービスに、自分で作成し公開するWebサービスを登録する方法について説明します。

JAXRインタフェースを利用する

JAXRは、UDDI、ebXMLなどのレジストリサービスに対して、同じインタフェースを用いてWebサービスの登録をすることができます。ここでは、JAXRインタフェースを用いてUDDI 2.0対応レジストリに登録を行う「JAXR_Publish」サンプルについて説明します。プロパティや登録内容などに使用する各値は、それぞれプロパティファイルから取得します。

```
//フィールドではコネクションを宣言します。
Connection connection = null;
```

```
//登録を実行するmainメソッドです。
public static void main(String[] args) {
    //プロパティファイルの設定内容を取得します。
    ResourceBundle bundle =
      ResourceBundle.getBundle("com.nec.webotx.webservice.sample.JAXRSample");
    //検索用URLを取得します。
    String queryURL = bundle.getString("query.url");
    //登録用URLを取得します。
    String publishURL = bundle.getString("publish.url");
    //ユーザ名を取得します。
    String username = bundle.getString("registry.username");
    //パスワードを取得します。
    String password = bundle.getString("registry.password");

    JAXRPublishSample sample = new JAXRPublishSample();
    //コネクションを張ります。
    sample.makeConnection(queryURL, publishURL);
    //登録を実行します。
    sample.executePublish(username, password);
}
```

```
//コネクションを張るためのメソッドです。
public void makeConnection(String queryURL, String publishURL) {
    //プロパティファイルの設定内容を取得します。
    ResourceBundle bundle =
      ResourceBundle.getBundle("com.nec.webotx.webservice.sample.JAXRSample");
```

```
        //Factory クラスを設定します。
        String factoryClass = "com.sun.xml.registry.uddi.ConnectionFactoryImpl";
        //各種プロパティ値の設定
        Properties props = new Properties();
        props.setProperty("javax.xml.registry.queryManagerURL", queryURL);//検索用 URL
        props.setProperty("javax.xml.registry.lifeCycleManagerURL", publishURL);//登録用 URL
        props.setProperty("javax.xml.registry.factoryClass", factoryClass);//ファクトリクラス
        String uuidString = bundle.getString("postal.uuidString");
        props.setProperty("javax.xml.registry.postalAddressScheme",  uuidString);//住所
        props.setProperty("javax.xml.registry.semanticEquivalences",
                          "urn:uuid:PostalAddressAttributes/StreetNumber," +
                          "urn:" + uuidString + "/MyStreetNumber|" +
                          "urn:uuid:PostalAddressAttributes/City," +
                          "urn:" + uuidString + "/MyCity|" +
                          "urn:uuid:PostalAddressAttributes/State," +
                          "urn:" + uuidString + "/MyState|" +
                          "urn:uuid:PostalAddressAttributes/Country," +
                          "urn:" + uuidString + "/MyCountry");

        //実際にコネクションを張ります。
        try {
            ConnectionFactory cf = ConnectionFactory.newInstance();
            cf.setProperties(props);
            connection = cf.createConnection();
        } catch (Exception e) {
            e.printStackTrace();
        }
}
```

```
Public void executePublish(String username, String password) {
        RegistryService rs = null;
        BusinessLifeCycleManager blcm = null;

        BusinessQueryManager bqm = null;
        ResourceBundle bundle =
            ResourceBundle.getBundle("com.nec.webotx.webservice.sample.JAXRSample");
        try {
            rs = connection.getRegistryService();
            blcm = rs.getBusinessLifeCycleManager();
            bqm = rs.getBusinessQueryManager();
        //login
        PasswordAuthentication passwdAuth =
            new PasswordAuthentication(username, password.toCharArray());
        Set creds = new HashSet();
        creds.add(passwdAuth);
        connection.setCredentials(creds);
```

```
// executePublish メソッドのつづき

        /*
         * Concept を登録します。Concept とは UDDI では tModel になります。
         */

        //Concept オブジェクトを生成します。
        Concept specConcept =
            blcm.createConcept(null, bundle.getString("concept.name"), "");
        //Concept についての説明を追加します。
        InternationalString is =
            blcm.createInternationalString(
            bundle.getString("concept.description"));
```

```
        specConcept.setDescription(is);
        //WSDL の参照先を追加します。
        ExternalLink wsdlLink =
              blcm.createExternalLink(
                    bundle.getString("link.uri"),
                    bundle.getString("link.description"));
        specConcept.addExternalLink(wsdlLink);

        String schemeName = "uddi-org:types";
        ClassificationScheme uddiOrgTypes =
        bqm.findClassificationSchemeByName(null, schemeName);

        Classification wsdlSpecClassification =
              blcm.createClassification(uddiOrgTypes, "wsdlSpec", "wsdlSpec");
        specConcept.addClassification(wsdlSpecClassification);

        Collection concepts = new ArrayList();
        concepts.add(specConcept);
        //登録を実行します。
        BulkResponse response = blcm.saveConcepts(concepts);

        Collection exceptions = response.getExceptions();
        javax.xml.registry.infomodel.Key concKey = null;
        String uuidString = null;
        if (exceptions == null) {
        //レスポンスから登録時に割り振られた Concept の ID を取得し、保持します。
        Collection keys = response.getCollection();
        Iterator keyIter = keys.iterator();
              if (keyIter.hasNext()) {
                    concKey = (javax.xml.registry.infomodel.Key) keyIter.next();
                    //IDを取得し、保持する。
                    uuidString = concKey.getId();
                    System.out.println("save concept successful");
              }
        } else {
              Iterator excIter = exceptions.iterator();
              Exception exception = null;
              while (excIter.hasNext()) {
                    exception = (Exception) excIter.next();
                    System.err.println("Exception on save: " + exception.toString());
              }
        }
```

```
//makeConnection メソッドのつづき

        /*
         * Organization を登録します。
         */

        //Organization オブジェクトを生成します。このオブジェクトに
        //登録内容を追加していきます。
        //生成時には、企業などの機構名を追加します。
        Organization org = blcm.createOrganization(bundle.getString("org.name"));
        //この機構についての説明を追加します。
        InternationalString s = blcm.createInternationalString(bundle.getString("org.description"));
        org.setDescription(s);

        //連絡先の担当者名や役職を追加します。
        User primaryContact = blcm.createUser();
        PersonName pName = blcm.createPersonName(bundle.getString("person.name"));
```

```
primaryContact.setPersonName(pName);

//連絡先となる電話番号を追加します。
TelephoneNumber tNum = blcm.createTelephoneNumber();
tNum.setNumber(bundle.getString("phone.number"));
Collection phoneNums = new ArrayList();
phoneNums.add(tNum);
primaryContact.setTelephoneNumbers(phoneNums);

//住所を追加します。
PostalAddress address = blcm.createPostalAddress(
        bundle.getString("postal.address1"),
        bundle.getString("postal.address2"),
        bundle.getString("postal.address3"),
        bundle.getString("postal.address4"),
        bundle.getString("postal.address5"),
        bundle.getString("postal.address6"),
        bundle.getString("postal.address7"));
 Collection postalAddresses = new ArrayList();
 postalAddresses.add(address);
 primaryContact.setPostalAddresses(postalAddresses);

 //e-mail アドレスを追加します。
 EmailAddress emailAddress =
        blcm.createEmailAddress(bundle.getString("email.address"));
        Collection emailAddresses = new ArrayList();
 emailAddresses.add(emailAddress);
 primaryContact.setEmailAddresses(emailAddresses);

 //Organization に追加します。
 org.setPrimaryContact(primaryContact);

 //企業の分類を追加します。
 ClassificationScheme cScheme = blcm.createClassificationScheme(
        blcm.createInternationalString(bundle.getString("classification.scheme")),
        blcm.createInternationalString(""));
 javax.xml.registry.infomodel.Key cKey =
        (javax.xml.registry.infomodel.Key) blcm.createKey(bundle.getString("schema.key"));
                cScheme.setKey(cKey);
 Classification classification = blcm.createClassification(cScheme,
        bundle.getString("classification.name"),
        bundle.getString("classification.value"));
 org.addClassification(classification);

 ClassificationScheme Scheme1 = blcm.createClassificationScheme(
        blcm.createInternationalString(bundle.getString("classification.scheme1")),
        blcm.createInternationalString(""));
 javax.xml.registry.infomodel.Key cKey1 =
        (javax.xml.registry.infomodel.Key) blcm.createKey(bundle.getString("schema.key1"));
        cScheme1.setKey(cKey1);
 classification = blcm.createClassification(cScheme,
        bundle.getString("classification.name1"),
        bundle.getString("classification.value1"));
 org.addClassification(classification);

 //企業の識別情報を追加します。
 ClassificationScheme cScheme2 = blcm.createClassificationScheme(
        blcm.createInternationalString(bundle.getString("classification.scheme2")),
        blcm.createInternationalString());
 javax.xml.registry.infomodel.Key cKey2 =
        (javax.xml.registry.infomodel.Key) blcm.createKey(bundle.getString("schema.key2"));
```

```
        cScheme2.setKey(cKey2);

    ExternalIdentifier ei = blcm.createExternalIdentifier(cScheme2,
        bundle.getString("classification.name2"),
        bundle.getString("classification.value2"));
    org.addExternalIdentifier(ei);

    //サービスの情報を追加していきます。
    Collection services = new ArrayList();
    //サービス名を追加します。
    Service service = blcm.createService(bundle.getString("service.name"));
    //サービスについての説明を追加します。
    service.setDescription
        (blcm.createInternationalString(bundle.getString("service.description")));

    //サービスを利用するための技術的な情報についての説明を追加します。
    Collection serviceBindings = new ArrayList();
    ServiceBinding binding = blcm.createServiceBinding();
    is = blcm.createInternationalString(bundle.getString("svcbinding.description"));
    binding.setDescription(is);

    //架空のサービスエンドポイント URL の登録を可能にします。
    binding.setValidateURI(false);
    //サービスエンドポイント URL を追加します。
    binding.setAccessURI(bundle.getString("svcbinding.accessURI"));

    //tModelの参照先を追加します。このメソッドの最初に登録した Concept をレジストリから
    //取得します。
    Collection namePatterns = new ArrayList();
    namePatterns.add(bundle.getString("concept.name"));
    //検索を実行します。検索条件には、登録した Concept 名を使用します。
    BulkResponse br =
        bqm.findConcepts(null, namePatterns, null, null, null);
    //取得した Concept と ID を照らし合わせ、登録した Concept と同じ ID かどうか
    //確認します。
    Collection specConcepts = br.getCollection();
    Iterator iter = specConcepts.iterator();
    Concept retConcept = null;
    if (!iter.hasNext()) {
        System.out.println("No WSDL specification concepts found");
    } else {
        while (iter.hasNext()) {
            Concept testConcept = (Concept) iter.next();
            String testId = testConcept.getKey().getId();
        //ID が一致した Concept を一旦、保持します。
        if (testId.equals(uuidString)) {
            retConcept = testConcept;
            System.out.println("find concept Successful");
        }
    }
    //Concept から tModel の参照先を取得し、Organization に追加します。
    if (retConcept != null) {
        Collection links = retConcept.getExternalLinks();
        if (links.size() > 0) {
            ExternalLink link = (ExternalLink) links.iterator().next();
        }
        SpecificationLink specLink = blcm.createSpecificationLink();
        specLink.setSpecificationObject(retConcept);
        binding.addSpecificationLink(specLink);
    }
       serviceBindings.add(binding);
```

```
            service.addServiceBindings(serviceBindings);
            services.add(service);
            //Organization に追加します。
            org.addServices(services);

            //登録を実行します。
            Collection orgs = new ArrayList();
            orgs.add(org);
            BulkResponse response2 = blcm.saveOrganizations(orgs);
            //レスポンスに例外が含まれているかチェックします。例外がなければ
            //"Successful"と表示して終了します。例外がある場合は、その内容を
            //表示します。
            Collection exceptions = response2.getExceptions();
            if (exceptions == null) {
                System.out.println("Successful");
            } else {
                Iterator excIter = exceptions.iterator();
                Exception exception = null;
                while (excIter.hasNext()) {
                    exception = (Exception) excIter.next();
                    System.err.println(exception.toString());
                }
            }
        }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 最後にレジストリへのコネクションをクローズします。
            if (connection != null) {
                try {
                    connection.close();
                } catch (JAXRException je) {
                }
            }
        }
}
}
```

UDDI 3.0 対応レジストリに登録する

UDDI 3.0 対応レジストリに登録する方法については、「**WebOTX UDDI Registry アプリケーション開発ガイド**」を参照してください。

2.1.6.Webサービスを探す

レジストリサービスを検索し、Web サービスを探す方法について説明します。

JAXRインタフェースを利用する

JAXR は、UDDI、ebXML などのレジストリサービスに対して、同じインタフェースを用いて Web サービスを検索することができます。ここでは、JAXR インタフェースを用いて UDDI 2.0 対応レジストリを検索して Web サービスを探す「JAXR_Query」サンプルについて説明します。プロパティや登録内容などに使用する各値は、それぞれプロパティファイルから取得します。

```
//フィールドではコネクションを宣言します。
Connection connection = null;
```

```
//検索を実行する main メソッドです。
public static void main(String[] args) {
        //プロパティファイルの設定内容を取得します。
        ResourceBundle bundle =
              ResourceBundle.getBundle("com.nec.webotx.webservice.sample.JAXRSample");
        //検索用 URL を取得します。
        String queryURL = bundle.getString("query.url");
        JAXRQuerySample queryClient = new JAXRQuerySample();
        //コネクションを張ります。
        queryClient.makeConnection(queryURL);
        //検索を実行します。
        queryClient.execute(args[0]);
}
```

```
//コネクションを張るためのメソッドです。
public void makeConnection(String queryURL) {
        //プロパティファイルの設定内容を取得します。
        ResourceBundle bundle =
              ResourceBundle.getBundle("com.nec.webotx.webservice.sample.JAXRSample");
        //各種プロパティの設定です。
        Properties props = new Properties();
        props.setProperty("javax.xml.registry.queryManagerURL", queryURL);//検索用 URL
        //コネクションを張ります。
        try {
              ConnectionFactory factory = ConnectionFactory.newInstance();
               factory.setProperties(props);
               connection = factory.createConnection();

        } catch (Exception e) {
              e.printStackTrace();
        }
}
```

```
//検索処理を実装したメソッドです。
public void execute(String query) {
try {
               RegistryService rs = connection.getRegistryService();
               BusinessQueryManager bqm = rs.getBusinessQueryManager();
               //引数の文字列を検索条件に含めます。
               Collection patterns = new ArrayList();
               patterns.add("%" + query + "%");
               System.out.println("start");
               //検索を実行し、結果を取得します。
               BulkResponse br =
                     bqm.findOrganizations(null, patterns, null, null, null, null);
               Collection collection = br.getCollection();
               System.out.println("[RESULT]");
               //検索によって得られてた結果をコンソールに表示します。
```

```
Iterator iter = collection.iterator();
while (iter.hasNext()) {
        Iterator ite;
        Organization org = (Organization) iter.next();
        //機構名
        System.out.println("Org Name: " + org.getName().getValue());
        //機構についての説明
        System.out.println("Org Description: "+org.getDescription().getValue());
        //担当者
        System.out.println("PersonName: "
              +org.getPrimaryContact().getPersonName().getFullName());
        //電話番号
        Collection tel = org.getPrimaryContact().getTelephoneNumbers("");
        ite = tel.iterator();
        while(ite.hasNext()){
            TelephoneNumber number = (TelephoneNumber)ite.next();
            System.out.println("TelephoneNumber: "+number.getNumber());
        };
        //E-mail
        Collection email = org.getPrimaryContact().getEmailAddresses();
        ite = email.iterator();
        while(ite.hasNext()){
            EmailAddressImpl address = (EmailAddressImpl)ite.next();
            System.out.println("EmailAddresses: "+address.getAddress());
        }
        //分類(カテゴリ)
        Collection classifications = org.getClassifications();
        ite = classifications.iterator();
        while(ite.hasNext()){
             Classification classification = (Classification)ite.next();
             System.out.println("Classification Name: "
                       +classification.getName().getValue());
             System.out.println("Classification: "+classification.getValue());
        }
        //Web サービスの情報
        Collection services = org.getServices();
        ite = services.iterator();
        while (ite.hasNext()) {
            Service service = (Service) ite.next();
            //サービス名
            System.out.println(
                 "Service Name: " + service.getName().getValue());
            //サービスについての説明
            System.out.println( "Service Description: "
                          + service.getDescription().getValue());
            Collection serviceBindings = service.getServiceBindings();
            Iterator bindings = serviceBindings.iterator();
            while (bindings.hasNext()) {
                ServiceBinding binding = (ServiceBinding) bindings.next();
                //サービスエンドポイント URL
                      System.out.println( "ServiceEndpointURI: " +
                            binding.getAccessURI());
                Collection specLinks = binding.getSpecificationLinks();
                if (specLinks.size() > 0) {
                      SpecificationLink link =
                            (SpecificationLink) specLinks.iterator().next();
                      Concept concept =
                            (Concept) link.getSpecificationObject();
                      System.out.println("Concept name: "
                            + concept.getName().getValue());
                      Collection links = concept.getExternalLinks();
```

```
                                    if (links.size() > 0) {
                                        ExternalLink extlink = (ExternalLink) links.iterator().next();
                                        //WSDL の URI
                                        System.out.println("WSDL URI: " + extlink.getExternalURI());
                                    } else continue;
                                }
                            }
                        }
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    //最後にコネクションをクローズします。
                    try {
                        connection.close();
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            }
```

UDDI 3.0 対応レジストリを検索する

UDDI 3.0 対応レジストリを検索する方法については、「**WebOTX UDDI Registry アプリケーション開発ガイド**」を参照してください。

2.1.7.Webサービスを利用する

Web サービスのクライアントを作成して、Web サービスを利用する方法について説明します。

JAX-RPC – スタティック(静的)スタブ方式

WSDL をコンパイルしてサービスエンドポイントインタフェースやスタブコードを生成し、それらを使ってクライアントを作成する方法です。この方法は、Web サービスのクライアントを作成する方法としては最も一般的なものです。ここでは、スタティックスタブ方式のクライアントを WebOTX Developer's Studio を活用して開発する方法を順に説明します。

Javaプロジェクトの作成

まず、Web サービスクライアントを作成するために、「**Java プロジェクト**」を作成します。Java プロジェクトの詳しい作成方法は、チュートリアルをご覧ください。ソース・フォルダに「**src**」を、出力フォルダに「**bin**」を指定してください。また、プロジェクトの**プロパティ**で、「**Java のビルドパス**」の「**ライブラリ**」タブにて[**外部 JAR の追加**]ボタンを押し、次の JAR ファイルを追加してください。

- <WebOTX_DIR>¥lib¥j2ee.jar
- <WebOTX_DIR>¥lib¥jaxrpc-impl.jar
- <WebOTX_DIR>¥lib¥saaj-impl.jar
- <WebOTX_DIR>¥lib¥endorsed¥resolver.jar
- <WebOTX_DIR>¥lib¥endorsed¥serializer.jar
- <WebOTX_DIR>¥lib¥endorsed¥xalan.jar
- <WebOTX_DIR>¥lib¥endorsed¥xml-apis.jar
- <WebOTX_DIR>¥lib¥endorsed¥xercesImpl.jar
- <J2SE_DIR>¥lib¥tools.jar

MEMO

<WebOTX_DIR>は WebOTX のインストールルートディレクトリ、<J2SE_DIR>は J2SE SDK のインストールルートディレクトリのことです。

Javaプロジェクトの準備

WSDL ファイルをプロジェクトのルートに置きます。

config.xmlファイルの作成

プロジェクトのルートに「**config.xml**」という名前でファイルを作成します。内容は、次のようにします。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location="Hello.wsdl" packageName="com.nec.webotx.webservice.sample" />
</configuration>
```

wsdl 要素にコンパイル対象の WSDL の情報を指定します。**location 属性**は、プロジェクトのルートに置いた WSDL のファイル名を指定します。**packageName 属性**は、WSDL コンパイラが生成するクラスが属するパッケージの名前を指定します。

wscompileコマンドの実行

メニューから、**実行** | **構成および実行**を選択し、実行ダイアログを開きます。**Java アプリケーション**を選択して[**新規**]ボタンを押して名前を指定し、新しい構成を作成します。「**メイン**」タブで、「**プロジェクト**」に作成しておいた Java プロジェクトの「**メインクラス**」に「**com.nec.webotx.webservice.xml.rpc.tools.wscompile.Main**」を指定します。「**メインクラスの検索時に外部 JAR を組み込む**」にチェックを入れ、「**検索**」ボタンを押すと、メインクラスとして選択可能なクラスの一覧が表示されますのでその中から選択してください。「**引数**」タブで、「**プログラム引数**」に「**-gen:client -d src -keep config.xml**」を指定します。以上の設定を確認し、[**実行**]ボタンを押します。プロジェクトの「**src**」フォルダ配下に、いくつかのソースコードが生成されます。生成されたソースコードが表示されていない場合は、「**src**」フォルダについて「**最新表示**」を行ってください。

クライアントコードの作成

「static」サンプルのソースコードをベースにして、クライアントコードの作成方法を説明します。

```
public class HelloClient {
```

```
    private javax.xml.rpc.Stub stub; //スタブを定義しておきます。

    //wscompile コマンドが生成したクラスの中で java.rmi.Remote を継承したインタフェースが
    //サービスエンドポイントインタフェースです。Web サービスがクライアントに提供している
    //インタフェースを記述したものです。
    private Hello_PortType binding; //サービスエンドポイントインタフェースを定義しておきます。

    public HelloClient() throws Exception {

        //javax.xml.rpc.Service を継承したインタフェースは、クライアントが
        //Web サービスを仮想化するために使用するインタフェースです。
        //このインタフェースを実装したクラスからポートを取得し、
        //スタブにキャストしています。
        stub = (javax.xml.rpc.Stub) new Hello_Service_Impl().getHelloPort();
        //スタブをさらにサービスエンドポイントインタフェースでキャストします。
        //これで Web サービスを呼び出す準備は整いました。
        binding = (Hello_PortType)stub;
    }

    //引数でアクセス先のサービスエンドポイントの URL を変更する場合のコンストラクタです。
    public HelloClient(java.lang.String url) throws Exception {
        stub = (javax.xml.rpc.Stub) new Hello_Service_Impl().getHelloPort();
        //スタブのプロパティをセットして URL を変更します。
        stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, url);
        binding = (Hello_PortType)stub;
    }

    //Web サービスを実行するメソッドです。
    public java.lang.String say_hello_Hello(java.lang.String in0) throws java.rmi.RemoteException {
        //実行したいオペレーションを呼び出します。
        return binding.say_hello_Hello(in0);
    }

    //クライアントを実行する main メソッドです。
    public static void main(String[] args) throws Exception {
        HelloClient client;
        if (args.length > 0) {
            client = new HelloClient(args[0]);
        } else {
            client = new HelloClient();
        }
        //Web サービスを実行し、結果を表示します。
        System.out.println(client.say_hello_Hello("webotx"));
    }
}
```

クライアントの実行

メニューから、**実行｜構成および実行** を選択し、実行ダイアログを開きます。**Java アプリケーション**を選択して[**新規**]ボタンを押して名前を指定し、新しい構成を作成します。「**メイン**」タブで、「**プロジェクト**」に作成しておいた Java プロジェクトを、「**メインクラス**」に作成したクライアントのクラスを指定します。引数を指定したい場合は、「**引数**」タブで指定します。以上の設定を確認したら[**実行**]ボタンを押します。

配布について

クライアントを JAR 形式にパッケージングする時は、作成したプロジェクトで右クリックし、エクスポートを選択します。エクスポートダイアログが起動したら、「JAR ファイル」を選択し、[次へ]ボタンを押します。エクスポートするリソースの選択では、「src」フォルダのみ選択し、エクスポート先を指定して[終了]ボタンを押します。

配布を行う時は、ここで作成した JAR ファイルとプロジェクトに追加していたライブラリ(tools.jar は除く)を配布するようにしてください。

JAX-RPC - ダイナミック(動的)プロキシ方式

公開されている WSDL を参照してクライアントを作成します。WSDL を参照したコーディング方法なので、動的にクライアントを生成して実行することも可能になります。ここでは、「DynamicProxy」サンプルをベースにクライアントコードの作成方法について説明します。また、動的にクライアントを生成して実行するためのアイディアを説明します。

クライアントコードの作成

「DynamicProxy」サンプルをベースにして、クライアントコードの作成方法を説明します。なお、このクライアントをコンパイル、または実行する環境では、次のライブラリがクラスパスに追加されている必要があります。

- <WebOTX_DIR>¥lib¥j2ee.jar
- <WebOTX_DIR>¥lib¥jaxrpc-impl.jar
- <WebOTX_DIR>¥lib¥saaj-impl.jar
- <WebOTX_DIR>¥lib¥endorsed¥resolver.jar
- <WebOTX_DIR>¥lib¥endorsed¥serializer.jar
- <WebOTX_DIR>¥lib¥endorsed¥xalan.jar
- <WebOTX_DIR>¥lib¥endorsed¥xml-apis.jar
- <WebOTX_DIR>¥lib¥endorsed¥xercesImpl.jar
- <J2SE_DIR>¥lib¥tools.jar(wscompile コマンドを使用する場合のみ必要)

```
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;

public class DynamicProxyHelloClient {

    public static void main(String[] args) {

        try {

            /*以下の 4 つの変数の値は、すべて WSDL から取得することができます。*/
            //(1)definitions 要素の targetNamespace 属性の値です。
            String nameSpaceUri = "http://sample/Hello";
            //(2)service 要素の name 属性の値です。
            String serviceName = "Hello";
            //(3)port 要素の name 属性の値です。
            String portName = "HelloPort";
            //(4)公開されている WSDL の URL の文字列です。この文字列は、最後に?wsdl が
            //ついている JAX-RPC 仕様で定めた形式でなければなりません。
            String urlString = "http://localhost:8080/HelloService/Hello?wsdl";
            /*WSDL から取得できる変数、ここまで*/

            //Web サービスを仮想オブジェクト化します。この部分は汎用的に使えます。
            URL wsdlUrl = new URL(urlString);
            ServiceFactory serviceFactory = ServiceFactory.newInstance();
            Service service =
              serviceFactory.createService(wsdlUrl, new QName(nameSpaceUri, serviceName));

            //Web サービスの仮想オブジェクトからポートを取得し、プロキシを生成します。
            //「Hello_PortType」クラスはサービスエンドポイントインタフェースです。
            Hello_PortType proxy =
              (Hello_PortType) service.getPort(new QName(nameSpaceUri, portName),
                                               Hello_PortType.class);

            //Web サービスにアクセスし、結果を出力します。
```

MEMO

「DynamicProxy」サンプルは、チュートリアルで作成した Web サービスにアクセスするダイナミックプロキシ方式で作成したクライアントです。

MEMO

<WebOTX_DIR>は WebOTX のインストールルートディレクトリ、<J2SE_DIR>は J2SE SDK のインストールルートディレクトリのことです。

MEMO

クライアントコード中で使っているサービスエンドポイントインタフェースは、JNDI などを使用して取得するか、wscompile コマンドを使用して生成してください。wscompile コマンドについてはスタティックスタブ方式を参照してください。

```
                System.out.println(proxy.say_hello_Hello("hello"));

            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
}
```

ダイナミックインボークのアイディア

さて、Web サービスの最初の目的のひとつに、「ダイナミックにサービスを見つけてダイナミックにサービスを利用する」という夢の実現がありました。このうち、「ダイナミックにサービスを見つける」という部分は、UDDI のようなレジストリサービスと JAXR のようなレジストリクライアントが実現します。残りの「ダイナミックにサービスを利用する」という部分を実現する方法の 1 つが、ここで説明している「ダイナミックプロキシ方式」のクライアントです。では、これから動的実行を実現するアイディアをいくつか説明します。このアイディアを取り込んであなたの目的を果たしていただけたら幸いです。

まずはじめに、必ずやらなければならないことが公開されている WSDL の URL を取得することです。そして、その WSDL を読み込んでクライアントコードに必要な 4 つの情報を取り出し、クライアントコードの生成ロジックに渡します。WSDL から情報を取得する方法は、DOM や SAX を使う方法もあれば、WebOTX が提供しているモジュール「wsdl4j.jar」の API を使う方法もあります。好きな方法を選択してください。

次に、「クライアントコードの作成」で説明したようなクライアントコードを生成するわけですが、ほとんどの部分は汎用的な部品ですので、実はサンプルがかなり流用できます。ここでただ 1 つ、やらなければならないのがサービスエンドポイントインタフェースの取得とクライアントコードへの反映です。サービスエンドポイントインタフェースは、公開されていたり JNDI 経由で取得できるかも知れませんが、現実を考えるとそれが実現できる可能性はほぼゼロでしょう。そこで、サービスエンドポイントインタフェースは WSDL から「wscompile コマンド」を利用して生成してしまいます。コマンドの実行方法は、前述のスタティックスタブ方式のところを参照してください。-d オプションを工夫すればクライアントコードを生成する場所を指定できるので、いくつもの WSDL を相手にすることができます。-keep オプションはソースコードを残すためのオプションなので、必要ないかもしれません。サービスエンドポイントインタフェースは、ポート毎に生成されるもので、WSDL の portType 要素の name 属性の値に「_PortType」をつけた名前で生成されます。難しいのは、サービスへアクセスして結果を得る部分でしょう。引数や返り値がどんな値なのかということをどのように取得するかが難しいのです。メソッド名や引数と返り値の型はサービスエンドポイントインタフェースから取得できますが、「どんな値」を受け渡しするのかがわからないといけません。String だとしても、名前なのか、住所なのか、それとも、、、ということです。もし、型が任意の JavaBean だとしたらさらに複雑です。こうした問題は、業界標準のようなあらかじめ決められたインタフェースがあるという前提があってはじめて解決するのかも知れません。

クライアントコードの生成ができたら、Java コンパイルをかけましょう。それができたらいよいよクライアントの実行です。

ここまでの流れを確立できたら、ant を使った自動化をやってみてください。それが、「ダイナミックインボーク」です。健闘を祈ります。

JAX-RPC - DII方式

JAX-RPC の API を直接呼出して使い、アクセスする Web サービスの情報を手動で設定しすることによってクライアントを作成する方法です。この方法は、WSDL がない場合や WSDL に依存せずに細かい設定を手動でしたい場合など、ごく限られた用途で使用する特殊な方法で、SOAP や JAX-RPC 仕様を理解した上級者向けに用意されたクライアント作成手段であると理解してください。通常、この方法を使うことは推奨されません。ここでは、「DII」サンプルをベースに、クライアントコードの作成方法について簡単に説明します。

環境構築

DII 方式で作成したクライアントをコンパイル、実行する環境には、次のライブラリがクラスパスに追加されている必要があります。

- <WebOTX_DIR>¥lib¥j2ee.jar
- <WebOTX_DIR>¥lib¥jaxrpc-impl.jar
- <WebOTX_DIR>¥lib¥saaj-impl.jar
- <WebOTX_DIR>¥lib¥endorsed¥resolver.jar
- <WebOTX_DIR>¥lib¥endorsed¥serializer.jar
- <WebOTX_DIR>¥lib¥endorsed¥xalan.jar
- <WebOTX_DIR>¥lib¥endorsed¥xml-apis.jar
- <WebOTX_DIR>¥lib¥endorsed¥xercesImpl.jar

MEMO

<WebOTX_DIR>は WebOTX のインストールルートディレクトリのことです。

クライアントコードの作成

「DII」サンプルをベースに、クライアントコードの作成方法について簡単に説明します。

MEMO

「DII」サンプルは、チュートリアルで作成した Web サービスにアクセスする DII 方式で作成したクライアントです。

```
import javax.xml.namespace.QName;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;

public class DIIHelloClient {

    //Web サービス名です。
    private static String qnameService = "Hello";
    //ポート名です。
    private static String qnamePort = "HelloPort";
    //Web サービスの名前空間 URI です。
    private static String BODY_NAMESPACE_VALUE = "http://sample/Hello";

    //固定値の定義です。
    private static String ENCODING_STYLE_PROPERTY =
      "javax.xml.rpc.encodingstyle.namespace.uri";
    private static String NS_XSD = "http://www.w3.org/2001/XMLSchema";
    private static String URI_ENCODING = "http://schemas.xmlsoap.org/soap/encoding/";

    public static void main(String[] args) {

        try {

            //サービスのインスタンスを作成します。
            ServiceFactory factory = ServiceFactory.newInstance();
            Service service = factory.createService(new QName(qnameService));
            //ポートから Call オブジェクトを作成します。
            QName port = new QName(qnamePort);
            Call call = service.createCall(port);
            //Call オブジェクトに各種設定を行います。
            call.setTargetEndpointAddress("http://localhost:8080/HelloService/Hello");
```

```
                call.setProperty(Call.SOAPACTION_USE_PROPERTY, new Boolean(true));
                call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
                call.setProperty(ENCODING_STYLE_PROPERTY, URI_ENCODING);
                QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");
                call.setReturnType(QNAME_TYPE_STRING);
                call.setOperationName(new QName(BODY_NAMESPACE_VALUE, "say_hello_Hello"));
                call.addParameter("String_1", QNAME_TYPE_STRING, ParameterMode.IN);
                //Web サービスを実行します。
                System.out.println((String) call.invoke(new String[]{ "hello" }));

            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
}
```

SAAJ

SAAJ の API を利用し、Web サービスと交換する SOAP メッセージを直接処理するタイプのクライアントです。任意の XML 文書を交換する document/literal な Web サービスにアクセスする場合、この方法を用いてクライアントを作成します。

環境構築

SAAJ の API を利用したクライアントをコンパイル、実行する環境には、次のライブラリがクラスパスに追加されている必要があります。

- <WebOTX_DIR>¥lib¥j2ee.jar
- <WebOTX_DIR>¥lib¥saaj-impl.jar
- <WebOTX_DIR>¥lib¥resolver.jar
- <WebOTX_DIR>¥lib¥serializer.jar
- <WebOTX_DIR>¥lib¥xalan.jar
- <WebOTX_DIR>¥lib¥xml-apis.jar
- <WebOTX_DIR>¥lib¥xercesImpl.jar

MEMO

<WebOTX_DIR>は WebOTX のインストールルートディレクトリのことです。

クライアントコードの作成

「SAAJ」サンプルをベースに、クライアントコードの作成方法について説明します。

```
import java.io.OutputStream;
import java.util.Iterator;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.Name;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPBodyElement;
import javax.xml.soap.SOAPConnection;
import javax.xml.soap.SOAPConnectionFactory;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Node;

public class SAAJClient {
        //サービスの URL です。
        private static final String URL = "http://localhost/saaj/sample";
        //SOAP メッセージの作成に使用する値です。
        private static final String NS = "http://sample/WebOTX";//名前空間
        private static final String PREFIX = "n";//プレフィックス
        private static final String ELEMENT1 = "data";//要素名
        private static final String ELEMENT2 = "No";//要素名

        //メインメソッド
        public static void main(String[] args) throws Exception {
                SAAJClient client = new SAAJClient();
                //Web サービスを実行し、結果をコンソールに表示します。
                //リクエストメッセージはプログラミングにより、1 要素ずつ作成します。
                client.invoke(client.createMessage(), System.out);
```

```
            //結果表示にあたり、空白行を挿入します。
            System.out.println();
            System.out.println();
            //Web サービスを実行し、結果をコンソールに表示します。リクエストメッセージ
            //は、あらかじめ SOAPBody が記述されている XML ファイルを読み込み、
            //作成します。
            client.invoke(client.createMessage("sample.xml"), System.out);
    }
    //Web サービスを実行します。引数で渡された SOAPMessage がリクエストメッセージに
    //なります。
    private void invoke(SOAPMessage message, OutputStream out) {
            try {
                //コネクションを張ります。
                SOAPConnectionFactory scf =
                        SOAPConnectionFactory.newInstance();
                SOAPConnection con = scf.createConnection();
                //Web サービスを実行して、サービスから返されたレスポンスメッセージ
                //を受け取ります。
                SOAPMessage reply = con.call(message, URL);
                //SOAPBody を取り出します。
                SOAPBody body = reply.getSOAPBody();
                //SOAPBody のエレメントを取り出します。
                Iterator ite = body.getChildElements();
                        //SOAPBody から取り出したノードを XML 形式に整形して
                        //出力します。
                        while (ite.hasNext()) {
                                Node node = (Node) ite.next();
                                Source source = new DOMSource(node);
                                TransformerFactory factory =
                                        TransformerFactory.newInstance();
                                Transformer transformer = factory.newTransformer();
                                Result output = new StreamResult(out);
                                transformer.transform(source, output);
                        }
                        //コネクションを閉じます。
                        con.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
    }

    //SOAPMessage を SAAJ プログラミングにより作成します。
    private SOAPMessage createMessage() throws SOAPException {
            //SOAPMessage を作成します。
            MessageFactory mf = MessageFactory.newInstance();
            SOAPMessage message = mf.createMessage();
            //SOAPMessage から SOAPPart を取り出し、さらに SOAPEnvelope を
            //取り出します。
            SOAPEnvelope env = message.getSOAPPart().getEnvelope();
            //SOAPEnvelope から SOAPBody を取り出す
            SOAPBody body = env.getBody();
            //SOAPBody に各要素を追加します。
            Name name = env.createName(ELEMENT1, PREFIX, NS);
            SOAPBodyElement bodyElement = body.addBodyElement(name);
            name = env.createName(ELEMENT2, PREFIX, NS);
            bodyElement.addChildElement(name).addTextNode("001");
            //作成した SOAPMessage を返します。
            return message;
    }

    //XML を読み込み、SOAP メッセージを作成します。
```

```
        private SOAPMessage createMessage(String filelocation) throws Exception {
                //SOAPMessage を作成します。
                MessageFactory mf = MessageFactory.newInstance();
                SOAPMessage message = mf.createMessage();
                //SOAPMessage から SOAPPart を取り出し、さらに SOAPEnvelope を
                //取り出します。
                SOAPEnvelope env = message.getSOAPPart().getEnvelope();
                //SOAPEnvelope から SOAPBody を取り出します。
                SOAPBody body = env.getBody();
                //DOM を使用して、SOAPBody が記述されている XML ファイルを読み
                //込み、SOAPBody に追加します。
                DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
                dbf.setNamespaceAware(true);
                DocumentBuilder db = dbf.newDocumentBuilder();
                Document doc = db.parse(filelocation);
                //SOAPBody に追加します。
                body.addDocument(doc);
                //作成した SOAPMessage を返します。
                return message;
        }
}
```

クライアントの実行

メニューから、**実行**｜**構成および実行**を選択し、実行ダイアログを開きます。**Java アプリケーション**を選択して[**新規**]ボタンを押して名前を指定し、新しい構成を作成します。「**メイン**」タブで、「**プロジェクト**」に作成しておいた Java プロジェクトを、「**メインクラス**」に作成したクライアントのクラスを指定します。引数を指定したい場合は、「**引数**」タブで指定します。以上の設定を確認したら[**実行**]ボタンを押します。

タイムアウト設定

JAX-RPC クライアントはサーバへの接続に関するタイムアウト時間を設定できます。 この機能を利用することで、一定時間応答のないサーバなどから強制的に切断して、次の処理に移行することができます。

設定方法

java 実行時にシステムプロパティとして以下を設定することで、HTTP タイムアウトの制御ができます。

- sun.net.client.defaultConnectTimeout： Web サービスとの接続までの待ち時間です。
- sun.net.client.defaultReadTimeout： Web サービスに接続してから応答を待っている時間です。

引数に含める単位はミリ秒です。初期値は-1(設定なし)です。0 を設定した場合接続を維持してサーバからの応答を待ち続けます。

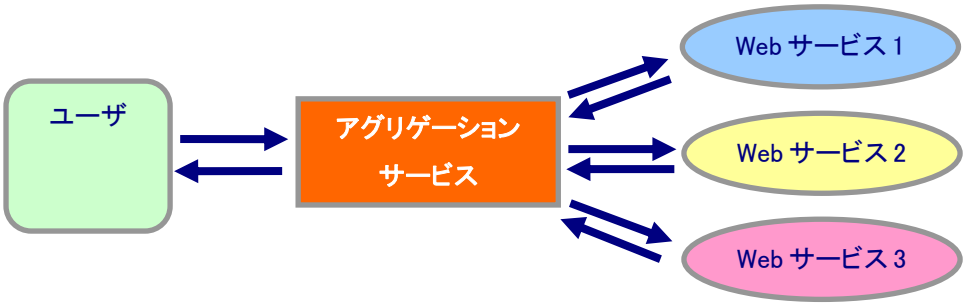
この値の詳細は Sun JDK に記載されています。「JDK ドキュメント」-「ネットワーク機能」-「ネットワークプロパティ」

これらの設定は JAX-RPC クライアントが実行される JVM 全体へ設定されます。他に HTTP プロトコルで接続するプログラムがある場合、影響を受ける場合がありますので注意してください。例えば、JAX-RPC クライアントを Servlet 上で動作させる時、動作する WebOTX ドメインのプロパティに設定した場合に HTTP クライアントとして動作するプログラム(Servlet, EJB など)全体に反映されます。

2.1.8.アグリゲーションサービスの開発

複数の Web サービスを束ねて、1 つの Web サービスとして見せる「**アグリゲーションサービス**」の構築について紹介します。

アグリゲーションサービスとは



アグリゲーションサービスは、それ自身がユーザに提供する Web サービスのコンテンツを実現するために、**自分自身以外の複数のサービスを利用**します。アグリゲーションサービス内部で動いているビジネスロジックでは、他の Web サービスのクライアントが複数動くことになります。アグリゲーションサービスを開発する時は、まず、アグリゲーションサービスから利用したい Web サービスのクライアントを作成し、それらを組み合わせたビジネスロジックを構築します。次に、通常の Web サービス作成手順で作成したビジネスロジックを Web サービス化します。

MEMO

WebOTX Enterprise Service Bus（オプション製品）を使うと、複雑なコーディングをすることなく、アグリゲーションサービスを構築することができます。

アグリゲーションサービスの構築手順

ここでは、「aggregation」サンプルの作成方法を例に、アグリゲーションサービスの構築手順を説明します。

「aggregation」サンプルは、アグリゲーションサービスから 2 つの Web サービスにアクセスし、それによって得られた結果をクライアントに返すシンプルな構成です。2 つの Web サービスは、文字列を受け取り、その文字列に Web サービス固有の文字列を付加して返す単純なものです。アグリゲーションサービスは、2 つの Web サービスそれぞれの URL と、それぞれの Web サービスに渡す文字列を引数としてクライアントから受け取り、2 つの Web サービスから返ってきた値を「Info」という JavaBean に格納してクライアントに返却します。

ビジネスロジックの作成①

1 つ目の Web サービスで動かすビジネスロジックを作成します。サンプルでは、「sample1」というプロジェクトで、1 つ文字列を受け取り、それに文字列を付加してから返却するという処理が実装されています。また、結果として「sample1.jar」ファイルを作成しています。

ビジネスロジックの作成②

2 つ目の Web サービスで動かすビジネスロジックを作成します。サンプルでは、「sample2」というプロジェクトで、1 つ文字列を受け取り、それに文字列を付加してから返却するという処理が実装されています。また、結果として「sample2.jar」ファイルを作成しています。

Webサービスの作成①

Web サービス作成ウィザードで、1 つ目のビジネスロジックを Web サービス化します。また、WAR もしくは EJB-JAR 形式のアーカイブを作成し、Web サービスを配備します。さらに、プロジェクトに生成されているクライアントをアグリゲーションサービスのビジネスロジックで利用するために、「src」フォルダ部分のみを指定して JAR アーカイブを作成します。サンプルでは、「sampleservice1」プロジェクトに作成しており、結果として「sample1.war」と「sampleservice1.jar」を作成しています。

Webサービスの作成②

Web サービス作成ウィザードで、1 つ目と同様に 2 つ目のビジネスロジックを Web サービス化します。サンプルでは、「sampleservice2」プロジェクトに作成しており、結果として「sample2.war」と「sampleservice2.jar」を作成しています。

アグリゲーションサービスのビジネスロジック作成

2 つの Web サービスのクライアントを作成し、それらを呼び出して使用しながらアグリゲーションサービス

のビジネスロジックを作成します。サンプルでは、「aggregation」プロジェクトに「sampleservice1」と「sampleservice2」を取り込んでそれらをマージするコードを作成しており、結果として「aggregation.jar」を作成しています。なお、アグリゲーションサービスのビジネスロジックを作るために作成するプロジェクトには、Web サービスのクライアントをコンパイル・実行するためのライブラリを追加してください。詳細は、『Web サービスを利用する』の項を参照してください。

アグリゲーションサービスのWebサービス作成

Web サービス作成ウィザードで、アグリゲーションサービスのビジネスロジックを Web サービス化します。サンプルでは、「AggregationService」プロジェクトを作成しており、結果として「aggregation.war」を作成しています。

アグリゲーションサービスの配備とクライアントの実行

アグリゲーションサービスを配備し、アグリゲーションサービスのプロジェクトに生成されているクライアントを実行します。

2.1.9.マイグレーション

旧バージョンの WebOTX、他社製アプリケーションサーバ、オープンソースソフトウェアなどを使って Web サービスを構築した方が、WebOTX Application Server V7.x に**アップグレード**または**移行**するときに必要な作業について説明します。

WebOTX V6.xからのアップグレード

V6 ランタイムを使用した Web サービスを WebOTX Application Server V7.x に配備する場合、特に作業は必要ありません。お持ちの Web サービスアプリケーションはそのまま配備して使用できます。

V5 ランタイムを使用した Web サービスは WebOTX Application Server V7.x では動作しません。WebOTX Developer V7.x の Web サービス作成ウィザードで Web サービス化をやり直してください。

WebOTX V5.3、V5.2 からのアップグレード

WebOTX V5.2、V5.3 用に作成した Web サービスアプリケーションは、WebOTX Application Server V7.x でそのまま動作させることができますが、アプリケーションを配備する前に次のようにして最適化されることをおすすめします。

- WAR ファイルに含める **server-config.wsdd** ファイルをテキストエディタで開き、次の記述がある場合、その部分を削除してください。

```
<globalConfiguration>
  <requestFlow>
    <handler type="java:org.apache.axis.handlers.JWSHandler">
      <parameter name="scope" value="session"/>
    </handler>
    <handler type="java:org.apache.axis.handlers.JWSHandler">
      <parameter name="scope" value="request"/>
      <parameter name="extension" value=".jwr"/>
    </handler>
  </requestFlow>
</globalConfiguration>
```

```
<service name="Version" provider="java:RPC">
  <parameter name="allowedMethods" value="getVersion"/>
  <parameter name="className" value="org.apache.axis.Version"/>
</service>
```

```
<transport name="local">
  <responseFlow>
    <handler type="java:org.apache.axis.transport.local.LocalResponder"/>
  </responseFlow>
</transport>
```

WebOTX V5.1、V4.2、V4.1 からのアップグレード

WebOTX V5.1、V4.2、V4.1 用に作成した Web サービスは WebOTX Application Server V7.x では動作しません。WebOTX Developer V7.x の Web サービス作成ウィザードで Web サービス化をやり直してください。

他社製品やオープンソースソフトウェアからの移行

WebOTX V5.1、V4.2、V4.1 からのアップグレードと同じ方法で移行してください。

Web Service Director 1.0/1.0aとの互換性

WebOTX V5.2、V5.3、ならびに Web Service Interoperability Pack で提供していた Web Service Director 1.0 または 1.0a のプロジェクトと、WebOTX Developer V7.x の Web サービスプロジェクトには互換性がありません。ただし、Web Service Director で作成したアプリケーションは、WebOTX Application Server V7.x にそのまま配備することができます。

2.1.10.コマンドリファレンス

wscompile

wscompile コマンドは、Java のインタフェースから WSDL を生成したり、WSDL からサービスエンドポイントインタフェースやスタブ等の Java ソースコードを生成することができます。ここでは、wscompile の使用方法について説明します。

環境構築

wscompile を実行するためには、以下の JAR ファイルをユーザクラスパスに追加する必要があります。

- <WebOTX_DIR>¥lib¥jaxrpc-impl.jar
- <WebOTX_DIR>¥lib¥j2ee.jar
- <J2SE_DIR>¥lib¥tools.jar

ライブラリはユーザクラスパス、またはバッチファイル内でクラスパスを通すようにしてください。Windows 環境変数などシステム設定に含めると WebOTX AS が正常に動作できない場合がありますのでご注意ください。

MEMO

<WebOTX_DIR>は WebOTX のインストールルートディレクトリのことです。
<J2SE_DIR>は J2SE SDK のインストールルートディレクトリのことです。

オプション

wscompile で使用するオプションについて説明します。wscompile は引数に、これらのオプションを指定して実行します。

オプション	説明
-define	WSDL を生成する場合に指定します。
-gen(または-gen:client)	クライアントサイドの Java のコードを生成する場合に指定します。
-gen:server	サーバサイドの Java のコードを生成する場合に指定します。
-gen:both	クライアントサイド、サーバサイド両方の Java のコードを生成する場合に指定します。
-d	生成ファイルの出力場所を指定します。-nd、-s を指定しない場合は、このオプションで指定した場所にマッピングファイル以外のすべてのファイルが生成されます。生成場所として指定可能なディレクトリは、すでに存在しているものに限ります。
-nd	WSDL ファイルの生成場所を指定します。
-s	Java ソースファイルの生成場所を指定します。
-mapping	マッピングファイルの生成場所を指定します。
-keep	Java ソースファイルを生成します。デフォルトでは、ソースファイル自体が生成されずにコンパイル結果のみが生成されます。
-f:wsi	WS-I に対応した Java のコードや WSDL を生成します。
-f:documentliteral	SOAP メッセージ形式を document/literal で生成します。
-f:rpcliteral	SOAP メッセージ形式を rpc/literal で生成します。
-f:amenableSequence	WSDL の message-part 要素の順番で SOAP メッセージを構築します。このオプションは-f:wsi と同時に利用することで有効になります。
-f:ignoreFilePathCheck	ファイルパス長のチェック機能を無視します。ファイルパス長チェック機能の有効、無効化は Windows 版のみ対応しています。
-f:useonewayoperations	一方向通信を行います。SOAP メッセージ形式が RPC では void のメソッドについて、Document では全てのメソッドが対象になります。Document のとき、ユーザ定義の例外は宣言できません。

-define、-gen、-gen:client、-gen:both のうち必ず 1 つは指定しなければなりません。複数同時に指定することはできません。RPC/literal、Document/literal においては void のメソッドを Web サービス化する場合、-f:useonewayoperations を使って一方向通信にするのが基本ですが、空のメッセージが返るようにしたい場合は、コマンド実行時の Java VM オプションで「**-DvoidReturn=true**」を設定してください。

設定ファイル

wscompile を実行するには、設定ファイル(XML ファイル)が必要となります。実行には、オプションと同様に引数に設定ファイルを指定する必要があります。設定ファイルは、WSDL を生成する場合と、サービスエンドポイントインタフェースやスタブなどの Java ソースコードを生成する場合では、指定する要素や属性が異なります。このファイル名に規定はありません。以降では config.xml として説明します。

- サービスエンドポイントインタフェースから WSDL を生成する場合

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration
      xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <service name="MyHelloService"
                targetNamespace="urn:Foo"
                typeNamespace="urn:Foo"
                 packageName="helloservice">
          <interface name="helloservice.HelloIF" />
    </service>
</configuration>
```

service 要素の name 属性には Web サービス名を、targetnamespace 属性には WSDL の名前空間 URI を、typeNamespace には schema 要素の名前空間 URI を、packageName には Java パッケージ名をそれぞれ指定します。

- WSDL から Java ソースコードを生成する場合

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
      xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <wsdl location="test.wsdl"
          packageName="test"/>
</configuration>
```

wsdl 要素の location 属性には WSDL ファイルを指定し、packageName 属性には生成する Java クラスのパッケージ名を指定します。

実行コマンド

```
java com.nec.webotx.webservice.xml.rpc.tools.wscompile.Main <オプション> <設定ファイル>
```

実行例

- サービスエンドポイントインタフェースから WSDL を生成する場合

```
java com.nec.webotx.webservice.xml.rpc.tools.wscompile.Main -define -d C:¥tmp config.xml
```

- WSDL から Java ソースコードを生成する場合

```
java com.nec.webotx.webservice.xml.rpc.tools.wscompile.Main -gen -d C:¥tmp -keep config.xml
```

WSDL を生成するために使用する Java インタフェースは、java.rmi.Remote を継承する必要があります。また、インタフェース内で宣言されているすべてのメソッドは java.rmi.RemoteException をスローしなければなりません。

2.1.11.セキュリティ

ここでは、SSL や Web Services Security の機能を利用して、セキュアな Web サービスを構築する手順について説明します。

SSLの機能

SSL は通信路を暗号化し、通信相手を認証することによって、盗聴、改ざん、なりすましを防止します。JAX-RPC や SAAJ における Web サービスでは、SOAP メッセージの外側を覆う HTTP のレイヤーが暗号化の対象になります。つまり、必ず SOAP メッセージ全てが暗号化され、SOAP メッセージの一部だけを暗号化することはできません。また、Web サービスクライアントと Web サービスが 1 対 1 のときにのみ有効です。メッセージを受け渡す仲介者がいるような場合は、SSL はあまり有効ではない場合があります。

Web Services Securityの機能

WebOTX は、OASIS Web Services Security 1.0 (WS-Security 2004)に対応し、SOAP メッセージレベルにおけるセキュリティ機能を提供しています。SSL との最大の違いは、認証や暗号化などを行うレイヤーが通信路ではなくメッセージであるということです。用途に応じてどちらの機能を利用するかをご検討ください。

WebOTX の Web Services Security 対応機能を利用すると次のようなことができます。SSL とは違った豊富な機能を持っています。

■デジタル署名

クライアントとサーバでやり取りされる双方向の SOAP メッセージに対して、それぞれの SOAP Body 要素、あるいはその中の特定の要素を対象としてデジタル署名をすることができます。これにより、メッセージの改ざん、なりすまし、否認を防止します。

■暗号化

クライアントとサーバでやり取りされる双方向の SOAP メッセージに対して、それぞれのオペレーション要素の内容（Content）、あるいはその中の特定のパラメータ要素の内容（Content）を暗号化することができます。これにより、メッセージの盗聴を防止します。

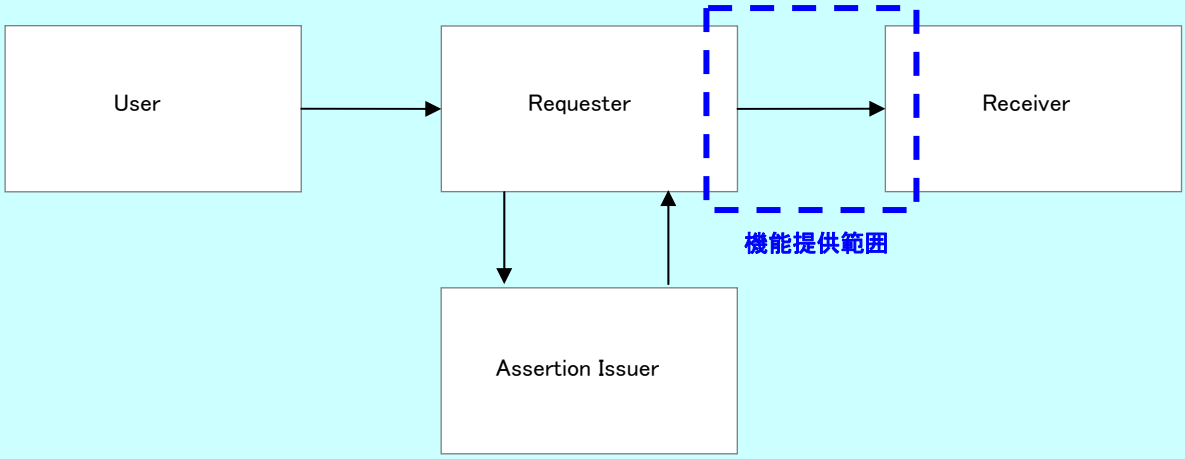
■タイムスタンプ

クライアントとサーバでやり取りされる双方向の SOAP メッセージに対して、それぞれにタイムスタンプを付与することができます。これにより、リプレイ攻撃などを防止します。タイムスタンプに署名したり、暗号化することもできます。

■認証

ユーザネームトークン、デジタル署名、SAML による認証機能を提供します。ユーザ名とパスワードの照合は独自の仕組みによって実現していただくことができるほか、WebOTX の持つユーザ認証機構を利用することも可能です。また、ユーザ名やパスワードは暗号化することもできます。SAML は、WebSAM SECUREMASTER と連携して、SAML Assertion の付与、検証機能を提供します。これにより、Web サービスにおけるシングルサインオン環境の実現に貢献します。WebOTX は、「Sender-Vouches モデル」と「Holder-of-key モデル」におけるリクエスタとレシーバ間のアサーションの付与/検証、および署名の付与/検証を行います。

Sender-Vouches モデル



■登場人物

- User … サービスを利用するエンティティ
- Requester … User からの要求を受け、Receiver にサービスを要求するエンティティ
- Assertion Issuer … User に関する情報を証明する SAML アサーションを発行するエンティティ
- Receiver … サービスを提供するエンティティ

■前提条件

- Receiver と Requester は事前に(PKI に基づく)信頼関係を構築しておく必要があります。

■処理概要

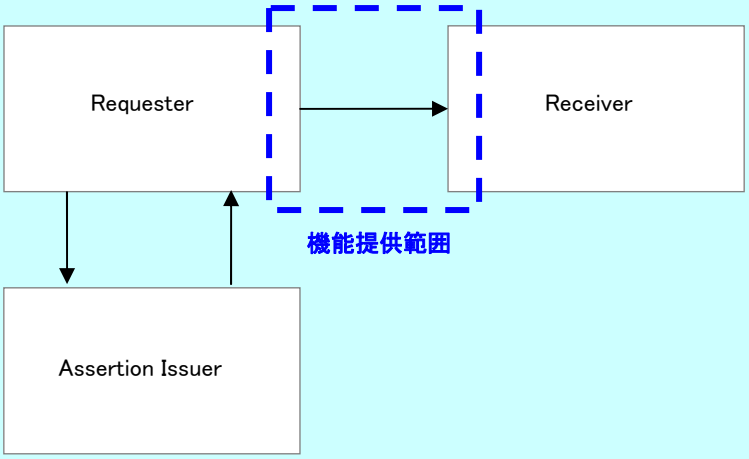
- User は Requester に Receiver へのアクセス要求を出します。
- Requester は Assertion Issuer に User のための Assertion 発行要求を出します。
- Assertion Issuer は Assertion を発行して Requester へ返します。
- **Requester は Assertion を SOAP メッセージの Security ヘッダへ格納し、Requester の秘密鍵で Assertion および 必要な部分に署名し、署名に使用した秘密鍵とペアになる証明書(あるいは証明書を特定するための情報)を Security ヘッダに格納し、この SOAP メッセージを Receiver へ送信します。**
- **Receiver は 受信した SOAP メッセージを解析し、署名に関連付けられている証明書を取得し、Assertion および その他の部分の署名の検証を行います。また、その証明書の検証も併せて行います。**

■注意

Assertion Issuer が発行する Assertion には、以下の<SubjectConfirmation>要素が含まれている必要があります。

```
<SubjectConfirmation>
  <ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:sender-vouches</ConfirmationMethod>
  ...
</SubjectConfirmation>
```

Holder-Of-Key モデル



■登場人物

- User … サービスを利用するエンティティ
- Assertion Issuer … User に関する情報を証明する SAMLAssertion を発行するエンティティ
- Receiver … サービスを提供するエンティティ

■前提条件

- Receiver と Assertion Issuer は事前に(PKI に基づく)信頼関係を構築しておく必要があります。

■処理概要

- User は、User の証明書を添えて Assertion Issuer に Assertion 発行要求を出します。
- Assertion Issuer は User の証明書を含めた Assertion を発行し、Assertion Issuer の秘密鍵で Assertion を署名して、User へ返します。(署名に使用した秘密鍵とペアになる証明書も Assertion に(署名要素に)含めます。)
- **User は Assertion を SOAP メッセージの Security ヘッダへ格納し、User の秘密鍵で必要な部分に署名し、この SOAP メッセージを Receiver へ送信します。**
- **Receiver は 受信した SOAP メッセージを解析し、Assertion から Assertion Issuer の 証明書を取り出し、Assertion の署名の検証を行います。また、その証明書の検証も併せて行います。次いで、Assertion から User の証明書を取り出し、User が付与した署名の検証を行います。**

■注意

Assertion Issuer が発行する Assertion には、以下の<SubjectConfirmation>要素が含まれている必要があります。

```
<SubjectConfirmation>
  <ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:holder-of-key</ConfirmationMethod>
  ...
</SubjectConfirmation>
```

SSLの使用方法

WebOTX では Web サーバとして、Web コンテナに内蔵されている Web サーバと、WebOTX Web Server、IIS、Apache 、Sun ONE などの外部 Web サーバが利用できます。ここでは、Web コンテナに内蔵されている Web サーバ、WebOTX Web Server で TLS/SSL(以降 SSL とします)をプライベート CA を立てす･に利用する方法について説明します。

[ご参考：SSL アクセラレータ]
SSL の暗号化・復号化処理をハードウェアで行い、Web サーバの負荷を軽減する方法があります。これを実現するのが「SSL アクセラレータ」です。SSL アクセラレータは、サーバの拡張スロットに差し込むボード、外付けの専用機、ルータやロードバランサなどのネットワーク機器に内蔵されているものなど、様々なタイプの製品があります。

Webコンテナ内蔵Webサーバを使う場合

WebOTX Web コンテナに内蔵されている Web サーバで SSL を利用する手順について説明します。ここでは、図 2.1.10.a のようなファイル構成を仮定して説明します。単方向/双方向認証と実際に使用するファイル名は、動作させる環境に応じて置き換えてください。

MEMO
説明の中で例示しているコマンドが 2 行になっていても、必ず 1 行で実行してください。

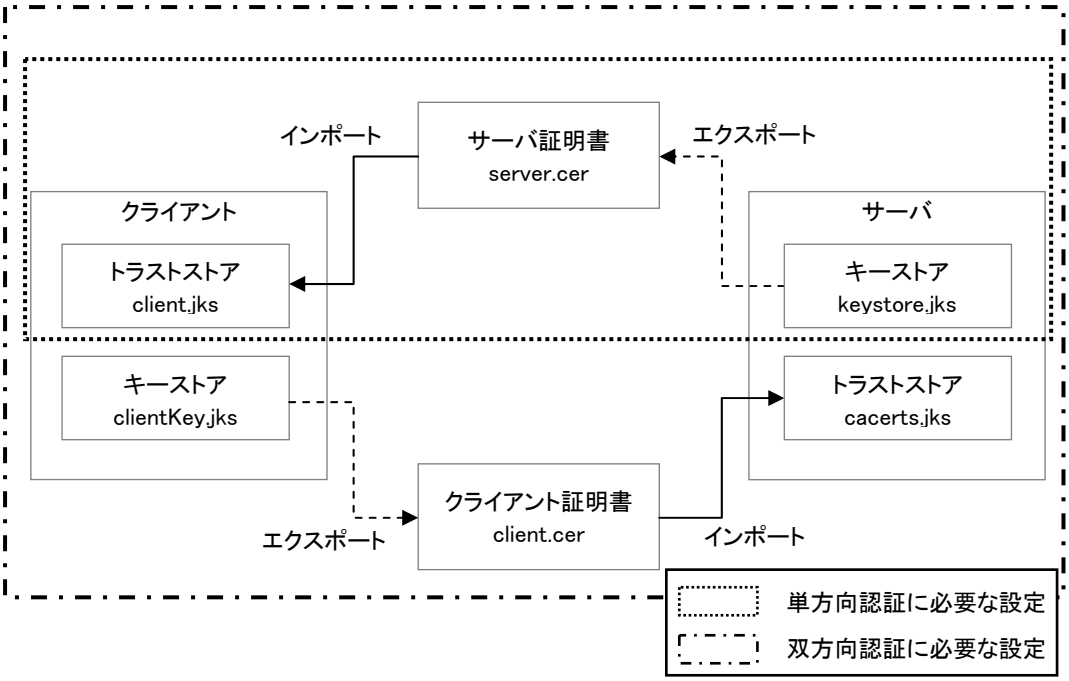


図 2.1.10a

単方向認証を行うためには、サーバのキーストアからサーバ証明書を取り出し、クライアント側のトラストストアへインポートします。

さらに双方向認証を行うためには、単方向認証の設定に加えて、クライアントのキーストアからクライアント証明書を取り出し、サーバ側のトラストストアへインポートします。

以上を設定するための手順として、J2SE のコマンドと WebOTX の運用管理コマンド(otxadmin)を利用した場合について説明します。

(1)J2SE の keytool コマンドを利用してサーバ用の鍵を作成します。下にコマンドの例を示します。<DOMAIN_ROOT>は Web サービスを配備するドメインのルートディレクトリです。ドメインにはじめから設定されているキーストア(keystore.jks)に鍵を追加します。keystore.jks のパスワードは「changeit」です。-dname オプションの CN 属性は必ずサーバのマシン名に一致させなければなりません。

```
keytool -genkey -alias server -dname "CN=server_name, O=X, L=Y, C=XX, OU=YY, S=XY" -keyalg RSA -keypass changeit -storepass changeit -keystore <DOMAIN_ROOT>¥config¥keystore.jks
```

(2)サーバの証明書をエクスポートします。

```
keytool -export -alias server -storepass changeit -file server.cer -keystore <DOMAIN_ROOT>¥config¥keystore.jks
```

(3)クライアント用のキーストア、鍵を作成します。

```
keytool -genkey -alias client -dname "CN=client_name, O=X, L=Y, C=XX, OU=YY, S=XY" -keyalg
RSA -keypass changeit -storepass changeit -keystore client.jks
```

双方向認証(クライアント認証)をするときには同様に別途キーストアをもう一つ作成します。例では clientKey.jks としています。

```
keytool -genkey -alias client -dname "CN=client_name, O=X, L=Y, C=XX, OU=YY, S=XY" -keyalg
RSA -keypass changeit -storepass changeit -keystore clientKey.jks
```

(4)クライアントの証明書をエクスポートします。これは双方向認証(クライアント認証)をするときのみ必要な作業です。

```
keytool -export -alias client -storepass changeit -file client.cer -keystore clientKey.jks
```

(5)クライアントのキーストアにサーバの証明書をインポートします。

```
keytool -import -trustcacerts -alias server -file server.cer -keystore client.jks -keypass changeit
-storepass changeit
```

(6)サーバのキーストアにクライアントの証明書をインポートします。これは双方向認証(クライアント認証)をするときのみ必要な作業です。

```
keytool -import -trustcacerts -alias client -file client.cer -keystore
<DOMAIN_ROOT>¥config¥keystore.jks -keypass changeit -storepass changeit
```

(7)運用管理コマンドを起動し、Web サービスを配備するドメインにログインします。続いて、create-ssl を実行し、HTTP リスナに SSL の設定を追加します。certname オプションでは SSL の認証に使うエイリアスを指定します。

```
otxadmin> login --user admin --password adminadmin --port 6212
```

(単方向認証のとき)

```
otxadmin> create-ssl --type http-listener --certname server --clientauthenabled=false
http-listener-2
```

(双方向認証のとき)

```
otxadmin> create-ssl --type http-listener --certname server --clientauthenabled=true
http-listener-2
```

(8)ドメインを再起動します。

(9)クライアント環境にクライアントのキーストアを配置します。例えば、WebOTX Developer の Web サービスプロジェクトに生成されたクライアントをそのまま Web サービスプロジェクト上で動作させる場合、その Web サービスプロジェクト内にキーストアを配置します。以降の説明では、Web サービスプロジェクトの keystore フォルダ配下に配置したものとして説明します。

(10)Web サービスのクライアントに関係するクラスの中に、次のコードを追加します。コンストラクタなど、インスタンス生成のタイミングで実行される場所に追加し、確実にサーバの呼び出し処理が実行される前に実行されるようにします。javax.net.ssl.trustStore キーと javax.net.ssl.keyStore キーの値は、クライアントのキーストアのパスになることに注意し、適宜変更してください。

(単方向認証のとき)

```
System.setProperty("java.protocol.handler.pkgs","com.sun.net.ssl.internal.www.protocol");
System.setProperty("javax.net.ssl.trustStore", "keystore/client.jks");
java.security.Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
```

(双方向認証のとき)

```
System.setProperty("java.protocol.handler.pkgs","com.sun.net.ssl.internal.www.protocol");
System.setProperty("javax.net.ssl.trustStore", "keystore/client.jks");
System.setProperty("javax.net.ssl.keyStore", "keystore/clientKey.jks");
System.setProperty("javax.net.ssl.keyStorePassword", " changeit");
java.security.Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
```

(11)EJB サービスエンドポイントの場合、nec-ejb-jar.xml の webservice-endpoint 要素に transport-guarantee 要素を追加します。値は INTEGRAL か CONFIDENTIAL にします。

```
<webservice-endpoint>
  ～
  <transport-guarantee>INTEGRAL or CONFIDENTIAL</transport-guarantee>
```

```
</webservice-endpoint>
```

（12）Web サービスクライアント実行時に指定する Web サービスの URL に含まれるホスト名、ポート番号がアクセスするサーバと一致していることを確認します。

SSL 通信をおこなうとき、WebOTX Developer が生成する Web サービスプロジェクトがデフォルトで指定する URL のままでは動作しません。Web サービスプロジェクト内に作成される Main クラスで実行する場合は第一引数に正しい URL を指定します。

WebOTX Webサーバを使う場合

WebOTX Web サーバで SSL を利用する手順を説明します。

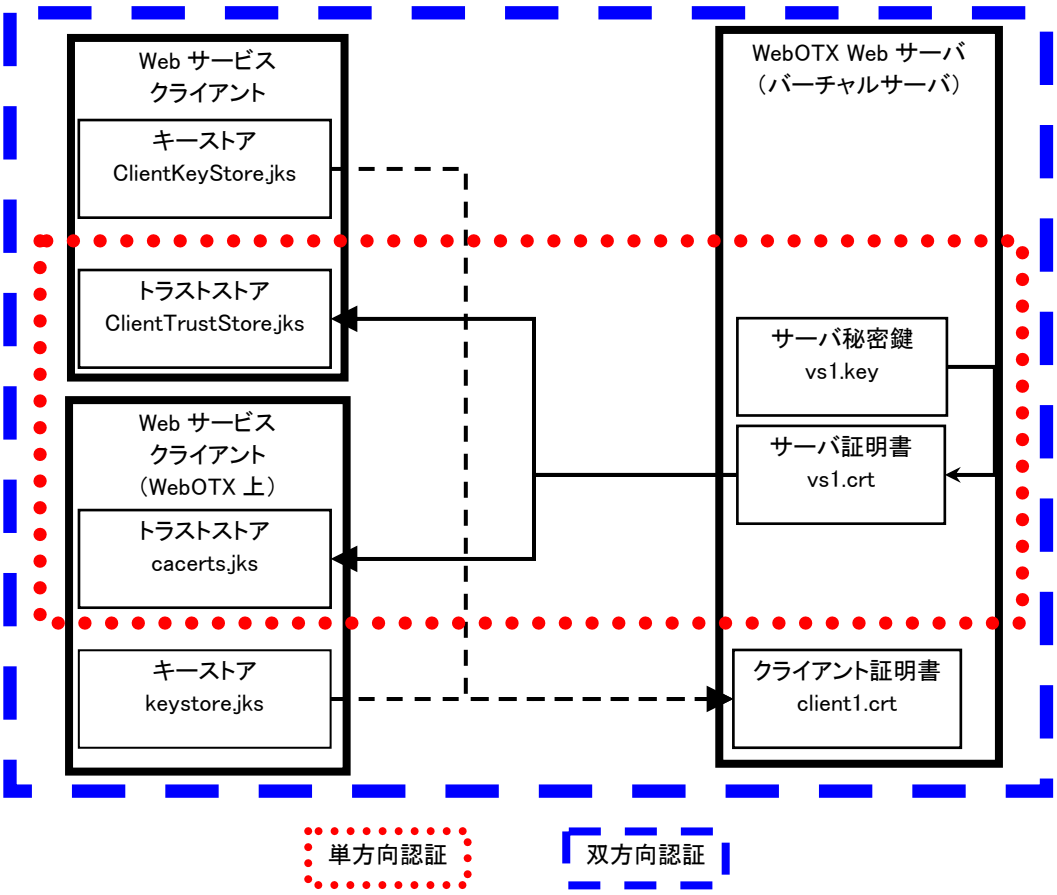


図 2.1.10d

■単方向認証の場合の手順

(1)WebOTX Web サーバの SSL 機能を有効にします。詳細については、WebOTX 共通マニュアル － 運用編 － 運用と操作 －WebOTX Web サーバ運用ガイドを参照してください。ここでは、otxadmin コマンドを使った設定例を示します。

```
[コマンド例]
>otxadmin

otxadmin>login --user admin --password adminadmin --port 6212

otxadmin>set server.WebServer.security-enabled=true
```

login コマンドの--port オプションは SSL を有効にしたい Web サーバと連動しているドメインに割り当てられたポート番号を指定します。

(2)任意のマシンに Open SSL をインストールします。

Open SSL については「http://www.openssl.org/」を参照してください。

MEMO

コマンド例で複数のコマンドがある場合は間隔をあけてあります。間隔があいていない場合、複数行になっていても 1 行で実行してください。

(3)openssl コマンドを使ってサーバ側の秘密鍵を作成します。秘密鍵は一つのバーチャルサーバにつき一つ作成します。

[コマンド例]
>openssl genrsa -rand 1.gif:2.jpg:3.gif 1024 >vs1.key

(4)秘密鍵から自己証明書を作成します。コマンド実行後、いくつか聞かれる質問のうち「Common Name (eg, YOUR name) []:」には、必ずバーチャルサーバの FQDN または IP アドレスを指定しなければなりません。

[コマンド例]
>openssl req -new -x509 -days 365 -key vs1.key >vs1.crt

(5)Web サービスクライアントが Java アプリケーションの場合、J2SE 付属の keytool コマンドを使ってクライアント用のトラストストアを作成します。ただし、Web サービスクライアントを WebOTX 上で動作させる場合、および、Web サービスクライアントが Web ブラウザの場合、この作業は必要ありません。

[コマンド例]
>keytool -genkey -alias client -dname "CN=client_fqdn, O=NEC, L=TOKYO, C=JP, OU=X, S=XY" -keyalg RSA -keypass changeit -storepass changeit -keystore ClientTrustStore.jks

-dname オプションの CN にはクライアントの FQDN または IP アドレスを指定します。

-keyalg、-keypass、-storepass で指定する値は適宜変更してください。

-keystore で指定している値は、図 2.1.10d に沿ったものです。

(6)サーバの証明書をクライアントにインポートします。トラストストアにインポートする場合は、J2SE 付属の keytool コマンドを使用します。コマンド実行後、「この証明書を信頼しますか？」という質問には「Y」と答えます。ただし、Web サービスクライアントを WebOTX 上で動作させる場合は、WebOTX のトラストストア(cacerts.jks)にインポートします。また、Web ブラウザにインポートする場合は、Web ブラウザの証明書インポート機能を使います。

[コマンド例:自作のトラストストアにインポートする場合]
>keytool -import -trustcacerts -alias server -file vs1.crt -keystore ClientTrustStore.jks -keypass changeit -storepass changeit

[コマンド例:WebOTX のトラストストアにインポートする場合]
>keytool -import -trustcacerts -alias server -file vs1.crt -keystore <Domain_ROOT>/config/cacerts.jks -keypass changeit -storepass changeit

-keypass、-storepass で指定する値は適宜変更してください。

自作のトラストストアにインポートする場合に-keystore で指定している値は、図 2.1.10d に沿ったものです。

<Domain_ROOT>は(1)で SSL を有効にした Web サーバと連動しているドメインがインストールされたルートフォルダ(ルートディレクトリ)です。

(7)<Domain_ROOT>/config/WebServer/ssl.conf をテキストエディタで開き、ServerName パラメータの値にバーチャルサーバのホスト名または IP アドレスとポート番号を、SSLCertificateKeyFile パラメータの値

にサーバの秘密鍵ファイルの絶対パスを、SSLCertificateFile パラメータの値にサーバの証明書の絶対パスを指定します。

[パラメータ設定例]
ServerName vs1.nec.com:443
SSLCertificateKeyFile "C:/key/vs1.key"
SSLCertificateFile "C:/crt/vs1.crt"

これらのパラメータは、バーチャルサーバーつにつき一つのみ設定できます。一つのバーチャルサーバに対して複数設定しないように注意してください。

<Domain_ROOT>は(1)で SSL を有効にした Web サーバと連動しているドメインがインストールされたルートフォルダ(ルートディレクトリ)です。

(8)Web サービスクライアントが Java アプリケーションの場合、次のコードを追加します。このコードはコンストラクタなどのインスタンス生成直後に実行される場所に追加し、必ずサーバ呼び出し処理の前に実行されるようにします。javax.net.ssl.trustStore キーの値はトラストストアのパスです。Web サービスクライアントを実行する Java VM のユーザホームディレクトリからの相対パス、または絶対パスで指定します。ただし、Web サービスクライアントを WebOTX 上で動作させる場合は、javax.net.ssl.trustStore キーは設定しないでください。

[コード例]
java.security.Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
System.setProperty("java.protocol.handler.pkgs","com.sun.net.ssl.internal.www.protocol");
System.setProperty("javax.net.ssl.trustStore", "C:/keystore/ClientTrustStore.jks");

[コード例:WebOTX 上で動作させる場合]
java.security.Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
System.setProperty("java.protocol.handler.pkgs","com.sun.net.ssl.internal.www.protocol")

(9)Web サービスクライアント実行時に指定する Web サービスの URL を次のことに注意して変更します。

- 「http://～」を「https://～」に変更します。
- ホスト名または IP アドレスを指定します。ホスト名が「localhost」では動作しません。
- バーチャルサーバの設定で SSL のポート番号を指定している場合、そのポート番号を指定します。

(10)(1)で SSL を有効にした Web サーバと連動しているドメインを再起動します。

■双方向認証の場合の手順

(1)単方向認証の場合の手順(1)～(9)を行います。

(2)クライアントの自己証明書を作成します。ここでは J2SE の keytool コマンドを使用します。クライアントの秘密鍵を含んだキーストアを作成し、キーストアから自己証明書をエクスポートします。ただし、Web サービスクライアントを WebOTX 上で動作させる場合は、WebOTX のキーストア(keystore.jks)にキーペアを追加します。また、Web サービスクライアントが Web ブラウザの場合、ここで作成したキーストアを Web ブラウザにインポートします。

[コマンド例]
>keytool -genkey -alias client1 -dname "CN=client_fqdn, O=NEC, L=TOKYO, C=JP, OU=X, S=XY" -keyalg RSA -keypass changeit -storepass changeit -keystore clientKeyStore.jks

> keytool -export -rfc -alias client1 -storepass changeit -file client1.crt -keystore clientKeystore.jks

[コマンド例：WebOTX 上で動作させる場合]
>keytool -genkey -alias client1 -dname "CN=client_fqdn, O=NEC, L=TOKYO, C=JP, OU=X, S=XY" -keyalg RSA -keypass changeit -storepass changeit -keystore <Domain_ROOT>/config/keystore.jks

>keytool -export -rfc -alias client1 -storepass changeit -file client1.crt -keystore <Domain_ROOT>/config/keystore.jks

-dname オプションの CN にはクライアントの FQDN または IP アドレスを指定します。

-keyalg、-keypass、-storepass で指定する値は適宜変更してください。

キーストアを作成する場合、-keystore で指定している値は、図 2.1.10dに沿ったものです。

Web サービスクライアントが Java アプリケーションの場合、ここで作成したキーストアを使いますので、削除しないでください。

WebOTX のキーストアは「<Domain_ROOT>/config/keystore.jks」です。

keystore.jks のキーストアパスワードの初期値は「changeit」です。

<Domain_ROOT>は（1）で SSL を有効にした Web サーバと連動しているドメインがインストールされたルートフォルダ（ルートディレクトリ）です。

Web ブラウザの場合、キーストアを pkcs#12 形式にしないとインポートできないことが多いことに注意してください。キーストア作成の際に「-storetype」オプションを追加して「pkcs12」を指定すると、キーストアは pkcs#12 形式になります。

（3）Web サービスクライアントが Java アプリケーションの場合、次のコードを追加します。このコードはコンストラクタなどのインスタンス生成直後に実行される場所に追加し、必ずサーバ呼び出し処理の前に実行されるようにします。javax.net.ssl.keyStore キーの値はキーストアのパスです。Web サービスクライアントを実行する Java VM のユーザホームディレクトリからの相対パス、または絶対パスで指定します。ただし、Web サービスクライアントを WebOTX 上で動作させる場合は、javax.net.ssl.keyStore キーは設定しないでください。また、javax.net.ssl.keyStorePassword キーの値はキーストアのパスワードに合わせて下さい。

[コード例]
System.setProperty("javax.net.ssl.keyStore", "C:/keystore/clientKeystore.jks");
System.setProperty("javax.net.ssl.keyStorePassword", "changeit");

[コード例：WebOTX 上で動作させる場合]
System.setProperty("javax.net.ssl.keyStorePassword", "changeit");

WebOTX のキーストア「<Domain_ROOT>/config/keystore.jks」のキーストアパスワードの初期値は「changeit」です。

<Domain_ROOT>は（1）で SSL を有効にした Web サーバと連動しているドメインがインストールされたルートフォルダ（ルートディレクトリ）です。

（4）<Domain_ROOT>/config/WebServer/ssl.conf をテキストエディタで開き、SSLCACertificateFile パラメータか SSLCACertificatePath パラメータのどちらか一つを変更して Web サーバにクライアントの証明書を認識させます。クライアントの証明書が一つのときは SSLCACertificateFile パラメータで証明書ファイルの絶対パスを指定します。クライアントの証明書が複数のときは SSLCACertificatePath パラメータ

の値を「require」変更します。

[パラメータ設定例：クライアントの証明書が一つのとき]
SSLCACertificateFile "C:/crt/client1.crt"
SSLVerifyClient require

[パラメータ設定例：クライアントの証明書が複数のとき]
SSLCACertificatePath "C:/crts"
SSLVerifyClient require

SSLCACertificateFile パラメータと SSLCACertificatePath パラメータは、バーチャルサーバー一つにつきどちらか一つのみ設定できます。両方同時に設定しないように注意してください。

これらのパラメータは、バーチャルサーバー一つにつき一つのみ設定できます。一つのバーチャルサーバに対して複数設定しないように注意してください。

（5）（1）で SSL を有効にした Web サーバと連動しているドメインを再起動します。

Webサービスセキュリティの使用方法

Webサービス作成ウィザードを使った場合の手順の流れ

Web サービス作成ウィザードを使って Web サービスを作成し、Web サービスセキュリティの機能を利用する場合、次のような手順で行います。

（1）初期設定

J2SE に対する初期設定を行います。

（2）キーストア、鍵、証明書を作成

署名、暗号化を利用する時は、それに必要な鍵や証明書を含んだキーストアをあらかじめ作成しておく必要があります。キーストアの作成は、J2SE の keytool コマンドなどを使用することができます。共通鍵（対称鍵）については J2SE の keytool コマンドでは生成できないので、後述する「xkeytool」を使用します。

（3）署名構成、暗号化構成の設定

WebOTX Developer のメニューからウィンドウ｜設定で設定ダイアログを起動し、WebOTX｜Web サービスの下にある署名構成、暗号化構成を設定します。

（4）Web サービス作成ウィザード

Web サービス作成ウィザードで Web サービスを作成します。後半の詳細設定画面よりあとに Web サービスセキュリティに関する設定画面があります。

（5）CallbackHandler の実装

Web サービス作成ウィザードにより作成された各種 CallbackHandler の雛形に、ユーザ名（またはエイリアス）、パスワード、SAML Assertion を取得するロジックを実装します。

（6）アーカイブ、配備

最後に、セキュリティ機能を利用しない Web サービスと同様に、アーカイブ、配備を行います。

（7）セキュリティ関連ファイルの配置

キーストア、証明書、CRL をドメインに配置します。

SAAJなどのAPIをつかって直接SOAPメッセージを処理する場合の手順の流れ

Web サービス作成ウィザードを使わずに SOAP メッセージを直接 XML 関係の API を使って扱う Web サービスならびに Web サービスクライアントを作成する場合、次のような手順で行います。

（1）疎通確認

セキュリティを適用していない状態の Web サービスクライアント～Web サービスでただしく通信できていることを確認します。

（2）初期設定

J2SE に対する初期設定を行います。

■Web サービスセキュリティモジュール動作に必要なライブラリ

Web サービスクライアントなどを作成する際、WebOTX 以外のクラスローダ環境で Web サービスセキュリティを動作させる場合に必要なライブラリは次の通りです。これらをクラスパスに追加してください。なお、WebOTX Application Server 上で動作させる場合には、これらを意識する必要はありません。

- <WebOTX_DIR>/lib/endorsed/resolver.jar
- <WebOTX_DIR>/lib/endorsed/serializer.jar
- <WebOTX_DIR>/lib/endorsed/xalan.jar
- <WebOTX_DIR>/lib/endorsed/xercesImpl.jar
- <WebOTX_DIR>/lib/endorsed/xml-apis.jar
- <WebOTX_DIR>/lib/activation.jar
- <WebOTX_DIR>/lib/commons-pack.jar
- <WebOTX_DIR>/lib/dsig.jar

- <WebOTX_DIR>/lib/j2ee.jar
- <WebOTX_DIR>/lib/log4j.jar
- <WebOTX_DIR>/lib/mail.jar
- <WebOTX_DIR>/lib/saaj-impl.jar （SAAJ や JAX-RPC を使用する場合のみ必要）
- <WebOTX_DIR>/lib/saml11.jar （SAML を使用する場合のみ必要）
- <WebOTX_DIR>/lib/wss4j.jar
- <WebOTX_DIR>/lib/xenc.jar
- <WebOTX_DIR>/lib/xsdlib.jar
- <WebOTX_DIR>/lib/xsutil.jar
- bcprov-jdk14-126.jar （JDK1.4 を使用している場合）

（3）キーストア、鍵、証明書を作成

署名、暗号化を行う時は、それに必要な鍵や証明書を含んだキーストアをあらかじめ作成しておく必要があります。キーストアの作成は、J2SE の keytool コマンドなどを使用することができます。共通鍵（対称鍵）については J2SE の keytool コマンドでは生成できないので、後述する「xkeytool」を使用します。

（4）ハンドラクラスを作成

ハンドラクラスを SOAP メッセージ送信用、受信用としてそれぞれ必要なものを作成し、Web サービスクライアント、Web サービスの本来のメッセージ処理が始まる前の部分に組み込みます。

（5）キーストア情報設定ファイルを作成

キーストア情報設定ファイルを送信用、受信用としてそれぞれ必要なものを作成し、Web サービスクライアント、Web サービスのクラスパスが通った場所に配置します。

（6）CallbackHandler を実装します。

ユーザ名（またはエイリアス）、パスワード、SAML Assertion を取得するロジックを含んだ各種 CallbackHandler を、送信側、受信側それぞれに必要なものを実装します。

（7）アーカイブ、配備

アーカイブや配備は、それぞれのアプリケーションの形態に合わせて行います。

（8）セキュリティ関連ファイルの配置

キーストア、証明書、CRL を所定の場所に配置します。

Webサービスセキュリティの初期設定

Web サービスセキュリティを利用する場合は、全ての開発を行う前に次の準備を行う必要があります。

Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files

Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files をお使いの J2SE に対してインストールする必要があります。WebOTX Application Server の媒体から、お使いの J2SE のバージョンに合わせて、「jce_policy-1.4.2.zip」または「jce_policy-1_5_0.zip」を取り出し、そのファイルを解凍して得られたファイルを<J2SE_HOME>/jre/lib/security に置かれている local_policy.jar、US_export_policy.jar の 2 ファイルに上書き置換してください。

Bouncy CastleJCE セキュリティプロバイダの入手

Bouncy CastleJCE セキュリティプロバイダの JAR ファイルを<WebOTX_DIR>/lib/ 配下にコピーします。この JAR ファイルは WebOTX Application Server の媒体に含まれています。

J2SE 1.4 をお使いの場合は、「bcprov-jdk14-126.jar」をお使いください。J2SE 5.0 をお使いの場合は、「bcprov-jdk15-136.jar」をお使いください。テスト用サーバをインストールしている時は、コピーの後、WebOTXAdmin ドメインおよび全ドメインを再起動してください。

Bouncy Castle JCE セキュリティプロバイダの登録

<J2SE_HOME>/jre/lib/security にある java.security に Bouncy Castle JCE セキュリティプロバイダを次のように登録してください。

J2SE 1.4 の場合

```
#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
security.provider.3=com.sun.rsajca.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=org.bouncycastle.jce.provider.BouncyCastleProvider
security.provider.6=sun.security.jgss.SunProvider
```

J2SE 5.0 の場合

```
#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=com.sun.security.sasl.Provider
security.provider.7=org.bouncycastle.jce.provider.BouncyCastleProvider
```

これらの作業をすべて終了したら、ドメインを(再)起動してください。

MEMO

<J2SE_HOME>は J2SE のインストールルートディレクトリを指します。

MEMO

<WebOTX_DIR>は WebOTX のインストールルートディレクトリを指します。

xkeytool

XKeyTool は、J2SE が提供する keytool のラッパーとして機能し、keytool と同じコマンドをサポートするユーティリティです。J2SE の keytool の機能に加えて、共通鍵（対称鍵）の生成にも対応します。

環境構築

xkeytool を実行するためには、以下の JAR ファイルをクラスパスに追加する必要があります。bcprov-jdk14-126.jar は WebOTX メディアの 2 枚目に含まれていますので、事前にコピーしておきます。事前準備についてはアプリケーション開発ガイドの 2 章を参照してください。

- <WebOTX_DIR>¥lib¥bcprov-jdk14-126.jar
- <WebOTX_DIR>¥lib¥log4j.jar
- <WebOTX_DIR>¥lib¥xenc.jar
- <WebOTX_DIR>¥lib¥xsutil.jar

MEMO

<WebOTX_DIR>は WebOTX のインストールルートディレクトリのことです。

コマンド

xkeytool 実行時に次のコマンドを1つだけ指定します。指定できるコマンドは次の通りです。

コマンド	説明
-certreq	証明書署名要求（CSR）を PKCS#10 形式で生成します。
-delete	キーストアからエントリを削除します。
-export	証明書及び鍵をファイルにエクスポートします。
-genkey	鍵ペアとその公開鍵の X.509 バージョン 1 自己署名証明書を生成します。
-help	キーストアコマンドに関するヘルプ情報を提供します。
-identitydb	JDK1.1 アイデンティティデータベースをキーストアにインポートします。
-import	証明書、証明書連鎖または、鍵をキーストアにインポートします。
-keyclone	新しいエイリアスで鍵エントリのコピーを生成します。
-keypasswd	鍵エントリの非公開鍵に関連付けられたパスワードを変更します。
-list	キーストアの内容を表示します。
-printcert	ファイル内、または標準入力ストリーム(stdin)からの証明書を印刷します。
-selfcert	X.509 バージョン 1 自己署名証明書を生成します。
-storepasswd	キーストアの保全性を保護するために使用されるパスワードを設定します。

オプション

xkeytool 実行時に次のオプションを複数指定することができます。各コマンドで指定できるオプションは決まっています。また、オプション同士に並びの制約はありません。

オプション	説明
-alias [alias]	コマンドを適用するエントリのエイリアスを指定します。
-dest [dest_alias]	コマンドの適用先であるエイリアスを指定します。
-dname [dname]	エイリアスに関連付けられる X.509 識別名を指定します。
-file [filename]	コマンドで使用するファイル名を指定します。
-keyalg [keyalg]	鍵生成アルゴリズムの名前を指定します。
-keypass [keypass]	鍵エントリの非公開鍵を保護するのに使用されるパスワードを指定します。

-keysize [keysize]	生成する鍵長を指定します。
-keystore [keystore]	使用するキーストアファイルを指定します。
-new [new_keypass]	鍵エントリの非公開鍵を保護するのに使用される新しいパスワードを指定します。
-noprompt	プログラムとユーザの対話を切断します。
-rfc	RFC1421 印刷可能エンコーディング形式の使用を指定します。
-sigalg [sigalg]	使用する署名アルゴリズムを指定します。この値は下位の非公開鍵のアルゴリズムから取得します。下位の非公開鍵が DSA タイプの場合、このオプションはデフォルトで SHA1withDSA となります。非公開鍵が RSA タイプの場合、このオプションはデフォルトで MD5withRSA となります。
-storepass [storepass]	キーストアの保全性を保護するために使用されるパスワードを指定します。パスワードは 6 文字以上でなければならなりません。パスワードは全てのコマンドで使用されなければなりません。
-storetype [storetype]	使用するキーストアのタイプを指定します。デフォルト値は java.home プロパティで指定されるディレクトリの/lib/security にある java.security ファイルの keystore.type プロパティで指定します。この値は java.security.KeyStore です。
-trustcacerts cacerts	キーストアファイルの証明書を鍵ストアの証明書に追加して使用するよう指定します。cacerts ファイルは java.home の /lib/security サブディレクトリにあります。
-v	詳細出力モードを使用します。
-validity [valDays]	証明書の有効日数を指定します。

オプションの初期値は次のようになっています。

コマンド	説明
-alias	mykey
-keyalg	DSA
-keysize	1024
-keystore	ユーザのホームディレクトリにある「.keystore」

コマンドに対応する指定可能なオプションは次の通りです。

コマンド	オプション	-alias	-dest	-dname	-file	-keyalg	-keypass
-certreq		○			○		○
-delete		○					
-export		○			○		※4
-genkey		○		※1		※2	○
-help							
-identitydb					○		
-import		○			○		○
-keyclone		○	○				○
-keypasswd		○					○
-list		○					
-printcert					○		
-selfcert		○		○			○
-storepasswd							

コマンド	オプション	-keysize	-trustcerts	-v	-keystore	-new	-noprompt
-certreq				○	○		
-delete				○	○		
-export		○		○	○		
-genkey		○		○	○	○	
-help							
-identitydb				○	○		
-import			※1	○	○		※1
-keyclone				○	○	○	
-keypasswd				○	○	○	
-list				○	○		
-printcert				○			
-selfcert				○	○		
-storepasswd				○	○	○	

コマンド	オプション	-rfc	-sigalg	-storepass	-storetype	-validity
-certreq			○	○	○	
-delete				○	○	
-export		※1		○	○	
-genkey			※1	○	※3	※1
-help						
-identitydb				○	○	
-import				○	○	
-keyclone				○	○	
-keypasswd				○	○	
-list		○		○	○	
-printcert						
-selfcert			○	○	○	○
-storepasswd				○	○	

※1：共通鍵に対する操作の場合は不要(または無効)です。

※2：共通鍵を生成する場合は以下のいずれかを指定する必要があります。

「TripleDES」、「DESede」、「AES」、「RIJNDAEL」、「ARCFOUR」、「RC4」

※3：共通鍵を生成する場合は JCEKS である必要があります。

※4：共通鍵をエクスポートする場合は必要です。

実行方法

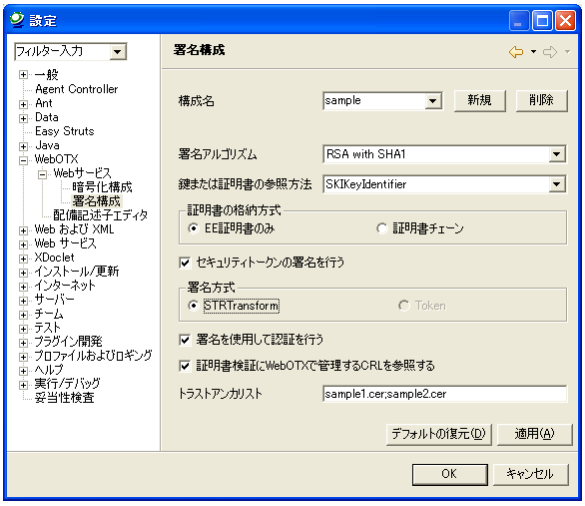
java -Djava.endorsed.dirs="<WebOTX_DIR>¥lib¥endorsed" com.nec.jp.xml.xenc.tool.XKeyTool <コマンド> <オプション>

実行例

```
java -Djava.endorsed.dirs="<WebOTX_DIR>¥lib¥endorsed" com.nec.jp.xml.xenc.tool.XKeyTool
 -genkey -keystore sample.jceks -alias sample -keyalg AES -keysize 128 -storetype jceks
```

署名構成

WebOTX Developer における署名構成の設定について説明します。この操作は、Web サービス作成ウィザードで Web サービスセキュリティを使う場合に必要です。



WebOTX では、署名についての詳細設定は、「構成」という単位で管理します。構成にはそれぞれ名前をつけて管理します。[新規]ボタンを押すと、新規に構成を作成します。一通り設定を行ったあと、[適用]ボタン、または[OK]ボタンを押すと今表示している構成が保存されます。[削除]ボタンを押すと、今表示している構成を削除します。

■構成名

署名構成の名前を英数文字で設定します。コンボボックスの中にカーソルを合わせると構成名を変更することができます。

■署名アルゴリズム

使用する署名アルゴリズムを選択します。RSA with SHA1、DSA with SHA1 は公開鍵暗号方式、Hmac-SHA1 は共通鍵暗号方式です。

■鍵または証明書の参照方法

署名検証に使用する公開鍵証明書または鍵の参照方法を指定します。

選択肢	説明
DirectReference	メッセージの内部あるいは外部にある証明書を URI で参照します。
IssuerSerial	メッセージの外部にある証明書を、発行者とシリアル番号をキーにして参照します。
SKIKeyIdentifier	メッセージの外部にある証明書を、所有者鍵識別子をキーにして参照します。
EmbeddedReference	メッセージの内部にある証明書を参照します。
EmbeddedKeyName	鍵をエイリアスをキーにして参照します。この方法は署名アルゴリズムで HMac-SHA1 を選択すると選択することができます。

■証明書の格納方式

署名に使用した秘密鍵とペアになる公開鍵証明書の BinarySecurityToken への格納形態を指定します。証明書チェーンを選択すると、EE 証明書と CA 証明書を格納します。この設定は、署名アルゴリズムが公開鍵暗号方式（RSAwithSHA1 または DSAwithSHA1）でかつ、証明書の参照方法が DirectReference または EmbeddedReference の場合にのみ有効になります。

■セキュリティトークンの署名を行う/署名方式

署名につけられるセキュリティトークンを署名する場合、チェックします。署名方式は、セキュリティトークンの署名方式です。STRTransform は、セキュリティトークンを wsse:SecurityTokenReference 要素で参照し、STR Dereference Transform を使用して署名します。Token は、セキュリティトークンを

ds:Reference 要素で参照し、署名します。

■署名を使用して認証を行う

署名を使用して認証を行う場合にチェックします。SOAP メッセージに付与されている通常の署名を検証するのに加えて、署名検証に使用した公開鍵証明書の証明書パスを検証し、送信者の正当性を確認します。

■証明書検証に WebOTX で管理する CRL を参照する

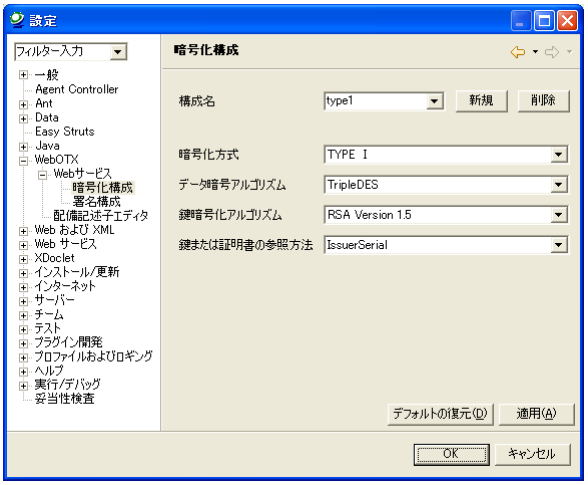
署名を使用した認証において、公開鍵証明書の証明書パスの検証の際に、WebOTX のドメイン単位で管理する CRL を参照するときはチェックします。

■トラストアンカリスト

公開鍵証明書検証で信頼済みとして扱われるトラストアンカ（ルート CA）を指定します。トラストアンカ証明書のファイル名を記述します。複数の時は、セミコロンで区切ります。

暗号化構成

WebOTX Developer's Studio における暗号化構成の設定について説明します。この操作は、Web サービス作成ウィザードで Web サービスセキュリティを使う場合に必要です。



■構成名

暗号化構成の名前を英数文字で設定します。コンボボックスの中にカーソルを合わせると構成名を変更することができます。

■暗号化方式

3 つの方式については、次の通りです。

・TYPE Ⅰ

ランダムに生成した対称鍵（セッション鍵）を用いてメッセージを暗号化し、暗号化に使用した対称鍵を公開鍵を用いて暗号化する。

・TYPE Ⅱ

ランダムに生成した対称鍵（セッション鍵）を用いてメッセージを暗号化し、暗号化に使用した対称鍵をあらかじめ暗号化側と復号化側で既知である対称鍵を用いて暗号化する。

・TYPE Ⅲ

あらかじめ暗号化側と復号化側で既知である対称鍵を用いて、メッセージを暗号化する。

■データ暗号化アルゴリズム

データの暗号化に使用する暗号化アルゴリズムを選択します。いずれも共通鍵暗号方式です。

■鍵暗号化アルゴリズム

鍵の暗号化に使用する暗号化アルゴリズムを選択します。いずれも公開鍵暗号方式です。

■鍵または証明書の参照方法

暗号化に使用した鍵または証明書の参照方法を指定します。

選択肢	説明
DirectReference	メッセージの内部あるいは外部にある証明書を URI で参照します。
IssuerSerial	メッセージの外部にある証明書を、発行者とシリアル番号をキーにして参照します。
SKIKeyIdentifier	メッセージの外部にある証明書を、所有者鍵識別子をキーにして参照します。
EmbeddedReference	メッセージの内部にある証明書を参照します。
EmbeddedKeyName	鍵をエイリアスをキーにして参照します。暗号化方式が TYPE Ⅱ、Ⅲの時に自動的に選択されます。

CallbackHandlerの実装

Client/ServerSenderCallbackHandler

メッセージの送信側でユーザ名（エイリアス）、パスワードを設定するためのクラスです。クラス名の最初の「Client」、「Server」は、このクラスがクライアントとサーバのどちらで働くかを表すためにWebサービス作成ウィザードが自動的に付加します。また、「TODO:」というコメントのあとに、ユーザネームトークンのユーザ名・パスワードの取得、署名付与に使用する鍵のエイリアス・パスワードの取得、暗号化に使用する鍵のエイリアス・パスワードの取得ができるように実装するだけで済みます。

SenderCallbackHandler の例

```
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import java.io.IOException;
import org.apache.ws.security.WSUserInfoCallback;

public class ClientSenderCallbackHandler implements CallbackHandler {

   public void handle(Callback[] callbacks) throws IOException , UnsupportedCallbackException {
      for (int i = 0; i < callbacks.length; i++) {
         if (callbacks[i] instanceof WSUserInfoCallback) {
              WSUserInfoCallback pc = (WSUserInfoCallback) callbacks[i];
              int usage = pc.getUsage();
              String username = null;
              String password = null;

               // TODO:
               // here call a function/method to lookup the password for
               // the given identifier

                switch (usage) {
                case WSUserInfoCallback.USERNAME_TOKEN:
                   username = "foo1";
                   password = "var1";
                   break;
                case WSUserInfoCallback.SIGNATURE:
                   username = "foo2";
                   password = "var2";
                   break;
                case WSUserInfoCallback.ENCRYPT:
                   username = "foo3";
                   password = "var3";
                   break;
                default:
                   break;
                }

              pc.setUsername(username);
              pc.setPassword(password);
         } else {
            throw new UnsupportedCallbackException(callbacks[i], "Unrecognized Callback");
         }
      }
   }
}
```

Client/ServerReceiverCallbackHandler

メッセージの受信側でパスワードを設定するためのクラスです。クラス名の最初の Client、Server は、このクラスがクライアントとサーバのどちらで働くかを表すために Web サービス作成ウィザードが自動的に付加します。また、「TODO:」というコメントのあとに、ユーザネームトークンのパスワードの取得、署名検証に使用する鍵のパスワードの取得、復号化に使用する鍵のパスワードの取得ができるように実装するだけで済みます。

ReceiverCallbackHandler の例

```
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import java.io.IOException;
import org.apache.ws.security.WSPasswordCallback;

public class ServerReceiverCallbackHandler implements CallbackHandler {

   public void handle(Callback[] callbacks) throws IOException , UnsupportedCallbackException {
      for (int i = 0; i < callbacks.length; i++) {
         if (callbacks[i] instanceof WSPasswordCallback) {
            WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
            int usage = pc.getUsage();
            String identifier = pc.getIdentifier();
            String password = null;

            // TODO:
            // here call a function/method to lookup the password for
            // the given identifier

            switch (usage) {
            case WSPasswordCallback.USERNAME_TOKEN:
               password = "";
               break;
            case WSPasswordCallback.SIGNATURE:
               password = "";
               break;
            case WSPasswordCallback.DECRYPT:
               password = "";
               break;
            default:
               break;
            }

            pc.setPassword(password);
         } else {
            throw new UnsupportedCallbackException(callbacks[i], "Unrecognized Callback");
         }
      }
   }
}
```

KeystoreCallbackHandler

キーストアのパスワードを取得するためのクラスです。Web サービス作成ウィザードが生成した雛形の場合、生成された雛形の「TODO:」というコメントのあとに、各キーストアファイルごとのパスワードが取得できるように実装すれば済みます。また、このクラスはクライアント側、サーバ側で共通して使用することを前提として生成しますが、送信側と受信側で別々にすることもできます。その場合は、キーストア情報設定ファイルを送信側と受信側で別々にし、それぞれに設定を行うようにします。

KeystoreCallbackHandler の例

```
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import java.io.IOException;
import org.apache.ws.security.WSKeyStorePasswordCallback;

public class KeystoreCallbackHandler implements CallbackHandler {

   public void handle(Callback[] callbacks) throws IOException , UnsupportedCallbackException {
      for (int i = 0; i < callbacks.length; i++) {
         if (callbacks[i] instanceof WSKeyStorePasswordCallback) {
            WSKeyStorePasswordCallback pc = (WSKeyStorePasswordCallback) callbacks[i];

            // keystore's file name
            String identifier = pc.getIdentifier();
            String password = null;

            // TODO:
            // here call a function/method to lookup the password for
            // the given identifier

            if (identifier.equals("keystore/store1.jks")) {
               password = "foo";
            }else if(identifier.equals("keystore/store2.jks")) {
               password = "var";
            }

            pc.setPassword(password);
         } else {
            throw new UnsupportedCallbackException(callbacks[i], "Unrecognized Callback");
         }
      }
   }
}
```

※キーストアファイルは「keystore/<キーストアファイル名>」で指定してください。今後の説明は、その設定に沿って行います。

Client/ServerSAMLAssertionCallbackHandler

メッセージの送信側で SAMLAssertion を取得するためのクラスです。クラス名の最初の Client、Server は、このクラスがクライアントとサーバのどちらで働くかを表すために Web サービス作成ウィザードが自動的に付加します。ここでは、WebSAM SECUREMASTER から取得した Assertion を、assertion 変数に代入します。

SAMLAssertionCallbackHandler の例

```
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import java.io.FileReader;
import java.io.IOException;
import org.apache.ws.security.WSSAMLAssertionCallback;

public class ClientSAMLAssertionCallbackHandler implements CallbackHandler {

   public void handle(Callback[] callbacks) throws IOException , UnsupportedCallbackException {
      for (int i = 0; i < callbacks.length; i++) {
         if (callbacks[i] instanceof WSSAMLAssertionCallback) {
            WSSAMLAssertionCallback pc = (WSSAMLAssertionCallback) callbacks[i];
            String assertion = null;

            // TODO: Acquisition of SAML Assertion.

            String assnFile = "foo.xml";
            assertion = readFile( assnFile );
            pc.setAssertion(assertion);
         } else {
            throw new UnsupportedCallbackException(callbacks[i], "Unrecognized Callback");
         }
   }
}

   private String readFile(String fname) throws IOException {
      FileReader reader = new FileReader(fname);
      char[] buf = new char[1024];
      StringBuffer sbuf = new StringBuffer();
      int n;
      while((n = reader.read(buf)) != 0) {
         sbuf.append(new String(buf,0,n));
         if (n<buf.length) break;
      }
      return sbuf.toString();
   }
}
```

Web Services Security関連ファイルの配置

Web サービス作成ウィザードを使った場合、クライアント環境となる Web サービスプロジェクト内では、すでにセキュリティ関連ファイルは規定の場所に配置されて生成されます。しかし、サーバ環境や Web サービス作成ウィザードが作成したものとは違ったクライアント環境では、キーストアファイル、証明書、CRL を所定の場所に手動で配置する必要があります。次の場所は WebOTX Application Server で既定となっているディレクトリです。キーストア情報設定ファイル、ハンドラクラスの初期値で置き場所を明示的に変えた場合は、その場所に配置してください。

<domain-root>/config/keystore/ 配下 ・・・キーストアファイル

<domain-root>/config/cert/ 配下 ・・・証明書

<domain-root>/config/crl/ 配下 ・・・CRL

MEMO

<domain-root >は使用するドメインのルートディレクトリのことです。

ハンドラクラスを作成する

WebOTX Developer の Web サービス作成ウィザードを使わずに、SOAP を直接扱う Web サービスや Web サービスクライアントを作成するときには、ハンドラクラスを作成する必要があります。Web サービス作成ウィザードを使う場合は、ウィザードがハンドラクラスを自動的に生成するため、特にこの作業を行う必要はありません。

Web サービスセキュリティ機能は、JAX-RPC ハンドラとして実装されていますが、その内部では SAAJ で定義される javax.xml.soap.SOAPMessage を受け取り、SOAP メッセージに対して様々な Web サービスセキュリティの処理を行い、処理後の javax.xml.soap.SOAPMessage を返すという単純な仕組みになっています。そのため、JAX-RPC ハンドラ機構に似た構造を簡単に自作して、直接 Web サービスセキュリティモジュールを呼び出して使用することが可能です。

■実装方法

(1) 次のインタフェースを作成します。コンパイルした class ファイルをパッケージ単位で JAR ファイルにアーカイブします。WebOTX 上で使用する場合、使用する WebOTX のドメインの lib ディレクトリ(例:<WebOTX_DIR>/domains/domain1/lib)に配置し、対象のドメインを再起動します。WebOTX 以外の環境では、クラスパスに追加します。

※WebOTX Enterprise Service Bus をインストールしているドメインについては、この作業は不要です。

```
package com.nec.webotx.jbi.binding.soap;
public interface SoapMessageHandlerIF {
   public void init(java.util.Map initParam);
   public boolean handleFault(javax.xml.soap.SOAPMessage soapMessage, java.util.Map properties);
   public boolean handleToRouter(javax.xml.soap.SOAPMessage soapMessage, java.util.Map
properties);
   public boolean handleFromRouter(javax.xml.soap.SOAPMessage soapMessage, java.util.Map
properties);
}
```

(2) org.apache.ws.security.jbi.WssSoapMessageHandler クラスを new して呼び出します。

org.apache.ws.security.jbi.WssSoapMessageHandler は、com.nec.webotx.jbi.binding.soap.SoapMessageHandlerIF の実装クラスです。

(3) パラメータリファレンスを見ながら初期化パラメータと値のプロパティを作成し、org.apache.ws.security.jbi.WssSoapMessageHandler の init メソッドの引数に与えて init メソッドを実行します。

(4) handle～メソッドを実行します。handleToRouter は送信側から受信側へ流れるリクエストメッセージについてセキュリティ処理をするとき、handleFromRouter は受信側から送信側へ流れるレスポンスメッセージについてセキュリティ処理をするとき、handleFault はレスポンスメッセージが SOAP Fault の場合にセキュリティ処理をするとき、にそれぞれ呼び出されます。

第 1 引数は、セキュリティ処理をするべき SOAP メッセージ、第 2 引数はセキュリティ処理した結果を次のハンドラに受け渡すためのプロパティです。org.apache.ws.security.jbi.WssSoapMessageHandler クラスを単数呼び出せばよい場合、または複数のうちの最初に呼び出すインスタンスの場合、new java.util.HashMap() のようにして空のプロパティを作り、引数に渡してください。逆に、複数のうちの 2 番目以降に呼び出すハンドラに該当する場合、先に作っておいたインスタンスをそのまま引数に渡します。

実装イメージ

```
SOAPMessage soapMessage = セキュリティ処理対象の SOAP メッセージ;

SoapMessageHandlerIF handler1 = new org.apache.ws.security.jbi.WssSoapMessageHandler();
SoapMessageHandlerIF handler2 = new org.apache.ws.security.jbi.WssSoapMessageHandler();

Map initParam1 = new HashMap();
initParam1.put("パラメータ名", "値");
handler1.init(initParam1);

Map initParam2 = new HashMap();
initParam2.put("パラメータ名", "値");
handler2.init(initParam2);
```

```
Map properties = new HashMap();

handler1.handleToRouter(soapMessage, properties);
handler2.handleToRouter(soapMessage, properties);
```

※受信側のハンドラクラスは必ず単数です。送信側に個数の制限はありません。

※送信側で複数のハンドラが必要な場合、(2)～(4)の作業を繰り返します。

キーストア情報設定ファイルの作成法

キーストア情報設定ファイルとは、キーストアの情報について記述するプロパティファイルです。Web サービス作成ウィザードを使用した場合には自動的に作成されるので、この作業を行う必要はありません。

キーストア情報設定ファイルの例

```
org.apache.ws.security.crypto.merlin.keystore.passwordCallbackClass=sample.SampleKeystoreCallbackHandler
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.file=keystore/sample.jks
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
```

1 つ目は、送信側あるいは受信側でキーストアのエイリアスやパスワードを取得するためのキーストアコールバックハンドラのクラスを設定します。

2 つ目はキーストアの形式を設定します。「jks」「pkcs12」「jceks」のうちの 1 つを設定します。

3 つ目はキーストアファイルの場所を指定します。絶対パスでも、ユーザーホームディレクトリからのパスを指定しても構いません。WebOTX Application Server ではキーストアの置き場所として <DOMAIN_ROOT>/config/keystore を用意していますので、そこにキーストアファイルを置く場合は上記の例のように「keystore/～」と指定できます。

4 つ目は固定値です。上記の例の通り記述してください。

キーストア情報設定ファイルが完成したら、送信側、受信側ともクラスパスの通っているところに配置してください。

パラメータリファレンス

Web サービスセキュリティを WebOTX ESB や SOAP を直接扱う Web サービスや Web サービスクライアントで扱う場合、ハンドラクラスの初期化パラメータと、送り込む値の詳細を知らなければ Web サービスセキュリティを動作させることはできません。ここでは、目的別にパラメータ名と設定する値について説明します。

各機能に共通のパラメータ設定

ここでは、ハンドラクラス 1 つにつき必ず設定する各機能に共通するパラメータについて説明します。ここで示すパラメータと値は、各ハンドラに必ず与えなければなりません。

■送信側

パラメータ	値	説明
action （処理の種別を指定します。複数設定したい場合は、複数のハンドラクラスを使用します。）	Signature	署名機能、署名認証を使用するときに指定します。
	Encrypt	暗号化機能を称するときに指定します。
	UsernameToken	ID/Password 認証、WebOTX 認証を使用するときに指定します。
	Timestamp	タイムスタンプ付与機能を使用するときに指定します。
	SAMLTokenUnsigned	SAML Assertion 付与機能（署名を付与しない）を使用するときに指定します。
	SAMLTokenSigned	SAML Assertion 付与機能（署名を付与する）を使用するときに指定します。
	NoSerialization	複数の送信用ハンドラを使用する場合、最後に指定したハンドラ以外の場合にスペース区切りで追加指定します。
deployment （どのメッセージに対して処理をするかを識別します。）	server-response (inbound-response)	Web サービスからのレスポンスメッセージに対して処理を有効にしたい場合に指定します。 ESB の SOAP BC 場合、inbound からレスポンスメッセージに対して処理を有効にしたい場合に指定します。
	client-request (outbound-request)	Web サービスクライアントからのリクエストメッセージに対して処理を有効にしたい場合にしていします。 ESB の SOAP BC 場合、outbound からのリクエストメッセージに対して処理を有効にしたい場合に指定します。
actor （メッセージの受信者の識別子を指定します。）	<URI>	受信者の識別子を表す URI を指定します。
mustUnderstand （メッセージの受信者が Security ヘッダを必ず処理しなければならないかどうかを指定します。）	true	処理が必須の場合に指定します。
	false	処理が任意の場合に指定します。

■受信側

パラメータ	値	説明
action	Signature	署名検証機能、署名認証、署名付きの SAML Assertion の検証機能を使用するときに指定します。

（処理の種別を指定

指定します。）	Timestamp	タイムスタンプ検証機能を使用するときに指定します。
	SAMLTokenUnsigned	SAML Assertion 検証機能を使用するときに指定します。
deployment （どのメッセージに対して処理をするかを識別します。）	server-request (inbound-request)	Web サービスが受け取ったリクエストメッセージに対して処理を有効にしたい場合に指定します。 ESB の SOAP BC 場合、inbound のが受け取ったリクエストメッセージに対して処理を有効にしたい場合に指定します。
	client-response (outbound-response)	Web サービスクライアントが受け取ったレスポンスメッセージに対して処理を有効にしたい場合にしています。 ESB の SOAP BC 場合、outbound が受け取ったのレスポンスメッセージに対して処理を有効にしたい場合に指定します。
actor （メッセージの受信者の識別子を指定します。）	<URI>	受信者の識別子を表す URI を指定します。
mustUnderstand （メッセージの受信者が Security ヘッダを必ず処理しなければならないかどうかを指定します。）	true	処理は必須
	false	処理は任意

署名・署名検証関連のパラメータ設定

●RSA または DSA アルゴリズムを利用する

■送信側

パラメータ	値	説明
action	Signature	
signatureAlgorithm（署名アルゴリズムを指定します。）	http://www.w3.org/2000/09/xmldsig#dsa-sha1	DSAwithSHA1 アルゴリズムを使用する場合に指定します。
	http://www.w3.org/2000/09/xmldsig#rsa-sha1	RSAwithSHA1 アルゴリズムを使用する場合に指定します。
signaturePropFile	<キーストア情報設定ファイル名>	キーストア情報設定ファイルのパスを指定します。
signatureKeyIdentifier（署名検証に使用する公開鍵証明書または鍵の参照方法を指定します。）	DirectReference	メッセージの内部あるいは外部にある証明書を URI で参照します。
	IssuerSerial	メッセージの外部にある証明書を、発行者とシリアル番号をキーにして参照します。
	SKIKeyIdentifier	メッセージの外部にある証明書を、所有者鍵識別子をキーにして参照します。
	EmbeddedReference	メッセージの内部にある証明書を参照します。
useSingleCert（署名に使用した秘密鍵とペアになる公開鍵証明書の BinarySecurityToken への格納形態を指定します。）	true	BinarySecurityToken に EE 証明書のみを格納します。
	false	BinarySecurityToken に証明書チェーン(EE 証明書 + CA 証明書)を格納します。
userInfoCallbackClass	<コールバックハンドラクラス名>	署名に使用する鍵のエイリアスとパスワードを取得するためのコールバックハンドラクラス名を指定します。
signatureParts（署名対象を指定します。）	{}{XPath}{}	XPath には署名対象を表す XPath 式を指定します。
	{}{XPath}{};STRTransform	XPath には署名対象を表す XPath 式を指定します。 また、署名に関連付けられるセキュリティトークンを wsse:SecurityTokenReference 要素で参照し、STR Dereference Transform を使用して署名します。
	{}{XPath}{};Token	XPath には署名対象を表す XPath 式を指定します。 また、署名に関連付けられるセキュリティトークンを ds:Reference 要素で参照し、署名します。

MEMO

useSingleCert パラメータは、署名アルゴリズムが RSAwithSHA1 または DSAwithSHA1 かつ 証明書の参照方法が DirectReference または EmbeddedReference の場合にのみ使用されます。その他の場合は、指定しても無視されます。本パラメータを省略した場合、true として扱われます。

MEMO

signatureParts パラメータを省略した場合、署名対象は SOAP エンベロープの Body 要素になります。

XPath の書き方

パラメータの値としての XPath 式は正確な記述が求められます。ほとんどの場合、省略形は用いることができません。

（例）

{}{//*[namespace-uri()="http://schemas.xmlsoap.org/soap/envelope/" and local-name()="Envelope"]/*[namespace-uri()="http://schemas.xmlsoap.org/soap/envelope/" and local-name()="Body"]}{}

また、ハンドラへのパラメータと値の与え方によって、エスケープ文字を用いらなければならない場合もありますので、十分注意してください。たとえば、ESB の SOAP BC の endpoints.xml では、「"」を「"」で表記する必要があります。また、Java コード内で直接表記する場合は、「"」を「¥"」で表記する必要があります。

■受信側

パラメータ	値	説明
action	Signature	
signaturePropFile	<キーストア情報設定ファイル名>	署名検証用の公開鍵証明書が格納されているキーストアに関する情報を記述したファイル名を指定します。
passwordCallbackClass	<コールバックハンドラクラス名>	パスワードを取得するためのコールバックハンドラクラス名を指定します。

●HMAC アルゴリズムを利用する

■送信側

パラメータ	値	説明
action	Signature	
signatureAlgorithm（署名アルゴリズムを指定します）	http://www.w3.org/2000/09/xmldsig#hmac-sha1	HMAC-SHA1 アルゴリズムを設定します。
signaturePropFile	<キーストア情報設定ファイル名>	キーストア情報設定ファイルのパスを指定します。
signatureKeyIdentifier	EmbeddedKeyName	署名検証に使用する鍵の参照方法を指定します。鍵を、エイリアスをキーにして参照します。
userInfoCallbackClass	<コールバックハンドラクラス>	署名に使用する鍵のエイリアスとパスワードを取得するためのコールバックハンドラクラス名を指定します。
signatureParts	{}{XPath}{}	署名対象を指定します。

■受信側

パラメータ	値	説明
action	Signature	
signaturePropFile	<キーストア情報設定ファイル名>	署名検証用の鍵が格納されているキーストアに関する情報を記述したファイル名を指定します。
passwordCallbackClass	<コールバックハンドラクラス名>	パスワードを取得するためのコールバックハンドラクラス名を指定します。

MEMO

signatureParts パラメータを省略した場合、署名対象は SOAP エンベロープの Body 要素になります。

暗号化・復号化関連のパラメータ設定

●暗号化方式 1 (鍵暗号あり(公開鍵暗号方式))を利用する

■送信側

パラメータ	値	説明
action	Encryption	
keyEncryption	true	鍵暗号化を行います。
encryptionKeyTransportAlgorithm (鍵暗号アルゴリズムを指定します。)	http://www.w3.org/2001/04/xmlenc#rsa-1_5	RSA Version1.5 Key Transport アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p	RSA-OAEP Key Transport アルゴリズムを使用します。
encryptionSymAlgorithm (データ暗号アルゴリズムを指定します。)	http://www.w3.org/2001/04/xmlenc#tripledes-cbc	TripleDES アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#aes128-cbc	AES128 アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#aes256-cbc	AES256 アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#aes192-cbc	AES192 アルゴリズムを使用します。
encryptionKeyIdentifier (暗号化に使用した鍵または証明書の参照方法を指定します。)	DirectReference	メッセージの内部あるいは外部にある証明書を URI で参照します。
	IssuerSerial	メッセージの外部にある証明書を、発行者とシリアル番号をキーにして参照します。
	SKIKeyIdentifier	メッセージの外部にある証明書を、所有者鍵識別子をキーにして参照します。
	EmbeddedReference	メッセージの内部にある証明書を参照します。
encryptionPropFile	<キーストア情報設定ファイル名>	暗号に使用する公開鍵証明書が格納されているキーストアに関する情報を記述したファイル名を指定します。
userInfoCallbackClass	<コールバックハンドラクラス名>	暗号化に使用する証明書のエイリアスを取得するためのコールバックハンドラクラス名を指定します。
encryptionParts (暗号化対象を指定します。)	{Content}{XPath}{}	XPath には暗号化対象を表す XPath 式を指定します。 この場合、Content 暗号化(暗号化対象要素の中身を暗号化)を行います。
	{Element}{XPath}{}	XPath には署名対象を表す XPath 式を指定します。 この場合、Element 暗号化(暗号化対象要素全体を暗号化)を行います。

MEMO

encryptionParts パラメータを省略した場合、暗号化対象は SOAP エンベロープの Body 要素になり、Content 暗号化を行います。

■受信側

パラメータ	値	説明
action	Encryption	
decryptionPropFile	<キーストア情報設定ファイル名>	復号用の鍵が格納されているキーストアに関する情報を記述したファイル名を指定します。
passwordCallbackClass	<コールバックハンドラクラス名>	復号に使用する鍵のパスワードを取得するためのコールバックハンドラクラス名

●暗号化方式 2 (鍵暗号あり(共通鍵暗号方式))を利用する

■送信側

パラメータ	値	説明
action	Encryption	
keyEncryption	true	暗号化に使用する鍵を暗号化するかどうかを指定します。鍵暗号化は行います。
encryptionKeyWrapAlgorithm（鍵暗号アルゴリズムを指定します。）	http://www.w3.org/2001/04/xmlenc#kw-tripledes	TripleDES Key Wrap アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#kw-aes128	AES128 Key Wrap アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#kw-aes256	AES256 Key Wrap アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#kw-aes192	AES192 Key Wrap アルゴリズムを使用します。
encryptionSymAlgorithm（データ暗号アルゴリズムを指定します。）	http://www.w3.org/2001/04/xmlenc#tripledes-cbc	TripleDES アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#aes128-cbc	AES128 アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#aes256-cbc	AES256 アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#aes192-cbc	AES192 アルゴリズムを使用します。
encryptionKeyIdentifier	EmbeddedKeyName	暗号化に使用した鍵の参照方法を指定します。鍵を、エイリアスをキーにして参照します。
encryptionPropFile	<キーストア情報設定ファイル名>	暗号化に使用するの鍵が格納されているキーストアに関する情報を記述したファイル名を指定します。
userInfoCallbackClass	<コールバックハンドラクラス名>	暗号化に使用する鍵のエイリアスとパスワードを取得するためのコールバックハンドラクラス名を指定します。
encryptionParts（暗号化対象を指定します。）	{Content}{XPath}{}	XPath には暗号化対象を表す XPath 式を指定します。 この場合、Content 暗号化(暗号化対象要素の中身を暗号化)を行います。
	{Element}{XPath}{}	XPath には署名対象を表す XPath 式を指定します。 この場合、Element 暗号化(暗号化対象要素全体を暗号化)を行います。

MEMO

encryptionParts パラメータを省略した場合、暗号化対象は SOAP エンベロープの Body 要素になり、Content 暗号化を行います。

■受信側

暗号化方式 1 と同じ

●暗号化方式 3 (鍵暗号なし)を利用する

■送信側

パラメータ	値	説明
action	Encryption	
keyEncryption	false	暗号化に使用する鍵を暗号化するかどうかを指定します。鍵暗号化は行いません。
encryptionSymAlgorithm（データ暗号アルゴリズムを指定します。）	http://www.w3.org/2001/04/xmlenc#tripledes-cbc	TripleDES アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#aes128-cbc	AES128 アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#aes256-cbc	AES256 アルゴリズムを使用します。
	http://www.w3.org/2001/04/xmlenc#aes192-cbc	AES192 アルゴリズムを使用します。

encryptionKeyIdentifier	EmbeddedKeyName	暗号化に使用した鍵の参照方法を指定します。鍵を、エイリアスをキーにして参照します。
encryptionPropFile	<キーストア情報設定ファイルのパス>	暗号化に使用する鍵が格納されているキーストアに関する情報を記述したファイル名を指定します。
userInfoCallbackClass	<コールバックハンドラクラスのパス>	暗号化に使用する鍵のエイリアスとパスワードを取得するためのコールバックハンドラクラス名を指定します。
encryptionParts (暗号化対象を指定します。)	{Content}{XPath}{}	XPath には暗号化対象を表す XPath 式を指定します。 Content 暗号化(暗号化対象要素の中身を暗号化)を行います。
	{Element}{XPath}{}	XPath には署名対象を表す XPath 式を指定します。 Element 暗号化(暗号化対象要素全体を暗号化)を行います。

■受信側

暗号化方式 1 と同じ

MEMO

encryptionParts パラメータを省略した場合、暗号化対象は SOAP エンベロープの Body 要素になり、Content 暗号化を行います。

認証関連のパラメータ設定

●ID/Password 認証を利用する

■送信側

パラメータ	値	説明
action	UsernameToken	
authType	IDPassword	ID/Password 認証を使用します。
passwordType（パスワードタイプを指定します）	PasswordText	パスワードをプレーンテキストとして UsernameToken に格納します。
	PasswordDigest	パスワードのダイジェスト値を UsernameToken に格納します。
userInfoCallbackClass	<コールバックハンドラクラス名>	ユーザ名とパスワードを取得するためのコールバックハンドラクラス名を指定します。

■受信側

パラメータ	値	説明
action	UsernameToken	
authType	IDPassword	ID/Password 認証を使用します。
passwordCallbackClass	<コールバックハンドラクラス名>	パスワードを取得するためのコールバックハンドラクラス名を指定します。

●WebOTX 認証を利用する

WebOTX 認証とは、WebOTX が管理しているユーザ情報を利用して認証を行うことです。

■送信側

パラメータ	値	説明
action	UsernameToken	
authType	WebOTX	WebOTX 認証を行います。
passwordType	PasswordText	パスワードをプレーンテキストとして UsernameToken に格納します。
userInfoCallbackClass	<コールバックハンドラクラス名>	ユーザ名とパスワードを取得するためのコールバックハンドラクラス名を指定します。

■受信側

パラメータ	値	説明
action	UsernameToken	
authType	WebOTX	WebOTX 認証を指定します。

●Signature 認証を利用する

■送信側

パラメータ	値	説明
action	Signature	
authType	Signature	Signature 認証を指定します。
signatureAlgorithm（署名アルゴリズムを指定します。）	http://www.w3.org/2000/09/xmldsig#dsa-sha1	DSAwithSHA1 アルゴリズムを使用します。
	http://www.w3.org/2000/09/xmldsig#rsa-sha1	RSAwithSHA1 アルゴリズムを使用します。

signaturePropFile	<キーストア情報設定ファイル名>	署名用の鍵が格納されているキーストアに関する情報を記述したファイル名を指定します。
signatureKeyIdentifier (署名検証に使用する公開鍵証明書の参照方法を指定します。)	DirectReference	メッセージの内部あるいは外部にある証明書を URI で参照します。
	IssuerSerial	メッセージの外部にある証明書を、発行者とシリアル番号をキーにして参照します。
	SKIKeyIdentifier	メッセージの外部にある証明書を、所有者鍵識別子をキーにして参照します。
	EmbeddedReference	メッセージの内部にある証明書を参照します。
useSingleCert (署名に使用した秘密鍵とペアになる公開鍵証明書の BinarySecurityToken への格納形態を指定します。)	true	BinarySecurityToken に EE 証明書のみを格納します。
	false	BinarySecurityToken に証明書チェーン(EE 証明書 + CA 証明書)を格納します。
userInfoCallbackClass	<コールバックハンドラクラス名>	署名に使用する鍵のエイリアスとパスワードを取得するためのコールバックハンドラクラス名を指定します。
signatureParts (署名対象を指定します。)	{}{XPath}{}	XPath には署名対象を表す XPath 式を指定します。
	{}{XPath}{};STRTransform	XPath には署名対象を表す XPath 式を指定します。また、署名に関連付けられるセキュリティトークンを wsse:SecurityTokenReference 要素で参照し、STR Dereference Transform を使用して署名します。
	{}{XPath}{};Token	XPath には署名対象を表す XPath 式を指定します。また、署名に関連付けられるセキュリティトークンを ds:Reference 要素で参照し、署名します。

MEMO

useSingleCert パラメータは、公開鍵証明書を BinarySecurityToken に格納してメッセージに添付する場合にのみ使用できます(署名アルゴリズムが RSAwithSHA1 または DSAwithSHA1 かつ証明書の参照方法が DirectReference または EmbeddedReference の場合にのみ使用できます)。その他の場合は、指定しても無視されます。
本パラメータを省略した場合、BinarySecurityToken には EE 証明書のみが格納されます。

MEMO

signatureParts パラメータを省略した場合、署名対象は SOAP エンベロープの Body 要素になります。

■受信側

パラメータ	値	説明
action	Signature	
authType	Signature	Signature 認証を使用する場合は、本パラメータに Signature を指定します。
signaturePropFile	<キーストア情報設定ファイル名>	署名検証用の公開鍵証明書が格納されているキーストアに関する情報を記述したファイル名を指定します。
certDir	<証明書格納ディレクトリ名>	公開鍵証明書検証で必要となる証明書(EE 証明書、中間 CA、トラストアンカ)が格納されるディレクトリ名を指定します。ここを「cert」と指定すると、WebOTX 既定の格納場所<DomainDir>/config/cert を指定したことになります。
trustAnchor	<トラストアンカ名 1>;<トラストアンカ名 2>;…	公開鍵証明書検証で信頼済みとして扱われるトラストアンカ(ルート CA)を指定します。トラストアンカ証明書のファイル名を指定します。セミコロン区切りで複数指定できます。
crlValidation (証明書検証で CRL を参照するかどうかを指定します。)	true	CRL を参照します。
	false	CRL を参照しません。
crlDir	<CRL 格納ディレクトリ名>	CRL が格納されるディレクトリ名を指定し

		ます。crlValidation が true の場合に有効となります。このディレクトリにある*.crl ファイルが CRL として扱われます。ここを「crl」と指定すると、WebOTX 既定の格納場所<DomainDir>/config/crl を指定したことになります。

タイムスタンプ関連のパラメータ設定

■送信側

パラメータ	値	説明
action	Timestamp	
timeToLive	<有効期間>	メッセージの有効期間を秒単位で指定します。本パラメータを省略した場合、有効期間は 300 秒になります。

■受信側

共通の設定以外のパラメータはありません。

SAML関連のパラメータ設定

■Sender-Vouches モデルの Assertion(署名を付与しない)を利用する

●送信側

パラメータ	値	説明
action	SAMLTokenUnsigned	
assertionId	<SAML Assertion の識別情報>	コールバックハンドラで取得する SAML Assertion を特定するための情報を指定します。コールバックハンドラで、識別情報をキーにして SAML Assertion を取得する仕組みを提供します。識別情報は、アプリケーションで独自に定義します。本パラメータは省略可能です。
samlTokenCallbackClass	<コールバックハンドラクラス名>	SAML Assertion を取得するためのコールバックハンドラクラス名を指定します。外部から SAML Assertion を受け取る仕組みを提供します。

●受信側

共通の設定以外のパラメータはありません。action パラメータは、「SAMLTokenUnsigned」です。

■Sender-Vouches モデルの Assertion (署名を付与する)を利用する

●送信側

パラメータ	値	説明
action	SAMLTokenSigned	
assertionId	<SAML Assertion の識別情報>	コールバックハンドラで取得する SAML Assertion を特定するための情報を指定します。コールバックハンドラで、識別情報をキーにして SAML Assertion を取得する仕組みを提供します。識別情報は、アプリケーションで独自に定義します。本パラメータは省略可能です。
samlTokenCallbackClass	<コールバックハンドラクラス名>	SAML Assertion を取得するためのコールバックハンドラクラス名を指定します。外部から SAML Assertion を受け取る仕組みを提供します。
signatureAlgorithm (署名アルゴリズムを指定します。)	http://www.w3.org/2000/09/xmldsig#dsa-sha1	DSAwithSHA1 アルゴリズムを指定します。
	http://www.w3.org/2000/09/xmldsig#rsa-sha1	RSAwithSHA1 アルゴリズムを指定します。
signaturePropFile	<キーストア情報設定ファイル名>	署名に使用する鍵が格納されているキーストアに関する情報を記述したファイル名を指定します。
signatureKeyIdentifier (署名検証に使用する公開鍵証明書の参照方法を指定します。)	DirectReference	メッセージの内部あるいは外部にある証明書を URI で参照します。
	IssuerSerial	メッセージの外部にある証明書を、発行者とシリアル番号をキーにして参照します。
	SKIKeyIdentifier	メッセージの外部にある証明書を、所有者鍵識別子をキーにして参照します。
	EmbeddedReference	メッセージの内部にある証明書を参照します。
useSingleCert	true	署名に使用した秘密鍵とペアになる公開鍵証明書の BinarySecurityToken への格納形態を指定します。BinarySecurityToken に EE 証明書のみ

		を格納します。
userInfoCallbackClass	<コールバックハンドラクラス名>	署名に使用する鍵のエイリアスとパスワードを取得するためのコールバックハンドラクラス名を指定します。
signatureParts	{}{XPath}{}	署名対象を指定します。XPath には署名対象を表す XPath 式を指定します。本パラメータを省略した場合、署名対象は SOAP エンベロープの Body 要素になります。

●受信側

パラメータ	値	説明
action	Signature SAMLTokenUnsigned	スペース区切りで記述します。
signaturePropFile	<キーストア情報設定ファイル名>	署名検証用の公開鍵証明書が格納されているキーストアに関する情報を記述したファイル名を指定します。
certDir	<証明書格納ディレクトリ名>	公開鍵証明書検証で必要となる証明書(EE 証明書、中間 CA、トラストアンカ)が格納されるディレクトリ名を指定します。ここを「cert」と指定すると、WebOTX 既定の格納場所<DomainDir>/config/cert を指定したことになります。
trustAnchor	<トラストアンカ名 1>;<トラストアンカ名 2>;…	公開鍵証明書検証で信頼済みとして扱われるトラストアンカ(ルート CA)を指定します。トラストアンカ証明書のファイル名を指定します。セミコロン区切りで複数指定できます。
crlValidation (証明書検証で CRL を参照するかどうかを指定します。)	true	CRL を参照します。
	false	CRL を参照しません。
crlDir	<CRL 格納ディレクトリ名>	CRL が格納されるディレクトリ名を指定します。crlValidation が true の場合に有効となります。このディレクトリにある*.crl ファイルが CRL として扱われます。ここを「crl」と指定すると、WebOTX 既定の格納場所<DomainDir>/config/crl を指定したことになります。

■Holder-of-key モデルの Assertion(署名を付与しない)を利用する

●送信側

「Sender-Vouches モデルの Assertion(署名を付与しない)」と同じです。action パラメータは「SAMLTokenUnsigned」です。

●受信側

パラメータ	値	説明
action	SAMLTokenUnsigned	
certDir	<証明書格納ディレクトリ名>	公開鍵証明書検証で必要となる証明書(EE 証明書、中間 CA、トラストアンカ)が格納されるディレクトリ名を指定します。ここを「cert」と指定すると、WebOTX 既定の格納場所<DomainDir>/config/cert を指定したことになります。
trustAnchor	<トラストアンカ名 1>;<トラストアンカ名 2>;…	公開鍵証明書検証で信頼済みとして扱われるトラストアンカ(ルート CA)を指定します。トラストアンカ証明書のファイル名を指定します。セミコロン区切りで複数指

		定できます。
crlValidation （証明書検証で CRL を参照するかどうかを指定します。）	true	CRL を参照します。
	false	CRL を参照しません。
crlDir	<CRL 格納ディレクトリ名>	CRL が格納されるディレクトリ名を指定します。crlValidation が true の場合に有効となります。このディレクトリにある*.crl ファイルが CRL として扱われます。ここを「crl」と指定すると、WebOTX 既定の格納場所<DomainDir>/config/crl を指定したことになります。

■Holder-of-key の Assertion（署名を付与する）を利用する

●送信側

パラメータ	値	説明
action	SAMLTokenSigned	
assertionId	<SAML Assertion の識別情報>	コールバックハンドラで取得する SAML Assertion を特定するための情報を指定します。コールバックハンドラで、識別情報をキーにして SAML Assertion を取得する仕組みを提供します。識別情報は、アプリケーションで独自に定義します。本パラメータは省略可能です。
samlTokenCallbackClass	<コールバックハンドラクラス名>	SAML Assertion を取得するためのコールバックハンドラクラス名を指定します。外部から SAML Assertion を受け取る仕組みを提供します。
signatureAlgorithm （署名アルゴリズムを指定します。）	http://www.w3.org/2000/09/xmldsig#dsa-sha1	DSAwithSHA1 アルゴリズムを指定します。
	http://www.w3.org/2000/09/xmldsig#rsa-sha1	RSAwithSHA1 アルゴリズムを指定します。
signaturePropFile	<キーストア情報設定ファイル名>	署名で使用する鍵が格納されているキーストアに関する情報を記述したファイル名を指定します。
signatureKeyIdentifier	SKIKeyIdentifier	署名検証に使用する公開鍵証明書の参照方法を指定します。
useSingleCert	true	署名に使用した秘密鍵とペアになる公開鍵証明書の BinarySecurityToken への格納形態を指定します。BinarySecurityToken に EE 証明書のみを格納します。
userInfoCallbackClass	<コールバックハンドラクラス名>	署名に使用する鍵のエイリアスとパスワードを取得するためのコールバックハンドラクラス名を指定します。
signatureParts	{}{XPath}{}	署名対象を指定します。XPath には署名対象を表す XPath 式を指定します。本パラメータを省略した場合、署名対象は SOAP エンベロープの Body 要素になります。

●受信側

パラメータ	値	説明
action	Signature SAMLTokenUnsigned	スペース区切りで記述します。
signaturePropFile	<キーストア情報設定ファイル名>	署名検証用の公開鍵証明書が格納されているキーストアに関する情報を記述したファイル名を指定します。

certDir	<証明書格納ディレクトリ名>	公開鍵証明書検証で必要となる証明書(EE 証明書、中間 CA、トラストアンカ)が格納されるディレクトリ名を指定します。ここを「cert」と指定すると、WebOTX 既定の格納場所<DomainDir>/config/cert を指定したことになります。
trustAnchor	<トラストアンカ名 1>;<トラストアンカ名 2>;…	公開鍵証明書検証で信頼済みとして扱われるトラストアンカ(ルート CA)を指定します。トラストアンカ証明書のファイル名を指定します。セミコロン区切りで複数指定できます。
crlValidation (証明書検証で CRL を参照するかどうかを指定します。)	true	CRL を参照します。
	false	CRL を参照しません。
crlDir	<CRL 格納ディレクトリ名>	CRL が格納されるディレクトリ名を指定します。crlValidation が true の場合に有効となります。このディレクトリにある*.crl ファイルが CRL として扱われます。ここを「crl」と指定すると、WebOTX 既定の格納場所<DomainDir>/config/crl を指定したことになります。

2.1.12.JAX-WSアプリケーション開発

ここでは、JAX-WS アプリケーションを開発するための手順について説明します。

2.1.12.1. 環境設定

まず、JAX-WS アプリケーションの開発に必要な設定について説明します。

JDK 5.0 またはJDK 6.0 のインストール

JAX-WS アプリケーションが動作するためには、JDK 5.0 または JDK 6.0 が必須となります。WebOTX インストール時、使用する JDK に JDK 5.0 または JDK 6.0 を指定してください。

環境変数の設定

環境変数 JAVA_HOME に、JDK インストールディレクトリを指定してください。また、javac コマンドや apt コマンドがコマンド検索パスに加わるよう、環境変数 PATH に<JAVA_HOME>/bin を追加してください。

MEMO

<JAVA_HOME>は JDK のインストールディレクトリに読み替えてください。

JAX-WSの有効化

WebOTX インストール直後は、JAX-WS ライブラリが WebOTX システム上で有効になっていません。WebOTX マニュアルのセットアップガイドの「インストール後の作業」に従って、あらかじめ、JAX-WS ライブラリを有効化しておく必要があります。

クラスパスの設定

JAX-WS アプリケーションを作成したり、JAX-WS クライアントアプリケーションを実行する際は、次のライブラリをクラスパスに含める必要があります。

```
<WebOTX_DIR>/lib/endorsed/saaj-api.jar
<WebOTX_DIR>/lib/wosv-ws.jar
<WebOTX_DIR>/lib/wojaxb.jar
<WebOTX_DIR>/lib/saaj-impl.jar
<WebOTX_DIR>/lib/activation.jar
<WebOTX_DIR>/lib/jsr173api.jar
<WebOTX_DIR>/lib/ant/ant.jar
<JAVA_HOME>/lib/tools.jar
```

MEMO

<WebOTX_DIR>は WebOTX のインストールディレクトリに読み替えてください。

2.1.12.2. サーバアプリケーション開発

ここでは、JAX-WS のサーバ側アプリケーションを開発するための手順について説明します。

サーバアプリケーションの作成

サーバアプリケーションを作成するには次の 2 通りの方法があります。l

- Web サービスエンドポイント実装クラスから開発する方法
- WSDL から開発する方法

前者は Web サービスエンドポイント実装クラスを作成し、その実装クラスから、Request/Response Bean などの配備に必要なクラスを自動生成する方法です。WSDL を自動生成することから、Web サービス固有の WSDL を記述する手間を省くことができます。

それに対して、後者は WSDL を作成し、その WSDL からサービスエンドポイントインタフェースを自動生成し、そのサービスエンドポイントインタフェースを実装する方法です。既存の WSDL が存在する場合や WSDL を直接記述できる場合に使用できます。

ここでは、二つの数字を受け取り、その和を計算して返す Web サービスのサンプルプログラムの作成方法について説明します。ここでは、サンプルプログラムの名前を AddNumbers とします。

Webサービスエンドポイント実装クラスから開発する方法

この方法でサンプルプログラムを作成するためのファイル構成は、以下の通りです。任意のディレクトリに以下のファイルを作成してください。それぞれのファイルの内容については、以下で説明します。

ディレクトリ	ファイル
src/sample/server/	AddNumbersImpl.java AddNumbersException.java
WEB-INF/	nec-jaxws.xml nec-web.xml web.xml

1. Web サービスエンドポイント実装クラスの作成

   Web サービスエンドポイント実装クラスを作成します。このとき、Web サービスとして公開するクラスには、javax.jws.WebService アノテーションを付与します。なお、javax.jws.WebService をインポートしている場合、アノテーションでのパッケージ名の指定を省略することができます。

AddNumbersImpl.java

```
package sample.server;
import javax.jws.WebService;

@WebService
public class AddNumbersImpl {
    public int addNumbers(int number1, int number2) throws AddNumbersException {
        if (number1 < 0 || number2 < 0) {
            throw new AddNumbersException("Negative number cannot be added!",
                "Numbers: " + number1 + ", " + number2);
        }
        return number1 + number2;
    }
}
```

   このアプリケーションはマイナスの数字を引数として受け取ると、次の AddNumbersException 例外を返します。

AddNumbersException.java

```
package sample.server;

public class AddNumbersException extends Exception {
    String detail;
    public AddNumbersException (String message, String detail) {
        super (message);
        this.detail = detail;
    }
    public String getDetail () {
        return detail;
    }
}
```

2. 配備するクラスの作成

   apt コマンドを使用して、配備するクラスを生成します。src ディレクトリで実行した apt コマンドの例は次の通りです。なお、以下のコマンドを実行する際に、出力ディレクトリ WEB-INF/classes が自動で作成されます。

```
src> apt -d ../WEB-INF/classes sample/server/AddNumbersImpl.java
```

WSDLファイルを出力する場合は、wsgenコマンドを使用してください。なお、以下のコマンドを実行する前に、あらかじめ出力ディレクトリ WEB-INF/classes を作成しておいてください。

```
src> javac -d ../WEB-INF/classes sample/server/AddNumbersImpl.java
src> wsgen -d ../WEB-INF/classes -s . -wsdl -classpath ../WEB-INF/classes
       sample.server.AddNumbersImpl
```

aptコマンドやwsgenコマンドの詳細に関しては、2.1.12.6 コマンドインタフェースを参照してください。

これにより、次のファイルが生成されます

ディレクトリ	ファイル
src/sample/server/jaxws	AddNumbers.java AddNumbersResponse.java AddNumbersExceptionBean.java
WEB-INF/classes/	AddNumbersImplService.wsdl (wsgen 実行時のみ) AddNumbersImplService_schema1.xsd (wsgen 実行時のみ)
WEB-INF/classes/sample/server/	AddNumbersImpl.class AddNumbersException.class
WEB-INF/classes/sample/server/jaxws	AddNumbers.class AddNumbersResponse.class AddNumbersExceptionBean.class

WSDLから開発する方法

この方法で AddNumbers サンプルを作成するためのファイル構成は、以下の通りです。任意のディレクトリに以下のファイルを作成してください。それぞれのファイルの内容については、以下で説明します。

ディレクトリ	ファイル
src/sample/server/	AddNumbersImpl.java
WEB-INF/	nec-jaxws.xml nec-web.xml web.xml
WEB-INF/wsdl	AddNumbers.wsdl

1. エンドポイントの WSDL の作成

   Web サービスのエンドポイントインタフェースを記述した WSDL をテキストエディタ等で作成します。

AddNumbers.wsdl

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

<definitions name="AddNumbers"
      targetNamespace="http://server.sample/"
      xmlns:tns="http://server.sample/"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
      <types>
            <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
                  elementFormDefault="qualified"
                  targetNamespace="http://server.sample/">
                  <complexType name="addNumbersResponse">
                        <sequence><element name="return" type="xsd:int"/></sequence>
```

```
            </complexType>
            <element name="addNumbersResponse" type="tns:addNumbersResponse"/>
            <complexType name="addNumbers">
                <sequence>
                    <element name="arg0" type="xsd:int"/>
                    <element name="arg1" type="xsd:int"/>
                </sequence>
            </complexType>
            <element name="addNumbers" type="tns:addNumbers"/>
            <element name="AddNumbersFault" type="tns:AddNumbersFault"/>
            <complexType name="AddNumbersFault">
                <sequence>
                    <element name="faultInfo" type="xsd:string"/>
                    <element name="message" type="xsd:string"/>
                </sequence>
            </complexType>
        </xsd:schema>
    </types>
    <message name="addNumbers">
        <part name="parameters" element="tns:addNumbers"/>
    </message>
    <message name="addNumbersResponse">
        <part name="result" element="tns:addNumbersResponse"/>
    </message>
    <message name="addNumbersFault">
        <part name="AddNumbersFault" element="tns:AddNumbersFault"/>
    </message>
    <portType name="AddNumbersPortType">
        <operation name="addNumbers">
            <input message="tns:addNumbers" name="add"/>
            <output message="tns:addNumbersResponse" name="addResponse"/>
            <fault name="addNumbersFault" message="tns:addNumbersFault"/>
        </operation>
    </portType>
    <binding name="AddNumbersBinding" type="tns:AddNumbersPortType">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <operation name="addNumbers">
            <soap:operation soapAction=""/>
            <input><soap:body use="literal"/></input>
            <output><soap:body use="literal"/></output>
            <fault name="addNumbersFault">
                <soap:fault name="addNumbersFault" use="literal"/>
            </fault>
        </operation>
    </binding>
    <service name="AddNumbersService">
        <port name="AddNumbersPort" binding="tns:AddNumbersBinding">
            <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
        </port>
    </service>
</definitions>
```

2. サービスエンドポイントインタフェースの生成

wsimport コマンドを使用してサービスエンドポイントインタフェースや、インポートした XML スキーマからマッピングされるバリュークラスを生成します。なお、以下のコマンドを実行する前に、あらかじめ出力ディレクトリ WEB-INF/classes を作成しておいてください。

```
src> wsimport -d ../WEB-INF/classes -s . -keep ../WEB-INF/wsdl/AddNumbers.wsdl
```

wsimportコマンドの詳細に関しては、2.1.12.6 コマンドインタフェースを参照してください。

これにより、次のファイルが生成されます

ディレクトリ	ファイル
src/sample/server/	AddNumbers.java
	AddNumbersFault.java
	AddNumbersFault_Exception.java
	AddNumbersPortType.java
	AddNumbersResponse.java
	AddNumbersService.java
	ObjectFactory.java
	package-info.java
WEB-INF/classes/sample/server/	AddNumbers.class
	AddNumbersFault.class
	AddNumbersFault_Exception.class
	AddNumbersPortType.class
	AddNumbersResponse.class
	AddNumbersService.class
	ObjectFactory.class
	package-info.class

wsimport コマンドで自動生成されたサービスエンドポイントインタフェースは次のようになります。

AddNumbersPortType.java

```
package sample.server;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

@WebService(name = "AddNumbersPortType", targetNamespace = "http://server.sample/")
public interface AddNumbersPortType {

    @WebMethod
    @WebResult(targetNamespace = "http://server.sample/")
    @RequestWrapper(localName = "addNumbers",
                    targetNamespace = "http://server.sample/",
                    className = "sample.server.AddNumbers")
    @ResponseWrapper(localName = "addNumbersResponse",
                     targetNamespace = "http://server.sample/",
                     className = "sample.server.AddNumbersResponse")
    public int addNumbers(
        @WebParam(name = "arg0", targetNamespace = "http://server.sample/")
        int arg0,
        @WebParam(name = "arg1", targetNamespace = "http://server.sample/")
        int arg1)
        throws AddNumbersFault_Exception;
}
```

3. サービスエンドポイントインタフェースの実装

wsimport コマンドで生成された、サービスエンドポイントインタフェースを実装します。

AddNumbersImpl.java

```
package sample.server;

@javax.jws.WebService(endpointInterface="sample.server.AddNumbersPortType")
public class AddNumbersImpl {

    public int addNumbers(int number1, int number2)
    throws AddNumbersFault_Exception {
        if (number1 < 0 || number2 < 0) {
            String message = "Negative number cannot be added!";
            String detail = "Numbers: " + number1 + ", " + number2;
            AddNumbersFault fault = new AddNumbersFault();
            fault.setMessage(message);
            fault.setFaultInfo(detail);
            throw new AddNumbersFault_Exception(message, fault);
        }
        return number1 + number2;
    }
}
```

実装クラスでは WebService アノテーションを記述し、endpointInterface には wsimport コマンドで自動生成されたサービスエンドポイントインタフェースのクラス名をフルパッケージで指定してください。

最後に、実装クラスをコンパイルします。

```
src> javac -d ../WEB-INF/classes sample/server/AddNumbersImpl.java
```

WARの作成

Web サーバに配備するための war を作成します。

エンドポイントの定義ファイル nec-jaxws.xml は次のようになります。

Webサービスエンドポイント実装クラスから作成した場合

nec-jaxws.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://www.nec.com/WebOTX/xml/ns/jax-ws/ri/runtime" version="2.0">
    <endpoint
        name="sample"
        implementation="sample.server.AddNumbersImpl"
        url-pattern="/addnumbers"/>
</endpoints>
```

<endpoint>の属性	説明
name	Web サービスエンドポイントの識別子を指定します。
implementation	Web サービスエンドポイントの実装クラス名を指定します。
url-pattern	Web サービスエンドポイントへアクセスするための URL を指定します。

WSDLから作成した場合

nec-jaxws.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://www.nec.com/WebOTX/xml/ns/jax-ws/ri/runtime" version="2.0">
    <endpoint
```

```
        name="sample"
        interface="sample.server.AddNumbersPortType"
        implementation="sample.server.AddNumbersImpl"
        wsdl="WEB-INF/wsdl/AddNumbers.wsdl"
        service="{http://server.sample/}AddNumbersService"
        port="{http://server.sample/}AddNumbersPort"
        url-pattern="/addnumbers"/>
</endpoints>
```

<endpoint>の属性	説明
interface	自動生成されたサービスエンドポイントインタフェースを指定します。
wsdl	wsimport で指定した WSDL ファイルを指定します。
service	WSDL ファイルの service 要素の name 属性に記述した、エンドポイントのサービス名を指定します。
port	WSDL ファイルの port 要素の name 属性に記述した、エンドポイントのポート名を指定します。

配備記述子 web.xml は次のようになります。

web.xml

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">
  <display-name>sample</display-name>
  <listener>

<listener-class>com.nec.webotx.webservice.xml.ws.transport.http.servlet.WSServletContextListener
</listener-class>
  </listener>
  <servlet>
    <servlet-name>sample</servlet-name>

<servlet-class>com.nec.webotx.webservice.xml.ws.transport.http.servlet.WSServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>sample</servlet-name>
    <url-pattern>/addnumbers</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
</web-app>
```

高信頼性メッセージング機能を利用する場合、「2.1.13 高信頼メッセージング」-「WS-RM・WS-Rサーバの利用方法」の説明に従って、<listener-class>、<servlet-class>のクラスを設定してください。WebOTX JAX-WSを単独で利用する場合は、<listener-class>、<servlet-class>にそれぞれ以下のクラスを指定してください。

タグ	指定するクラス
<listener-class>	com.nec.webotx.webservice.xml.ws.transport.http.servlet.WSServletContextListener
<servlet-class>	com.nec.webotx.webservice.xml.ws.transport.http.servlet.WSServlet

nec-web.xml は次のようになります。

nec-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<nec-web-app>
  <context-root>/jaxws-sample</context-root>
</nec-web-app>
```

最後に、サーバアプリケーションと各種設定ファイルを WAR ファイルにアーカイブします。WAR ファイルに含めるファイルは以下のように配置します。

ディレクトリ	ファイル
WEB-INF/	nec-jaxws.xml, nec-web.xml, web.xml
WEB-INF/wsdl/	AddNumbers.wsdl (WSDL からアプリケーションを作成した場合)
WEB-INF/classes/	サーバアプリケーションのクラスファイル

上記構成の WEB-INF ディレクトリが用意できたら、以下のコマンドを実行して WAR ファイルを作成します。

```
> jar cvf jaxws-sample.war WEB-INF
```

アプリケーションの配備

作成したWARファイルを、運用管理コマンドや運用管理コンソール、統合運用管理ツールなどで配備します。ここでは運用管理コマンドでの例を紹介します。その他のツールについては、運用編の「運用管理ツールとコマンド操作ガイド」を参照してください。

まず、otxadmin コマンドを実行後、ドメインにログインします。ここでは、あらかじめ登録されているドメイン管理ユーザでログインする例を示します。

```
otxadmin> login --user admin --password adminadmin --port 6212
```

次に、deployコマンドでWebアプリケーションを配備します。例として、jaxws-sample.warを配備するコマンドを以下に示します。

```
otxadmin> deploy jaxws-sample.war
```

なお、WebOTX Standard Edition や Enterprise Edition で、Web コンテナをマルチプロセスモードでインストールした場合は、次のようにアプリケーショングループとプロセスグループを指定してください。配備対象のアプリケーショングループを apg1, プロセスグループを pg1 とした場合の例を示します。

```
otxadmin> deploy --apgroup apg1 --pgroup pg1 jaxws-sample.war
```

再配備を行う場合は--force オプションを付加してください。deploy コマンドオプションの詳細については、運用管理コマンドリファレンスマニュアルの「運用管理エージェント運用管理コマンド(otxadmin)」ー「deploy」を参照してください。

2.1.12.3. クライアントアプリケーション開発

ここでは、JAX-WS のクライアント側アプリケーションを開発するための手順について説明します。

クライアントアプリケーションの作成

クライアントアプリケーションは、プロキシ(サービスエンドポイントインタフェースを実装したスタブ)、もしくはDispatchを利用してリモートのWebサービスエンドポイントにアクセスすることができます。ここでは、プロキシを使用したアプリケーションを説明します。Dispatchを使用したアプリケーションについては、2.1.12.4 Dispatchの使用をご覧ください。

2.1.12.2 サーバアプリケーションの開発で作成したAddNumbersサンプルのクライアントアプリケーション

を作成するためのファイル構成は、以下の通りです。サーバアプリケーションを開発したディレクトリ以外の場所に、次のファイルを作成してください。ファイルの内容については、以下で説明します。

ディレクトリ	ファイル
src/sample/client/	AddNumbersClient.java

クライアントアプリケーションを開発する手順は次のようになります。

1. プロキシの生成

   wsimport コマンドを使用してサービスエンドポイントインタフェースのプロキシやサービスエンドポイントインタフェースのリファレンスを取得するためのサービスクラス、Request/Response Bean などを生成します。

   あらかじめ、AddNumbers サンプルを WebOTX に配備したのち、src ディレクトリにて以下のコマンドを実行してください。なお、以下のコマンドを実行する前に、あらかじめ出力ディレクトリ classes を作成しておいてください。

```
src> wsimport -d ../classes -s . -keep http://localhost/jaxws-sample/addnumbers?wsdl
```

   wsimport コマンドの詳細に関しては、2.1.11.6 コマンドインタフェースを参照してください。

   これにより、次のファイルが生成されます。(Web サービスエンドポイント実装クラスからサーバアプリケーションを作成した場合)

ディレクトリ	ファイル
src/sample/server/	AddNumbers.java AddNumbersException.java AddNumbersException_Exception.java AddNumbersImpl.java AddNumbersImplService.java AddNumbersResponse.java ObjectFactory.java package-info.java
classes/sample/server/	AddNumbers.class AddNumbersException.class AddNumbersException_Exception.class AddNumbersImpl.class AddNumbersImplService.class AddNumbersResponse.class ObjectFactory.class package-info.class

2. クライアントアプリケーションの作成

   プロキシを使用したクライアントアプリケーションは次のようになります。

AddNumbersClient.java

```
package sample.client;
import sample.server.AddNumbersImpl;
import sample.server.AddNumbersImplService;
import sample.server.AddNumbersException_Exception;

public class AddNumbersClient {
```

```
    public static void main (String[] args) {
        try {
            AddNumbersImpl port = new AddNumbersImplService().getAddNumbersImplPort();
            int num1 = 10;
            int num2 = 20;

            int result = port.addNumbers(num1, num2);
            System.out.printf("addNumbers(%d, %d) = %d¥n", num1, num2, result);

            int result2 = port.addNumbers(num1, -num2); // exception
            System.out.printf("addNumbers(%d, %d) = %d¥n", num1, -num2, result2);

        } catch (AddNumbersException_Exception e) {
            System.out.println("Catch AddNumbersException_Exception: " +
                                e.getFaultInfo().getDetail());
        }
    }
}
```

上記コードのように、サービスクラスの getAddNumbersImplPort()メソッドで、プロキシ(スタブ)を取得することができます。

AddNumbers サンプルの開発を WSDL から作成する方法で行った場合、上記の AddNumbersImpl を AddNumbers、AddNumbersImplService()を AddNumbersService()、getAddNumbersImplPort() を getAddNumbersPort()、AddNumbersException_Exception を AddNumbersFault_Exception と読み替えてください。

以上の作業が終了したら、クライアントクラスをコンパイルします。

```
src> javac -d ../classes sample/client/AddNumbersClient.java
```

最後に、クライアントクラスを実行します。classes ディレクトリで実行してください。JDK 6.0 を使用する場合は、次の Java のシステムプロパティを追加して実行してください。

```
-Djavax.xml.ws.spi.Provider=com.nec.webotx.webservice.xml.ws.spi.ProviderImpl
```

```
classes> java sample.client.AddNumbersClient
```

正常に実行すれば、次のような結果が出力されます。

```
addNumbers(10, 20) = 30
Catch AddNumbersException_Exception: Numbers: 10, -20
```

2.1.12.4. Dispatchの使用

ここでは、Dispatch を使用した場合の JAX-WS のクライアント側アプリケーションについて説明します。

Dispatchを使用したクライアントアプリケーション

Dsipatch は動的な Web サービス呼び出しのインタフェースとして利用できます。Dispatch を使用したクライアントアプリケーションを、以下の 3 つの方法に分けて説明します。

- javax.xml.transform.Source を使用する方法
- javax.xml.bind.JAXBContext を使用する方法
- javax.xml.soap.SOAPMessage を使用する方法

以下の例で Dispatch オブジェクトを生成している、createDispatch メソッドの第 3 パラメタで指定する Service.Mode には以下を指定します。

Service.Mode の値	説明
MESSAGE	SOAP メッセージ全体を扱います。Source および SOAPMessage を利用する場合に指定できます。

PAYLOAD	SOAP メッセージの本体のみを扱います。Source および JAXB を利用する場合に指定できます。

javax.xml.transform.Sourceを使用する方法

メッセージを Stream や DOM/SAX で扱う方法です。

AddNumbersClient.java

```
package sample.client;

import sample.server.*;
import java.io.StringReader;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.namespace.QName;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.Source;

public class AddNumbersClient {
	public static void main(String[] args) {

		Service service = new AddNumbersImplService();
		QName portQName = new QName("http://server.sample/", "AddNumbersImplPort");
		String request = "<ns1:addNumbers xmlns:ns1=¥"http://server.sample/¥">" +
				"<arg0>10</arg0><arg1>20</arg1></ns1:addNumbers>";
		try {
			Dispatch<Source> sourceDispatch = service.createDispatch(
					portQName, Source.class, Service.Mode.PAYLOAD);
			Source result = sourceDispatch.invoke(
					new StreamSource(new StringReader(request)));
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

javax.xml.bind.JAXBContextを使用する方法

JAXB を利用して、メッセージをシリアライズ/デシリアライズする方法です。

AddNumbersClient.java

```
package sample.client;

import sample.server.*;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.namespace.QName;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBElement;

public class AddNumbersClient {
	public static void main(String[] args) {
		Service service = new AddNumbersImplService();
		QName portQName = new QName("http://server.sample/", "AddNumbersImplPort");
		try {
			JAXBContext jaxbContext = JAXBContext.newInstance(ObjectFactory.class);
			Dispatch jaxbDispatch =
					service.createDispatch(portQName, jaxbContext, Service.Mode.PAYLOAD);
			ObjectFactory factory = new ObjectFactory();
```

```
                AddNumbers numbers = new AddNumbers();
                numbers.setArg0(10);
                numbers.setArg1(20);
                JAXBElement<AddNumbers> addNumbers = factory.createAddNumbers(numbers);
                JAXBElement<AddNumbersResponse> response =
                        (JAXBElement<AddNumbersResponse>) jaxbDispatch.invoke(addNumbers);
                AddNumbersResponse result = response.getValue();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

javax.xml.soap.SOAPMessageを使用する方法

メッセージを SAAJ の SOAPMessage で扱う方法です。

AddNumbersClient.java

```
package sample.client;

import sample.server.*;
import javax.xml.soap.*;
import java.io.StringReader;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.namespace.QName;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.Source;

public class AddNumbersClient {
    public static void main(String[] args) {

        Service service = new AddNumbersImplService();
        QName portQName = new QName("http://server.sample/", "AddNumbersImplPort");
        String request = "<soapenv:Envelope" +
                " xmlns:soapenv=¥"http://schemas.xmlsoap.org/soap/envelope/¥"" +
                " xmlns:xsd=¥"http://www.w3.org/2001/XMLSchema¥" " +
                " xmlns:ns1=¥"http://server.sample/¥">" +
                "<soapenv:Body><ns1:addNumbers><arg0>10</arg0><arg1>20</arg1>" +
                "</ns1:addNumbers></soapenv:Body></soapenv:Envelope>";
        try {
            SOAPMessage message = null;
            MessageFactory factory = MessageFactory.newInstance();
            message = factory.createMessage();
            message.getSOAPPart().setContent(
                    (Source) new StreamSource(new StringReader(request)));
            message.saveChanges();
            Dispatch<SOAPMessage> dispatch = service.createDispatch(
                    portQName, SOAPMessage.class, Service.Mode.MESSAGE);
            SOAPMessage response = dispatch.invoke(message);
            Source result = response.getSOAPPart().getContent();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2.1.12.5. ハンドラの利用

JAX-WS では、Web サービスのメッセージを処理するためのプラグインフレームワークとしてハンドラを利

用することができます。ハンドラはクライアントおよびサーバで登録でき、登録したハンドラはメッセージの送受信ごとに呼び出されます。

ハンドラには Logical Handler と Protocol Handler の 2 種類があります。

- Logical Handler

  Logical Handler はメッセージコンテキストおよびメッセージの本体を扱うことができます。メッセージのプロトコル固有部分は扱うことができません。

- Protocol Handler

  Protocol Handler はメッセージコンテキストおよびプロトコル固有メッセージを扱うことができます。

LogicalHandler

Logical Handler は javax.xml.ws.handler.LogicalHandler を継承して作成します。

以下のメソッドを実装する必要があります。

```
public boolean handleMessage(LogicalMessageContext context);
public boolean handleFault(LogicalMessageContext context);
public void close(MessageContext ctx);
```

なお、非同期通信でハンドラが呼び出される場合には handleMessage および handleFault の戻り値として true を必ず返却するように実装してください。

Logical Handler のサンプルは次のようになります。

SampleLogicalHandler.java

```
package sample.server;
import javax.xml.ws.handler.*

public class SampleLogicalHandler implements LogicalHandler<LogicalMessageContext> {
	public boolean handleMessage(LogicalMessageContext context) {
		// 通常のメッセージの送受信ごとに呼び出されます。
		return true;
	}
	public boolean handleFault(LogicalMessageContext context) {
		// Falut メッセージの送受信ごとに呼び出されます。
		return true;
	}
	public void close(MessageContext ctx) {
		// メッセージ、Fault メッセージ、例外をディスパッチする直前に呼び出されます。
	}
}
```

ProtocolHandler

Protocol Handler は javax.xml.ws.handler.LogicalHandler 以外の、javax.xml.ws.handler.Handler インタフェースを継承したものになります。JAX-WS では SOAP プロトコルを扱うための javax.xml.ws.handler.soap.SOAPHandler インタフェースを提供しています。

SOAPHandler を継承した Protocol Handler は、以下のメソッドを実装する必要があります。

```
public Set<QName> getHeaders();
public boolean handleMessage(SOAPMessageContext context);
public boolean handleFault(SOAPMessageContext context);
public void close(MessageContext ctx);
```

Protocol Handler のサンプルは次のようになります。

SampleSOAPHandler.java

```
package sample.server;
```

```
import java.util.Set;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;

public class SampleSOAPHandler implements SOAPHandler<SOAPMessageContext> {
    public Set<QName> getHeaders() {
        return null;
    }
    public boolean handleMessage(SOAPMessageContext context) {
        // 通常のメッセージの送受信ごとに呼び出されます。
        return true;
    }

    public boolean handleFault(SOAPMessageContext context) {
        // Falut メッセージの送受信ごとに呼び出されます。
        return true;
    }

    public void close(MessageContext messageContext) {
        // メッセージ、Fault メッセージ、例外をディスパッチする直前に呼び出されます。
    }
}
```

ハンドラの登録

ハンドラを登録する方法は、アプリケーションで Binding を取得して、Binding.setHandlerChain()で登録する方法と、バインディングファイルを作成して、そのバインディングファイルに登録するハンドラを記述しておく方法の 2 通りの方法があります。

アプリケーションで登録する場合のサンプルは次のようになります。

```
SampleSOAPHandler handler = new SampleSOAPHandler();
List<Handler> handlerList = new ArrayList<Handler>();
handlerList.add(handler);
((BindingProvider)port).getBinding().setHandlerChain(handlerList);
```

バインディングファイルに記述して登録する場合のサンプルは次のようになります。

sample-handler.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
        <handler-chain>
            <handler>
                <handler-class>XXXX.SampleSOAPHandler</handler-class>
            </handler>
        </handler-chain>
</handler-chains>
```

<handler-class>には、登録するハンドラのクラス名を記述します。

上記のバインディングファイルを、javax.jws.HandlerChain アノテーションで指定します。

```
import javax.jws.HandlerChain;
@HandlerChain(file="sample-handler.xml")
```

バインディングファイルは、クラスパスの通った場所、たとえば、WAR ファイルの WEB－INF/classes/配下などに置いておく必要があります。

複数のハンドラが登録された場合、以下の順番で呼び出されます。

- アウトバウンドメッセージの場合

  Logical Handler が登録されていれば、登録された順番に呼び出されます。その後、Protocol Handler が登録されていれば、登録された順番に呼び出されます。

- インバウンドメッセージの場合

  Protocol Handler が登録されていれば、登録された逆順で呼び出されます。その後、Logical Handler が登録されていれば、登録された逆順で呼び出されます。

2.1.12.6. コマンドインタフェース

ここでは、JAX-WS が提供するコマンドについて説明します。

wsimport

wsimport は、WSDL から、JAX-WS クライアントで利用するスタブやサービスクラス、および Web サービスの動作に必要なクラスを生成するために利用します。

利用方法

コマンドおよび ant のタスクとして実行します。

コマンドとして利用する場合は、以下のコマンドを実行してください。

Windows の場合：

```
> wsimport.bat [option] <wsdl>
```

Unix の場合：

```
$ wsimport.sh [option] <wsdl>
```

<wsdl>には Web サービスの wsdl を指定します。

[option]に指定できるオプションは後述の wsimport オプション一覧を参照してください。

ant のタスクとして実行する場合は、wsimport タスクを記述します。以下は wsimport タスクの例です。

```
<wsimport
      wsdl="..."
      destdir="directory for generated class files"
      sourcedestdir="directory for generated source files"
      keep="true|false"
      extension="true|false"
      verbose="true|false"
      version="true|false"
      wsdlLocation="..."
      catalog="catalog file"
      package="package name"
      <binding dir="..."
      includes="..." />
</wsimport>
```

各属性に指定する値は後述の wsimport オプション一覧を参照してください。

wsimport タスクを使用するためには、次の<taskdef>要素をプロジェクトに追加しておく必要があります。

```
<taskdef name="wsimport" classname="com.nec.webotx.webservice.tools.ws.ant.WsImport">
      <classpath path="jaxws.classpath"/>
</taskdef>
```

jaxws.classpath には、build.xml 内で定義した wosv-ws.jar および JDK 付属の tools.jar を含むクラスパス定義への参照を指定してください。

wsimport オプション一覧

コマンドラインオプション	Ant 属性	説明
<wsdl>	wsdl	サービスの wsdl ファイルを指定します。
-d <directory>	destdir	生成される出力ファイルの格納先を指定します。
-b <path>	binding	外部の JAX-WS または JAXB のバインディングファイルを指定します。ファイルごとに-b が必要となります。
-catalog	catalog	外部エンティティ参照を解決するためのカタログファイルを指定します。
-extension	extension	(仕様で定義されていない)ベンダ固有の拡張機能を許可します。現在のところ、ベンダ拡張の非標準 SOAP1.2 バインディングを使用する際に指定します。
-help	-	ヘルプを表示します。
-httpproxy:<host>:<port>	-	HTTP プロキシサーバを指定します。デフォルトのポートは 8080 です。
-keep	keep	生成したソースファイルを削除しません。
-p	package	ターゲットパッケージを指定します。このオプションを指定した場合、すべての wsdl/スキーマバインディングのカスタマイズや、仕様で規定されているデフォルトのパッケージ名アルゴリズムより、この値が優先されます。
-s <directory>	sourcedestdir	生成したソースファイルの格納先を指定します。
-verbose	verbose	コンパイラの出力メッセージを表示します。
-version	-	バージョン情報を表示します。
-wsdllocation <location>	wsdllocation	WebServiceClient アノテーションの.wsdlLocation に設定する値を指定します。

wsgen

wsgen は、Web サービスエンドポイント実装クラスから、Web サービスの動作に必要なクラスおよび WSDL を作成するために利用します。

利用方法

コマンドまたは ant のタスクとして実行します。

コマンドとして利用する場合は、以下のコマンドを実行してください。

Windows の場合:

```
> wsgen.bat [option] <SEI>
```

Unix の場合:

```
$ wsgen.sh [option] <SEI>
```

<SEI>にはサービスエンドポイントの実装クラスを指定します。

[option]に指定できるオプションは後述の wsgen オプション一覧を参照してください。

ant のタスクとして実行する場合は、wsgen タスクを記述します。以下は wsgen タスクの例です。

```
<wsgen
        sei="..."
        destdir="directory for generated class files"
        classpath="classpath" | cp="classpath"
        resourcedestdir="directory for generated resource files such as WSDLs"
        sourcedestdir="directory for generated source files"
        keep="true|false"
        verbose="true|false"
        genwsdl="true|false"
        protocol="soap1.1|Xsoap1.2"
        servicename="..."
        portname="...">
        extension="true|false"
        <classpath refid="..."/>
</wsgen>
```

各属性に指定する値は後述の wsgen オプション一覧を参照してください。

wsgen タスクを使用するためには、次の<taskdef>要素をプロジェクトに追加しておく必要があります。

```
<taskdef name="wsgen" classname="com.nec.webotx.webservice.tools.ws.ant.WsGen">
    <classpath path="jaxws.classpath"/>
</taskdef>
```

jaxws.classpath には、build.xml 内で定義した wosv-ws.jar および JDK 付属の tools.jar を含むクラスパス定義への参照を指定してください。

wsgen オプション一覧

コマンドラインオプション	Ant 属性	説明
<SEI>	sei	サービスエンドポイントの実装クラス名を指定します。
-classpath <path>	classpath	入力クラスファイルの検索場所を指定します。
-cp <path>	cp	-classpath と同じです。
-d <directory>	destdir	生成される出力ファイルの格納先を指定します。
-extension	extension	(仕様で定義されていない)ベンダ固有の拡張機能を許可します。現在のところ、-wsdl:Xsoap1.2 を使用する際に必要なオプションとなります。
-help	-	ヘルプを表示します。
-keep	keep	生成したソースファイルを削除しません。
-r <directory>	resourcedestdir	WSDL 等のリソースファイルの格納先を指定します。常に-wsdl オプションと組み合わせて指定します。
-s <directory>	sourcedestdir	生成したソースファイルの格納先を指定します。
-verbose	verbose	コンパイラの出力メッセージを表示します。
-version	-	バージョン情報を表示します。

-wsdl[:protocol]	protocol	wsdl を生成します(デフォルトでは生成しません)。protocol には wsdl:binding で使用するプロトコルを指定します。指定できるプロトコルは"soap1.1"(デフォルト)および"Xsoap1.2"です。
-servicename <name>	servicename	wsdl に出力される wsdl:service の名前を指定します。常に-wsdl オプションと組み合わせて指定します。
-portname <name>	portname	wsdl に出力される wsdl:port の名前を指定します。常に-wsdl オプションと組み合わせて指定します。

apt

apt は、JDK 5.0 付属のコマンドで、JSR175 で規定されているアノテーションを処理するためのツールです。Web サービスエンドポイント実装クラスのソースから、Web サービスの動作に必要なクラスを作成するために利用します。

利用方法

ant のタスクとして実行します。コマンドで実行する場合は JDK 付属のマニュアルを参照してください。

apt を ant のタスクとして実行する場合は、apt タスクを記述します。以下は apt タスクの例です。

```
<apt
      verbose="true|false"
      classpath="classpath"
      destdir="directory for generated class files"
      sourcedestdir="directory for generated source files"
      nocompile="true|false"
      print="true|false"
      factorypath="<path>"
      factory="name of AnnotationProcessorFactory to use"
      <option key="keyname" value="keyvalue"/>
      <source ... > ... </source>
</apt>
<!-- javac オプションは省略 -->
```

各属性に指定する値は後述の apt オプション一覧を参照してください。

apt タスクを使用するためには、次の<taskdef>要素をプロジェクトに追加しておく必要があります。

```
<taskdef name="apt" classname="com.nec.webotx.webservice.tools.ws.ant.Apt">
      <classpath path="jaxws.classpath"/>
</taskdef>
```

jaxws.classpath には、build.xml 内で定義した wosv-ws.jar および JDK 付属の tools.jar を含む、クラスパス定義への参照を指定します。

apt オプション一覧(javac オプションは省略)

Ant 属性	説明
source	サービスエンドポイントの実装クラスのソースファイルを指定します。
classpath	入力クラスファイルの検索場所を指定します。
cp	-classpath と同じです。

destdir	生成される出力ファイルの格納先を指定します。
sourcedestdir	生成したソースファイルの格納先を指定します。
option	アノテーションプロセッサに渡されるオプションです。
nocompile	ソースファイルをコンパイルしません。
print	指定した型のテキスト表示を出力します。
factorypath	AnnotationProcessorFactory の検索場所を指定します。
factory	使用する AnnotationProcessorFactory の名前を指定します。 デフォルトの検索処理をバイパスします。

2.1.13.高信頼メッセージング

ここでは、WS-RM（WS-ReliableMessaging）、WS-R（Web Service Reliability）機能を利用した高信頼なWeb サービスを構築する手順について説明します。なお、本機能の Web サービス基盤技術は JAX-WS に変更されていますので、ご注意下さい。

WS-RM・WS-Rの機能

Web サービスに用いられる SOAP のメッセージは、多くの場合 HTTP プロトコルにより運ばれます。HTTP プロトコルを利用した場合は、ネットワークの一時的な障害や通信相手のマシンが停止しているといった場合にメッセージの到着を保証できません。従来、メッセージを確実に届けなければならない場合は、通信エラーが発生した場合に再送処理する実装を独自に組み込む必要がありました。

WS-RM や WS-R による高信頼メッセージングは、メッセージの欠落を回避するための再送や、再送によって発生する同一メッセージの重複および送信順序と異なって到着する現象である順序の乱れの防止機能を提供します。

WS-RM・WS-Rのインストール

ここでは永続化のために使用するデータベースの準備や高信頼メッセージングプログラムの配備をします。

設定ファイルの更新

（1）「<WebOTX_DIR>/lib/wsr/build.properties」をテキストエディタで開き、インストールされている環境に合わせて値を変更します。各パラメータの意味は次の通りです。

プロパティ名	説明
j2ee.home	WebOTX のインストールディレクトリを指定します。
admin.host	WebOTX のインストールホスト名を指定します。通常は localhost です。
admin.user	WebOTX の admin ユーザ名を指定します。インストール後の初期値は admin です。
admin.password	WebOTX の admin ユーザのパスワードを指定します。
admin.port	WebOTX の admin ポート番号を指定します。
admin.apg	アプリケーションプロセスグループ名を指定します。
admin.pg	プロセスグループ名を指定します。
connection_pool_name	RM サービスリソースアダプタのコネクションプール名を指定します(通常は変更する必要はありません)。
connection_resource_name	RM サービスリソースアダプタのコネクタリソース名を指定します(通常は変更する必要はありません)。
webotx.tool.otxadmin	otxadmin コマンドのファイル名です。
server.start.delay	サーバ起動後の遅延時間です。単位は分です。

（2）「<WebOTX_DIR>/config/wsr/config.properties」をテキストエディタで開き、インストールされている環境に合わせて値を変更します。各パラメータの意味は次の通りです。ここではインストールするために必要なプロパティのみを説明します。以下に説明のないプロパティは変更しないでください。

プロパティ名	説明
dataStore.webotx.jndiName	上記で作成した JDBC データソースの JNDI 名を指定します。
protocol.destination.baseUrl	RM サービスの制御メッセージ受信用の URL URL 表記(http://localhost/rmm/)中の localhost の部分をインストールしたマシンのホスト名もしくは IP アドレスに変更します。
protocol.reply.pattern	Response, Callback, Poll のいずれかを指定します。
protocol.wsr.replyTo.bareURI	protocol.reply.pattern が Callback のときに protocol.destination.baseUrl と同じ値を指定します。

MEMO

protocol.destination.baseUrl は通信相手先で、送達確認などの制御メッセージを送り返すために使用しますので、外部のマシンで有効な名前またはアドレスを指定してください。**localhost** や **127.0.0.1** といったホスト名やアドレスは指定できません。

protocol.wsx.replyTo.address	protocol.reply.pattern が Callback のときに protocol.destination.baseUrl と同じ値を指定します。
soap.thread.pool.size	送信スレッドの最大数。デフォルト値は10です。
expiration.groupInfo.duration	セッション、メッセージの有効期限です。デフォルト値は 1 日です。メッセージ送信後、受信側が 1 日以上後にメッセージを取得するようなシステムでは、この値を変更します。値は Duration 文字表記で記述します。 具体的な記述方法は、 http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#duration を参照してください。 簡単な例を示します。 P1D： 1 日 P1M： 1ヶ月 P1Y： 1 年 PT1H： 1 時間 PT30M： 30分 PT60S： 60秒 これらを組み合わせて記述することもできます。 P1Y2M3DT4H50M20S： 1 年2ヶ月3日4時間50分20秒 また、PT0S は指定できません。
expiration.deadQueue.timeToLive	メッセージが有効期限切れになってから削除されるまでの時間を指定します。単位はミリ秒です。

MEMO

protocol.reply.pattern では高信頼メッセージングプロトコルの制御メッセージの受け取り方法のオプションを選択します。
Response: HTTP の response を使って制御メッセージの通信をします。
Callback: 送信とは逆向きの HTTP コネクションを受信側から新たに作って制御メッセージの通信をします。
Poll: メッセージ送信後、送信側から新たな HTTP コネクションを作って制御メッセージを受け取る。

JDBCドライバのコピーとデータソースの登録

インストール対象のドメインが起動されているときは、一旦終了します。

Oracle の JDBC ドライバライブラリを、<WebOTX_DIR>/domains/<ドメイン名>/lib/ext へコピーします。

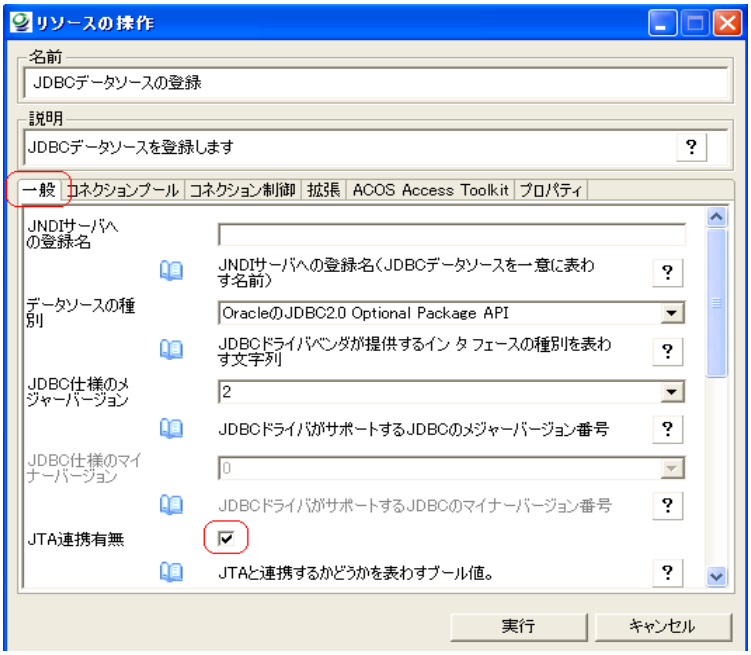
Oracle 9i の場合、<ORACLE_HOME>/jdbc/lib ディレクトリにある ojdbc14.jar と nls_charset12.jar

Oracle 10g の場合、<ORACLE_HOME>/jdbc/lib ディレクトリにある ojdbc14.jar と orai18n.jar

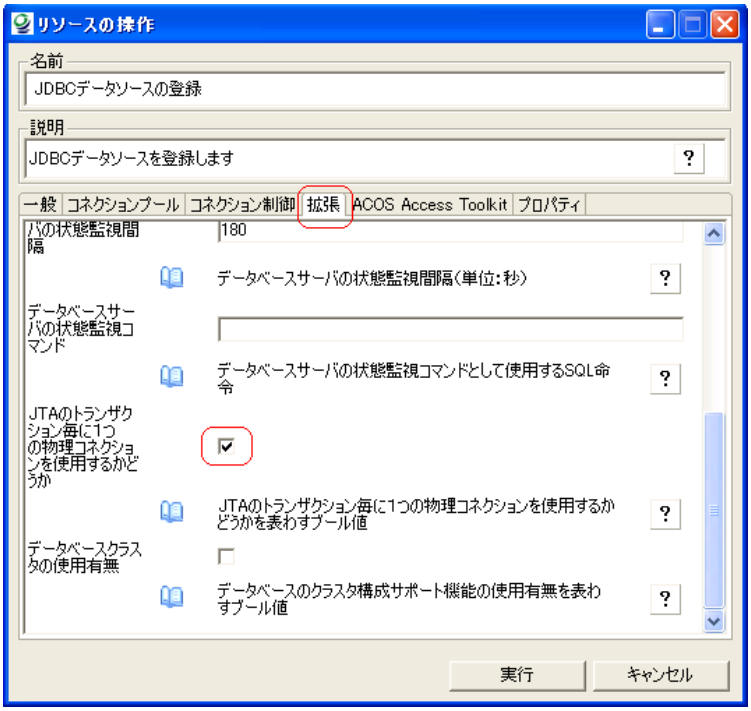
コピー後、インストール対象のドメインを起動します。

高信頼メッセージング機能がデータベースにアクセスするために使用する JDBC データソースを、WebOTX 統合運用管理ツール（パースペクティブ）を用いて登録します。統合運用管理ツールの画面上で**インストール対象のドメイン｜リソース**をクリックします。次に「**JDBC データソース**」を右クリックして「**JDBC データソースの登録**」を選択します。「**リソースの操作**」画面が開きます。この画面を用いて JDBC データソースを登録します。このとき、次の 2 点をチェックしてから登録します。

「一般」タブの「JTA 連携有無」をチェックします。



「拡張」タブの「JTA のトランザクション毎に1つの物理コネクションを使用するかどうか」をチェックする。



データベースのユーザ登録

高信頼メッセージング機能が使用するデータベースのユーザ登録を行います。

以下の SQL 文を実行します。**「ユーザ名」**の部分には前述の JDBC データソースの登録で指定したユーザ名を指定します。

```
SQL> CREATE USER ユーザ名 IDENTIFIED BY ユーザ名;
SQL> GRANT CREATE SESSION, CREATE TABLE, unlimited tablespace TO ユーザ名;
SQL> GRANT EXECUTE ON SYS.DBMS_SYSTEM TO ユーザ名;
SQL> GRANT CREATE TYPE, CREATE PROCEDURE, EXECUTE ANY PROCEDURE, EXECUTE
ANY TYPE TO ユーザ名;
```

MEMO

SQL 文を実行するときには、データベース管理システムに付属するコマンドを使用します(Oracle の場合は sqlplus など)

データベーステーブルの構築

rmmadmin コマンドを使ってテーブルを生成します。

・Windows

```
> <WebOTX_DIR>¥bin¥rmmadmin.bat create-table
```

・UNIX

```
$ <WebOTX_DIR>/bin/rmmadmin.sh create-table
```

高信頼メッセージングプログラムの配備

次のコマンドで高信頼メッセージングのプログラムを配備します。デフォルトドメイン(domain1)以外のドメインに配備する場合、エディタで次のコマンドのドメイン名を環境に合わせて変更してください。

●Standard/Enterprise Edition の場合

・Windows

```
> <WebOTX_DIR>¥lib¥wsr¥install_Std.bat
```

・UNIX

```
$ <WebOTX_DIR>/lib/wsr/install_Std.sh
```

●Standard-J Edition の場合

・Windows

```
> <WebOTX_DIR>¥lib¥wsr¥install_StdJ.bat
```

・UNIX

```
$ <WebOTX_DIR>/lib/wsr/install_StdJ.sh
```

Standard Edition/Enterprise Editionで動作させる場合の注意点

WS-RM・WS-R を Standard Edition、または Enterprise Edition で動作させる場合、ドメイン内全てのプロセスグループの Java システムプロパティに以下のキーと値を入力する必要があります。

```
キー：wsrm.configuration
値：%WebOTX_HOME%/domains/domain1/config/RM
```

この設定を行わない場合、WS-RM・WS-R が動作するプロセスグループ以外が動作しない場合があります。

WS-RM・WS-Rに対応したアプリケーションについて

高信頼メッセージング機能は、JAX-WS API から利用することができます。高信頼メッセージングを利用する場合であってもアプリケーションの記述内容は、通常の JAX-WS アプリケーションと変わりません。

ただし、高信頼メッセージング機能は非同期通信機能のみ提供しますので、JAX-WS で呼び出すオペレーションは、結果を持たない一方通行の呼び出しに限定されます。

WS-RM・WS-Rクライアントの利用方法

高信頼メッセージングのクライアント機能を利用する方法を説明します。

ハンドラの追加

クライアント機能を利用するためには、高信頼メッセージングが提供する「ハンドラ」が JAX-WS 実行環境から呼び出されるようにします。その方法はいくつかありますが、そのうちの 1 つである、WSDL のカスタムファイルを利用する方法を説明します。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bindings
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="http://your-hostname/your-servlet-context/your-url-pattern?wsdl"
    xmlns="http://java.sun.com/xml/ns/jaxws">
    <bindings node="wsdl:definitions">
        <package name="your-package"/>
    </bindings>
    <bindings node="wsdl:definitions"
xmlns:javaee="http://java.sun.com/xml/ns/javaee">
        <javaee:handler-chains>
            <javaee:handler-chain>
                <javaee:handler>
<javaee:handler-class>com.nec.webotx.webservice.wsrm.transport.client.RMHandler</javaee:handler-class>
                </javaee:handler>
            </javaee:handler-chain>
        </javaee:handler-chains>
    </bindings>
</bindings>
```

javaee:handler-class 要素に RMHandler クラスを指定することで高信頼メッセージング機能を利用することができます。

上記内容の your-で始まる箇所を開発するアプリケーションの WSDL 定義に合わせて変更したファイルを作成します。赤枠の中が高信頼メッセージング機能に固有の記述です。他はハンドラを追加する際の

MEMO

RMHandler のパッケージ名を含む正式名称は次のとおりで

一般的な記述です。

JAX-WS の機能を利用してスタブを生成します。このとき上記ファイルを WSDL ファイルとともに指定します。生成処理の一例ですが、具体的には Wsimport で処理する際に binding の指定を行ないます。上記ファイルを置いたディレクトリが custom-dir、また、上記ファイルのファイル名が custom-file に設定されていることを仮定しています。Ant を使った生成処理の記述例は以下のとおりです。

```
    <taskdef name="wsimport"
classname="com.nec.webotx.webservice.tools.ws.ant.Wsimport">
        <classpath refid="jaxws.classpath"/>
    </taskdef>

    <target name="generate-client">
        <wsimport
                debug="${debug}"
                verbose="${verbose}"
                keep="${keep}"
                extension="${extension}"
                destdir="${your-destdir}"
                wsdl="${your-wsdl}">

            <binding dir="${custom-dir}" includes="${custom-file}"/>

        </wsimport>
    </target>
```

アプリケーションの記述や配備方法は通常の JAX-WS アプリケーションと変わりません。

クライアント側キューの生成

キューを生成します。

運用管理コンソールから admin ユーザでログインし、[WebOTX 管理ドメイン]→[ドメイン名(.domain1 など)]→[アプリケーションサーバ]→[RM サービス]を選択します。選択した管理対象に関する操作情報から[送信キューの生成]を選択し、エンドポイント URI の値とトランスポートの値を入れてから実行します。

（例）

・送信キュー

エンドポイント URI：http://your-hostname/your-servlet-context/your-url-pattern

トランスポート：WS-Reliability/1.1

以上でクライアントで高信頼メッセージング機能を使用する際の準備は完了です。

WS-RM・WS-Rサーバの利用方法

高信頼メッセージングのサーバ機能を利用する方法を説明します。

web.xmlの記述変更

サーバ機能を利用する際には JAX-WS アプリケーションを war ファイルとして取りまとめる際に作成する web.xml ファイルの記述内容を以下の例のように変更します。

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

<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">
  <description>your-description</description>
  <display-name>your-display-name</display-name>
  <listener>
<listener-class>com.nec.webotx.webservice.xml.ws.transport.http.servlet.WSServletContextListener</listener-class>
  </listener>
  <servlet>
    <description>your-description</description>
    <display-name>your-display-name</display-name>
```

す。
com.nec.webotx.webservice.wsrm.transport.client.RMHandler

MEMO
サーバ側で作成する受信キューとここで作成する送信キューのエンドポイント URI の値は文字列として一致するようにしてください。

MEMO
WSServletContextListener のパッケージ名を含む正式名称は次のとおりです。
com.nec.webotx.webservice.xml.ws.transport.http.servlet.WSServletContextListener
RMServlet のパッケージ名を含む正式名称は次のとおりです。
com.nec.webotx.webservice.wsrm.trans

```
    <servlet-name>your-servlet-name</servlet-name>
  <servlet-class>com.nec.webotx.webservice.wsrm.transport.servlet.RMServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>your-servlet-name</servlet-name>
    <url-pattern>/your-url-pattern</url-pattern>
  </servlet-mapping>
```

port.servlet.RMServlet

listerner-class 要素に WSServletContextListener を指定します。また、servlet-class 要素に RMServlet を指定します。

サーバ側キューの生成

キューを生成します。

運用管理コンソールから admin ユーザでログインし、[WebOTX 管理ドメイン]→[ドメイン名(.domain1 など)]→[アプリケーションサーバ]→[rm-service]を選択します。選択した管理対象に関する操作情報から[受信キューの生成]を選択し、エンドポイント URI の値を入れてから実行します。

（例）

・受信キュー

エンドポイント URI：http://your-hostname/your-servlet-context/your-url-pattern

MEMO

クライアント側で作成した送信キューとここで作成する受信キューのエンドポイント URI の値は文字列として一致するようにしてください。

WS-RM・WS-Rのアンインストール

次のコマンドを実行すると、高信頼メッセージング機能をアンインストールできます。WebOTX が起動している状態で実行します。デフォルトドメイン（domain1）以外のドメインの配備を解除する場合、テキストエディタでコマンドを開いてドメイン名を環境に合わせて変更してください。

●Standard/Enterprise Edition の場合

・Windows

```
> <WebOTX_DIR>¥lib¥wsr¥uninstall_Std.bat
```

・UNIX

```
$ <WebOTX_DIR>/lib/wsr/uninstall_Std.sh
```

●Standard-J Edition の場合

・Windows

```
> <WebOTX_DIR>¥lib¥wsr¥uninstall_StdJ.bat
```

・UNIX

```
$ <WebOTX_DIR>/lib/wsr/uninstall_StdJ.sh
```

次のコマンドでテーブルを削除します。

・Windows

```
> <WebOTX_DIR>¥bin¥rmmadmin.bat drop-table
```

・UNIX

```
$ <WebOTX_DIR>/bin/rmmadmin.sh drop-table
```