# WebObjects Web Services Programming Guide

**Internet & Web > Web Services**

**2007-07-11**

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010



Simultaneously published in the United States and Canada.

# Contents

# Figures, Tables, and Listings

# Introduction to WebObjects Web Services Programming Guide

**Important:** The following tools are deprecated and no longer supported in WebObjects 5.4 and later: EOModeler, RuleEditor, WebObjects Builder, WOALauncher, and Java Client. WebObjects templates are not available for creating new projects in Xcode on Mac OS X v10.5 and later.

**Note:** This document was previously titled *Web Services* .

Web services

SOAP

## Organization of This Document

This document has the following chapters:



## See Also

The following list itemizes resources you can use to increase your Web services knowledge.

- The AmazonClient project in `/Developer/Examples/JavaWebObjects/AmazonClient` is an implementation of a client for Amazon.com Web services.
- *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI* (Sams) gives great detail on the elements of Web-service development and deployment.
- *Architecting Web Services* (Apress) provides a high-level view of Web-service development.
- *Java & XML* (O'Reilly) introduces you to XML and processing XML documents using SAX (Simple API for XML).
- *Web Services Routing Protocol (WS-Routing)* (http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-routing.asp ).

- *Simple Object Access Protocol (SOAP) 1.1* (http://www.w3.org/TR ).
- Axis (http://xml.apache.org/axis ) is the SOAP implementation used in WebObjects.
- *Web Services Security Core Specification* at http://www.oasis-open.org/committees/wss .
- *Canonical XML Version 1.0* (http://www.w3.org/TR ).
- *Exclusive XML Canonicalization Version 1.0* (http://www.w3.org/TR ).
- *Web Services Description Language (1.1)* (http://www.w3.org/TR/wsdl ).

In this document, SOAP (Simple Object Access Protocol) refers to SOAP version 1.1 of the specification.

# Web Services Overview

You can think of Web services as distributed applications. Instead of creating an instance of a class and invoking its methods, a Web service **consumer** locates a Web service and invokes the operations it provides. The Web service **provider** (the application implementing the Web service) can be on the same Java virtual machine as the one using it, or it can be thousands of miles away. Furthermore, the applications may be written in different languages and running in disparate platforms. Because of this, Web service consumers as well as Web service providers need a way of transferring information that is language and platform independent. This is where SOAP lends a hand.

Web services are based on SOAP (Simple Object Access Protocol). It provides an infrastructure for the exchange of structured data in a distributed environment. SOAP itself is based on XML (Extensible Markup Language). XML is an SGML (Standard Generalized Markup Language)-based language that facilitates the structuring of data in documents. In addition, data elements in XML documents provide information about the data they contain through element names and attributes.

SOAP was created to facilitate the exchange of information by heterogeneous systems. XML provides it with structure through schemas and element scope through namespaces. SOAP is a transport-agnostic protocol: messages can be sent using HTTP, SMTP, and other protocols. For more on XML, including XML Schema and XML Namespaces, see *Extensible Markup Language (XML)* at http://www.w3.org/XML .

This chapter introduces Web service concepts. If you're familiar with Web service technology, you can go to the next chapter.

The chapter has the following sections:



## What Are Web Services?

Web services provide an implementation-independent way for applications to communicate with each other. Currently, many companies use electronic-data-interchange (EDI) systems to communicate with their business partners. EDI, however, requires the use of slow modems and dedicated phone lines. Also, a change in the structure of the data exchanged requires that the systems of all partners involved be updated. Web services, which are based on SOAP messages that wrap XML documents, provide a flexible infrastructure that leverages the ubiquitous HTTP (or HTTPS) over TCP/IP. This means that your organization probably has all the hardware

and software infrastructure needed to deploy Web services already. In addition, thanks to XML's structure and flexibility, each partner can extract only the information it needs from a message, which gives participants a great deal of freedom.

But Web services provide more than an information-exchange system. When an application implements some of its functionality using Web services, it becomes more than the sum of its parts. For example, you can create a Web service **operation** that uses a Web service operation from another provider to give its consumers (also known as *service requestors* ) information tailored to their needs. Web service operations are akin to the methods of a Java class; a provider is an entity that publishes a Web service, while the entities that use the Web service are called consumers.

Current Web service technology allows an organization to easily integrate its systems, creating an enterprise-wide solution that leverages the work that is performed best by smaller groups within the enterprise. For example, the Payroll system is the one that should deal with an employee's compensation, while the Human Resources system is more appropriate for the management of vacation and sick-leave time. However, an Employee Information system should gather the information that both the Payroll and Human Resources systems contain, but should not duplicate it. The Employee Information system could display a window or Web page that an employee can view to analyze both her salary and accrued vacation time, without having to directly access the data stores used by the other two systems. Payroll and vacation information would be available through Web service operations provided by separate applications tailored to their particular objectives.

Web services can also be deployed over the Internet; however, you should ensure that sensitive information is not compromised. A SOAP message can hop through several computers across a network before reaching its destination, which exposes it to be viewed and modified by entities that you don't know about. There are several standards and specifications that help you protect the messages you send and to make sure that the messages you receive have not been compromised. See "Security in Web Services" (page 21) for more information.

What Web services really provide is access to business logic. This business logic can be implemented in any language. Most companies implementing Web services for the first time only add a Web service front end to their existing applications. WebObjects makes this easy.

## Web Service Discovery

The Web Services Description Language (WSDL) is an XML-based language used to describe a Web service. This description allows an application to dynamically determine a Web service's capabilities; for example, the operations it provides, their parameters, return values, and so forth. A UDDI (Universal Description, Discovery and Integration) repository is a searchable directory of Web services that Web service requestors can use to search for Web services and obtain their WSDL documents. WSDL documents, however, do not need to be published in a repository for consumers to take advantage of them. You can obtain a WSDL document through a Web page or an email message.

## Web Services and SOAP

SOAP is the messaging mechanism that you use when you consume Web service operations or provide Web service operations to your clients.

All Web service communication is done through SOAP messages. These messages have an envelope, represented by the `Envelope` element, and a body, enclosed by the `Body` element, containing the message's content. In addition, the `Envelope` can contain a `Header` element enclosing one or more header entries. The header mechanism is what provides SOAP with decentralized extensibility; this is how extensions such as Digital Signature and Web Services Security Core Language (WSS-Core) are implemented. For more information on signatures and security, see "Security in Web Services" (page 21) .

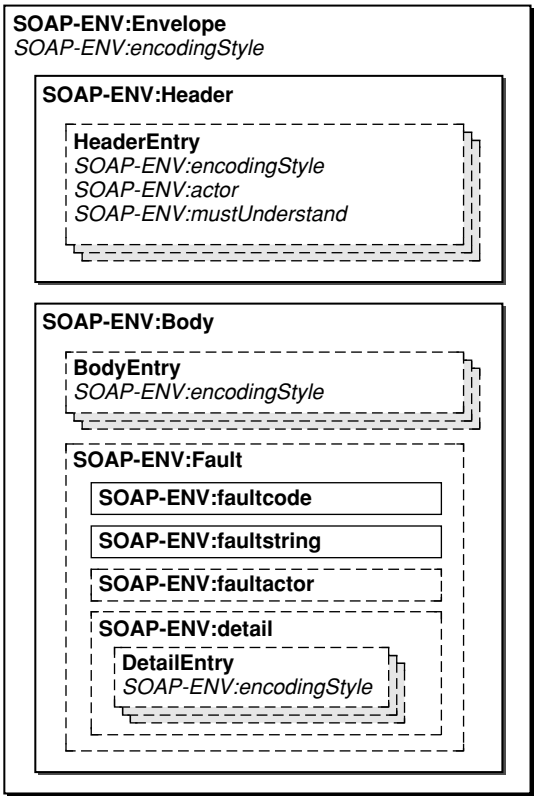
SOAP provides two ways for representing Web service operation invocations: RPC (Remote Procedure Call) messaging and document-style messaging. RPC messaging provides a way of representing method invocations in SOAP messages. Because of this, however, the structure of the messages representing operation invocations is fairly rigid. Document-style messaging, on the other hand, provides greater flexibility; it allows messages to contain arbitrary data elements. However, parsing such messages is more complicated.

Direct to Web Services (a technology that allows you to rapidly create Web services based on a data model) uses RPC because it allows the mapping of entity attributes to operation parameters. However, you can use either RPC messaging or document-style messaging in the Web services that you write. Take into account that document-style messaging requires specialized processing to extract the necessary data from the message. You must also implement error processing in case required data elements are not present in a message.

## Ingredients of a SOAP Message

As Figure 1-1 shows, a SOAP message, represented by the `Envelope` element, contains a mandatory `Body` element and an optional `Header` element. The `Body` element can contain a number of body entries. The optional `Fault` element is present only in messages that report a processing exception.

**Figure 1-1** Structure of a SOAP message




Listing 1-1 shows an RPC SOAP message.

**Listing 1-1** Example of an RPC SOAP message

```
<soapenv:Envelope
    xmlns:soapenv="soap_ns"
    xmlns:xsd="xml_schema_ns"
    xmlns:xsi="type_ns">
    <soapenv:Body>
        <ns1:getStockPrice
            xmlns:ns1="app_ns"
            soapenv:encodingStyle="encoding_ns">
            <stockSymbol xsi:type="xsd:string">AAPL</stockSymbol>
        </ns1:getStockPrice>
    </soapenv:Body>
</soapenv:Envelope>
```

Listing 1-2 shows a document-style SOAP message.

**Listing 1-2** Example of a document-style SOAP message

```
<soapenv:Envelope
    xmlns:soapenv="soap_ns"
    xmlns:xsd="xml_schema_ns"
    xmlns:xsi="type_ns">
    <soapenv:Body>
        <ns1:customerOrder
```

```
    soapenv:encodingStyle="encoding_ns"
    xmlns:ns1="app_ns">
    <order>
        <customer>
            <name>Plastic Pens, Inc.</name>
            <address>
                <street>123 Yukon Drive</street>
                <city>Phoenix</city>
                <state>AZ</state>
                <zip>85021</zip>
            </address>
        </customer>
        <orderInfo>
            <item>
                <partNumber>88</partNumber>
                <description>Blue pen</description>
                <quantity>250</quantity>
            </item>
            <item>
                <partNumber>563</partNumber>
                <description>Red stappler</description>
                <quantity>30</quantity>
            </item>
        </orderInfo>
    </order>
  </ns1:customerOrder>
</soapenv:Body>
</soapenv:Envelope>
```

Table 1-1 describes the elements of a SOAP message.

**Table 1-1** The elements of a SOAP message

| Element | Parent | Use | Description |
|---|---|---|---|
| `Envelope` | *None* | 1 | Root element of the message. |
| `Header` | `Envelope` | ? | Encloses header entries. |
| *Header entries* | `Header` | * | Heather entries provide additional information on the message's content. For example, digital signatures, authorization data, and so on. |
| `Body` | `Envelope` | 1 | Encloses the message's body entries. |
| *Body entries* | `Body` | * | Body entries make up the content of the message. Their element names depend on the message's content. |
| `Fault` | `Body` | ? | Body entry used to report a problem. When used, no other body entry can be present. |
| `faultcode` | `Fault` | 1 | Indicates the reason for the fault. Intended for application use. |
| `faultstring` | `Fault` | 1 | Human-readable version of the fault reason. |
| `faultactor` | `Fault` | ? | Indicates which entity along the message path raised the fault. |

| Element | Parent | Use | Description |
|---|---|---|---|
| `detail` | `Fault` | ? | Encloses detail entries. |
| *Detail entries* | `detail` | * | Contain application-specific information about the fault. |

All the attributes that the SOAP envelope schema defines are global (they are not associated with a particular element). Also, each element in a SOAP message is free to use any attribute, regardless of where it's defined, either in SOAP's schema or another one, which is one of SOAP's extensibility features. This means that elements are free to use any number of attributes. Table 1-2 describes the attributes that the SOAP specification defines.

**Table 1-2** Attributes defined in the SOAP specification

| Attribute | Value | Description |
|---|---|---|
| `actor` | A URI. | Specifies the entity that is to process the element. When absent, the actor is the ultimate recipient of the message. This attribute is used mainly to assign header entries to specific entities. |
| `mustUnderstand` | `"0"` or `"1"`. | Indicates whether the element's actor must process the element. When set to `1` and if the actor is unable to process the element, the actor must respond with a `Fault`. |
| `encodingStyle` | A list of URIs. | Indicates the encoding style used for the element's content. |

For more information on SOAP, see Simple Object Access Protocol (SOAP) at http://www.w3.org/TR .

# Web Service Description

For a consumer to be able to use a Web service's operations, it must know what operations the Web service provides, the parameters they take, the type of the values they return, and so on. With intimate knowledge of the Web service, you can write a Web service client that takes full advantage of the service. However, the idea behind Web services is to provide a way for an application to dynamically find a Web service that satisfies its requirements and to learn how to use it. One of the building blocks that bring that vision closer to reality is WSDL (Web Services Description Language). Like SOAP, WSDL is an XML-based language. A WSDL document tells a service requestor where a Web service is located and how to use it.

A WSDL document describes Web services in two ways: an abstract description or interface and a concrete implementation. The interface section provides a high-level description of the operations the Web services described by the document provide and their parameter types and return types. The implementation section binds each operation described in the interface section with its implementation (the methods that perform the work).

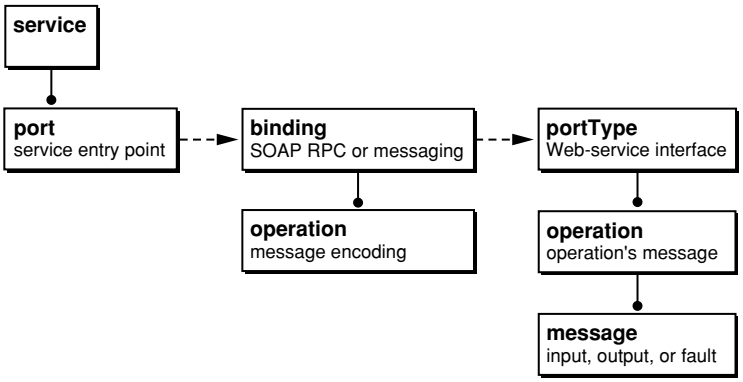
These are some of the XML elements that WSDL defines to describe Web services:

- `portType`: This element provides the interface to one or more Web services. It describes each operation provided by the services as a set of input (from the consumer) and output (from the provider) messages that can be generated as a result of invoking the operation. Included in the list of possible messages are fault messages, which the Web services way of notifying the occurrence of a processing problem or exception.

- `message` : This element describes a SOAP message. It lists the message's elements, which are referred to as *parts* , and their types.
- `types` : This element list all the data types used as parameters or return types used in `message` elements. Essentially, it's an XML Schema definition.
- `binding` : This element specifies the transport used to send messages between the Web services' consumers and their provider and implements a `portType` . It also defines the type of encoding used for each message.
- `port` : This element defines the URL that the Web service consumers use to access the Web service. It implements a `binding` .
- `service` : This element encloses one or more `port` elements.

Figure 1-2 shows the relationship between the major elements of a WSDL document. Missing from the figure are the `definitions` element, which is the root element of WSDL document and the `types` element.

**Figure 1-2** Organization of a WSDL document




For the most part, you don't have to concern yourself with reading or writing WSDL documents. WebObjects generates the WSDL documents needed to provide Web services and makes available methods to access the information contained in the WSDL documents for Web services you want to consume. For more information on WSDL, see *Web Services Description Language (WSDL)* at http://www.w3.org/TR .

# SOAP Engine

A SOAP engine (or processor) aids both consumers of Web services and their providers to accomplish their task without having to worry about the intricacies of SOAP message handling. As far as the consumer is concerned, it invokes an operation in a similar way a remote procedure call is invoked. The Web service provider needs to implement only the logic required by the business problem it solves. The consumer's SOAP processor converts the method invocation into a SOAP message. This message is transmitted through a transport, such as HTTP or SMTP, to the service provider's SOAP processor, which parses the message into a method invocation. The provider then executes the appropriate logic and gives the result to its SOAP processor, which parses the information into a SOAP response message. The message is transmitted through a transport to the consumer. It's SOAP processor parses the response message into a result object that it returns to the invoking entity.

Axis is the third generation of Apache SOAP (an implementation of SOAP from the Apache Software Foundation). Axis is a SOAP engine as well as a code generator and WSDL processing tool. WebObjects uses Axis to both provide and consume Web services.

The idea behind Axis is to serve as a bridge between your time-tested code and the world of Web services. By using Axis as its SOAP engine, WebObjects allows you to leverage the business logic you have already created and use it as the backbone of your Web services strategy.

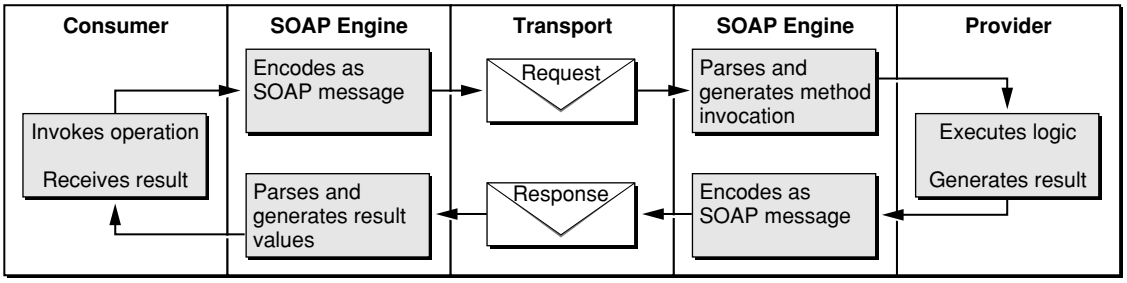
Axis processes SOAP messages using a series of handlers , which are classes responsible for processing a message or part of a message in a certain way. In fact, you are free to add your own handlers to customize message processing. For more information on Axis, visit http://xml.apache.org/axis .

## The Axis SOAP Engine

WebObjects uses the Axis framework to both serve and consume Web services. Axis is an interface between your business logic and the Web services world.

The Axis Web service processing model is shown in Figure 1-3 .

**Figure 1-3** The SOAP Message processing cycle




Axis implements a very extensible message processing model. It uses **handlers** and **handler chains** to allow its functionality to be tailored to a wide variety of situations and requirements. A handler is an atomic component that acts on a specific part of a SOAP message; for example, a handler can be in charge of performing authentication on the message's sender before allowing it to be processed by the provider. A special handler, the *pivot handler* (another name for the service's provider), is in charge of executing the Web service's logic. It's called pivot handler because it is where the message's processing cycle changes from request processing to response processing.

A handler chain is a group of handlers that can be viewed as a unit. An important concept to grasp is that handlers and handler chains are not Web service–specific. For example, you can develop handlers that process SOAP messages from transports other than HTTP or SMTP to increase security without having to change the Web service implementation. If you start now, you may be able to sell those handlers to others for a nice profit.

Handlers are simply Java classes that act on an `org.apache.axis.MessageContext` object. A MessageContext contains several useful objects, but the most important are the `requestMessage` and the `responseMessage`. Handlers processing incoming messages normally access the `requestMessage` object, while those processing the outgoing messages access the `responseMessage` object. However, Axis provides no restrictions; a handler can access and modify whatever it pleases. This is helpful if you need a handler to act both on incoming and outgoing messages.

Figure 1-4 shows the relationship between handlers and chains in Axis from the point of view of a Web service provider, while Figure 1-5 does the same from the perspective of a consumer.

**Figure 1-4** Web service processing—provider view

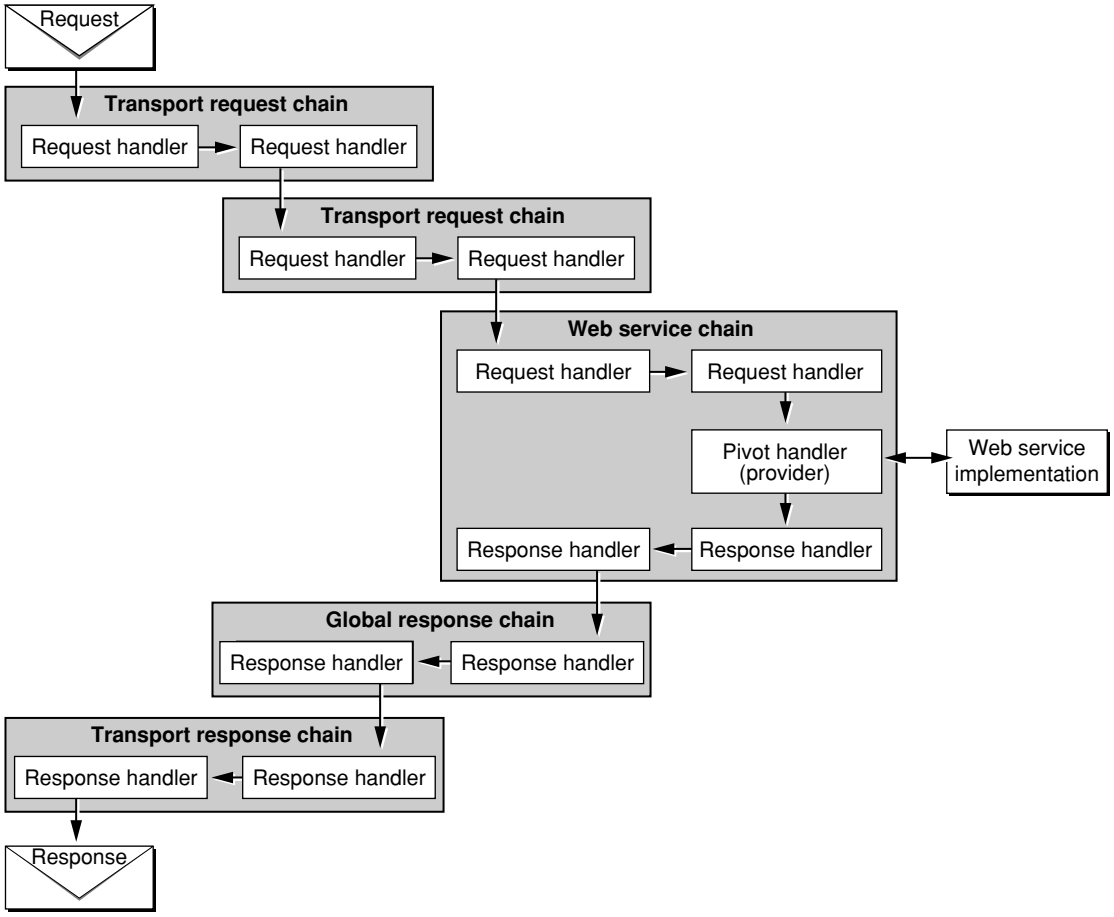
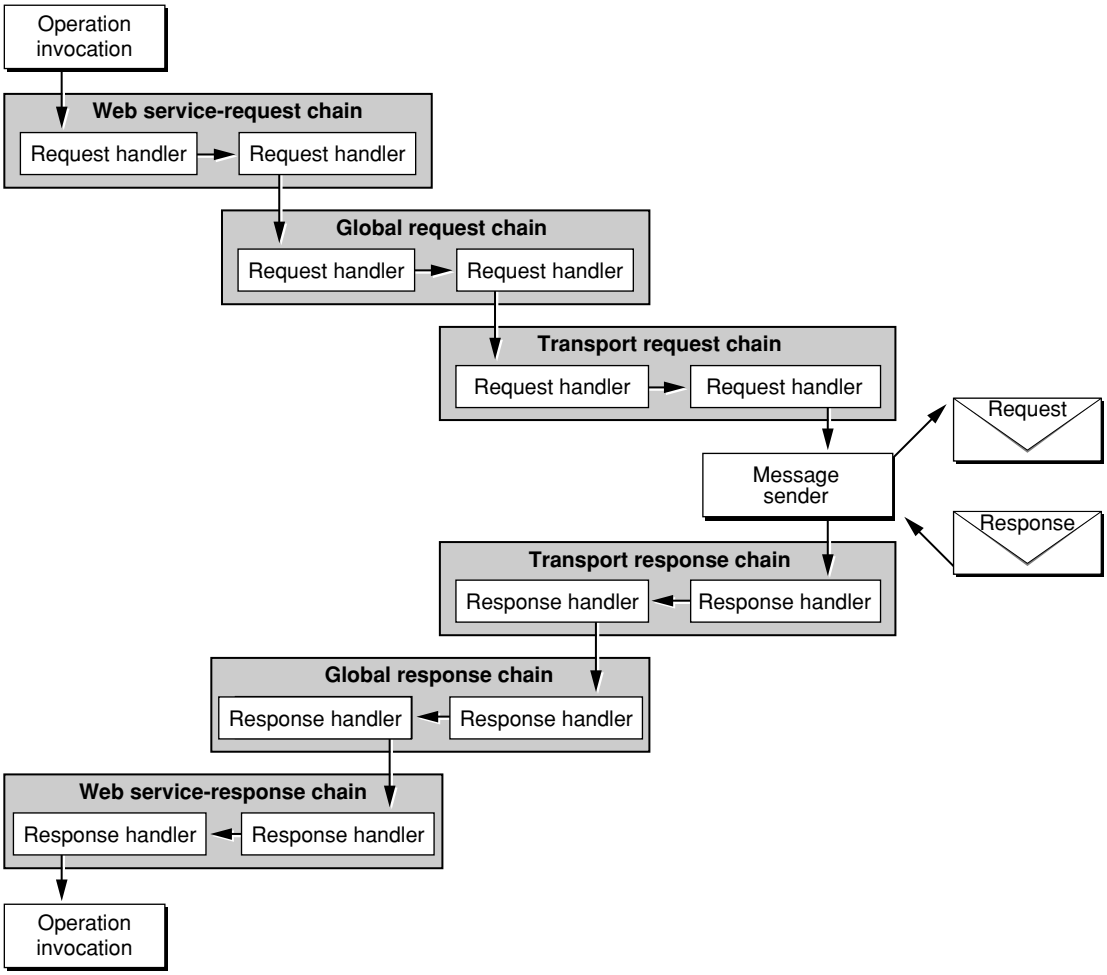


**Figure 1-5** Web service processing—consumer view




Notice that there are three types of chains: transport, global, and Web service. A transport chain can deal with issues specific to the transport used to send and receive SOAP messages. A global chain is one that processes every SOAP message, regardless of the transport used or the target Web service. Finally, a Web service chain is one tailored for a specific Web service. For more information on handlers and chains, see Axis's documentation at http://xml.apache.org/axis .

# Serialization and Deserialization of Objects

Complex classes require a custom serialization and deserialization strategy. WebObjects provides serializers and deserializers for some of its classes, as shown in Table 1-3 .

**Table 1-3** Serializers and deserializers provided in WebObjects

| Class | Serializer | Deserializer |
|---|---|---|
| `com.webobjects.eocontrol.EOEnterpriseObject` | x | x |

| Class | Serializer | Deserializer |
|---|---|---|
| com.webobjects.eocontrol.EOGlobalID | x | x |
| com.webobjects.foundation.NSArray | x | |
| com.webobjects.foundation.NSData | x | x |
| com.webobjects.foundation.NSDictionary | x | |
| com.webobjects.foundation.NSKeyValueCoding.Null | x | x |
| com.webobjects.foundation.NSRange | x | x |
| com.webobjects.foundation.NSSet | x | |
| com.webobjects.foundation.NSTimestamp | | x |
| com.webobjects.foundation.NSTimeZone | x | x |
| com.webobjects.webservices.support.xml.WOStringKeyMap | x | x |
| java.util.Calendar | | x |

If you have special classes that require a special serializer and deserializer, you have to create them. Writing serializer and deserializer classes is a simple process, but requires knowledge of SAX (Simple API for XML). To learn how to process SAX callbacks in your serializers and deserializers, see *Java & XML* (O'Reilly).

# Security in Web Services

When you need to transmit sensitive information across a network, you should consider the security implications of putting that information on the wire. You should use a level of security that corresponds with the sensitivity of the information you transmit. You should also take into account that adding security processing to message transmission has a detrimental effect on the performance of your applications.

One way you can address security concerns when serving Web services over an unsecure network, such as the Internet, is by exposing them only to trusted entities and specific IP addresses. However, this could reduce the speed at which you can add business partners to your enterprise because it would require manual configuration.

Using standard transport security, such as HTTPS (HyperText Transmission Protocol, Secure) or Secure Sockets Layer (SSL), you can ensure that a message is protected from eavesdroppers during transit. However, while these protocols protect a message as it's transmitted, they do not protect such data once it has reached its destination.

Web Services Security Core Specification (WSS-Core) is a specification that provides a security framework that can be used to secure Web services. It encompasses two major areas: digital signatures and encryption. With digital signatures you can ensure that a particular entity is the sender of a message, even when the message itself may be unprotected. You use encryption when you want to keep communications private, that is, when you want no entity other than the recipient to be able to read a message.

The following list itemizes the four areas that a security model for data communication should address:

- **Integrity** : Allows a message's recipient to ensure that a message hasn't been modified in transit.
- **Confidentiality** : Ensures that a message can be read only by the intended recipient.
- **Authentication** : Allows a recipient to ensure that a particular party is the originator of a message.
- **Nonrepudiation** : Allows a recipient to ensure that a sender cannot deny having sent a message.

Table 2-1 shows the level of security the protocols mentioned earlier provide.

**Table 2-1** Features provided by the major security approaches

| | Integrity | Confidentiality | Authentication | Nonrepudiation |
|---|---|---|---|---|
| SSL/HTTPS | x | x | | |
| Digital signature | x | | x | x |
| Encryption | x | x | x | x |

This chapter has the following sections:

- "Web Services Security" (page 22) provides an overview of the WSS-Core specification. It covers digital signatures and encryption.

- "Canonical XML Documents" (page 24) explains why it's necessary to normalize data before signing or encrypting it.

# Web Services Security

Web Services Security Core Language (WSS-Core) is a specification that specifies how to provide a security infrastructure to SOAP (Simple Object Access Protocol) messages. This quality-of-protection model can be extended to incorporate various security models and encryption technologies. Actually, the specification does not provide for the management of keys, certificates, or encryption mechanisms. It just specifies the elements that a SOAP message must have to ensure that it contains an acceptable level of protection against snooping and other types of security threats. For detailed information on WSS-Core, see *Web Services Security Core Specification* at http://www.oasis-open.org/committees/wss .

SOAP messages using WSS-Core include security tokens that represent claims. For example, a security token can indicate that a Web service client is operating under the "Mary" user name and that she's authorized to view certain data. The security token format is extensible, and a message can use more than one security-token format.

WSS-Core itself is an extension of SOAP. It adds elements to a SOAP message that can be used to enclose security-related information to its `Header` element. However, WSS-Core is not a security protocol and it does not provide one.

## Digital Signatures

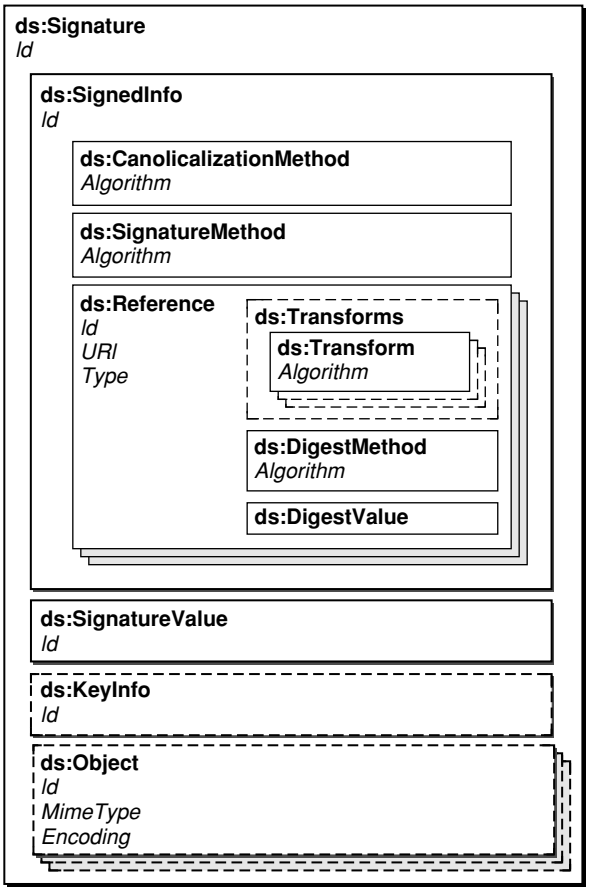
The XML Signature specification, which is based on public-key infrastructure (PKI) is an very important part of WSS-Core. It specifies the format of digital signatures in XML documents. XML Signature uses X.509 certificates (digital certificates) to authenticate the purported sender of a message. This is possible because digital certificates, which are issued by a Certification Authority, bind the subject identified by a certificate with its public key. For more information on PKI, see *PKI Basics: A Technical Perspective* at http://www.pkiforum.org .

The most secure approach to protect a SOAP message is to encrypt the message's payload. However, encrypting entire messages would bring about performance penalties. Therefore, as a general rule only a portion of a message or the message's hash is encoded using the sender's private key. (A hash is a number derived from a string such that any change to the string produces a different number.) This is the electronic version of a manual signature. That way, the message can be easily read by people, while the signature guarantees its integrity and identifies its sender. When the recipient receives a message, it creates a hash of the signed content, decrypts the signature using the sender's public key, and compares the two; if they match it means that the message is authentic.

XML Signature also adds support for integrity and nonrepudiation to XML-encoded messages. Digital signatures can sign individual elements of an XML document. This provides the actors involved in a transaction the ability to sign only the sections of the document for which they are responsible for. In addition, the signed elements do not have to be of the same type.

Figure 2-1 shows the structure of a digital signature element.

**Figure 2-1** Structure of a digital signature element




The `signature` element would normally be enclosed in a header-entry element named `Security` in a SOAP message. Listing 2-1 shows a SOAP message with a digital signature. Notice that the message's `Signature` element signs the content of the `Body` element.

**Listing 2-1** Example SOAP message using a digital signature

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/xx/utility">
    <SOAP-ENV:Header>
        <wsse:Security
            xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/xx/secext">
            <ds:Signature>
                <ds:SignedInfo>
                    <ds:CanonicalizationMethod
                        Algorithm="http://www/w3.org/TR/2000/CR-xml-c14n20001026" />
                    <ds:SignatureMethod
                        Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1" />
                    <ds:Reference URI="#Secret">
                        <ds:Transforms>
                            <ds:Transform
                        Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026" />
```

```
                    </ds:Transforms>
                    <ds:DigestMethod
                        Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
                    <ds:DigestValue>...</ds:DigestValue>
                </ds:Reference>
            </ds:SignedInfo>
            <ds:SignatureValue>...</ds:SignatureValue>
        </ds:Signature>
    </wsse:Security>
</SOAP-ENV:Header>
<SOAP-ENV:Body
    xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-12"
    wsu:Id="Secret">
    <m:GetCurrentTemperature
        xmlns:m="My-URI">
        <m:zipCode>80913</m:zipCode>
    </m:GetCurrentTemperature>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

For detailed information on XML Signature, see *An Introduction to XML Digital Signatures* at http://www.xml.com/lpt/a/2001/08/08/xmldsig.html .

### Encryption

You can encrypt parts of a message when you need to make sure that only its recipient is able to read it. In public-key cryptography, you encrypt data using your private key. The recipient then uses your public key to decrypt the encrypted information. This makes paramount the proper management of public and private keys. Enter public-key infrastructure (PKI).

PKI provides for the creation and issuance of certificates and public and private keys that message encryption and signing processes require. The W3C is developing a standard for XML-based key management. See *XML Key Management* at http://www.w3.org for details.

## Canonical XML Documents

During secure message processing, a message's recipient must ensure that the message was not modified during its travel from the sender. Digital signatures provide proof that the content they sign has not been altered. To make sure signed content—which is probably not encrypted—has not been modified, a recipient must create a hash of the signed information and apply the sender's public key. If the result matches the contents of the `SignatureValue` element, the content has not been modified.

The problem that arises is that an XML fragments that are functionally equivalent may not be identical. While XML goes a long way in defining a format for structure data, it still leaves room for ambiguity. Look at Listing 2-2 .

**Listing 2-2** Person elements

```
<person id="1001" group="admin">
    <last_name>Morton</last_name>
    <first_name>Ashley</last_name>
```

```
</person>

<person id='1001' group='admin">
    <first_name>
        Ashley
    </first_name>
    <last_name>
        Morton
    </last_name>
</person>
```

The `person` elements listed are functionally the same; however, they would produce different hashes.

XML fragments (or documents) in *canonical form* are the normalized versions of XML fragments. These fragments can be signed, verified, encrypted or decrypted with the assurance that formatting artifacts, such as superfluous spaces or attribute ordering, do not play a role in the process. In other words, when you encrypt a document in canonical form and then decrypt it, you get back a document that is almost identical to the original. For more information, see *Canonical XML* at http://www.w3.org/TR .

The canonical form of an XML element includes all the namespaces in the element's scope, even if they don't apply to the element being normalized. To solve this drawback, another initiative, called Exclusive XML Canonicalization, is used. Essentially, a document in *exclusive canonical form* includes only the information pertinent to the element being normalized. For more information, see *Exclusive XML Canonicalization* at http://www.w3.org/TR .

# Developing Web Service Applications

You can publish as Web service operations the public methods of any class that contains a no-argument constructor. Also, methods that correspond to document-style operations, must return an `org.w3c.dom.Document`. You can find documentation for these classes at http://xml.apache.org/axis , and http://www.w3.org respectively.

This chapter contains the following sections:



## Providing a Web Service

As a companion to this document, in `projects/Calculator`, you find the Calculator project. It's a simple WebObjects application project used to build an application that serves a Web service called Calculator. The service provides four operations: `add`, `subtract`, `multiply`, and `divide`. The operations take two parameters of type `double` and return a value of type `double`. The `Calculator.java` class, the workhorse of the Calculator Web service, is listed in Listing 3-1 .

**Listing 3-1** `Calculator.java` class in Calculator project

```
public class Calculator extends Object {

    public static double add(double addend1, double addend2) {
        double sum = addend1 + addend2;
        return sum;
    }

    public static double subtract(double minuend, double subtrahend) {
        double difference = minuend - subtrahend;
        return difference;
    }

    public static double multiply(double multiplicand1, double multiplicand2) {
        double product = multiplicand1 * multiplicand2;
        return product;
    }

    public static double divide(double dividend, double divisor) {
```

```
        double quotient = dividend / divisor;
        return quotient;
    }
}
```

To provide a Web service based on `Calculator.java` the Application object registers `Calculator.java` as a Web service with the following method invocation:

```
WOWebServiceRegistrar.registerWebService(Calculator.class, true);
```

The WOWebServiceRegistrar class (`com.webobjects.webservices.appserver`) provides methods to register and unregister classes as Web services, set a security delegate, register XSLT (Extensible Stylesheet Language Transformations) scripts for operations, and so on.

To become a Web service provider, build and run the Calculator application. To view the WSDL document for the Web service, point your Web browser to `http://localhost:4210/WebObjects/Calculator.woa/ws/Calculator?wsdl`.

# Consuming a Web Service

The companion project Calculator_Client, located in `project/Calculator_Client`, contains the source files used to create the Calculator_Client application, which consumes the Calculator Web service described in "Providing a Web Service" (page 27). Its main class is `CalculatorClient.java`, shown in Listing 3-2.

**Listing 3-2** `CalculatorClient.java` class in Calculator_Client project

```
import java.net.*;
import java.util.Enumeration;

import com.webobjects.foundation.*;
import com.webobjects.webservices.client.*;

public class CalculatorClient extends Object {

    /**
     * Object through which the Web service's operations are invoked.
     */
    private WOWebServiceClient _serviceClient = null;

    /**
     * Address for the Web service's WSDL document.
     */
    private String _service_address =
        "http://localhost:4210/cgi-bin/WebObjects/Calculator.woa/ws/Calculator?wsdl";

    /**
     */
    public CalculatorClient() {
        super();
    }

    /**
     * Obtains the Web service's operation names.
     * @return the Web service's operation names.
```

```
     */
    public NSArray operations() {
        NSArray operations =
(serviceClient().operationsDictionaryForService(serviceName())).allValues();
        NSMutableArray operation_names = new NSMutableArray();
        Enumeration operations_enumerator = operations.objectEnumerator();
        while (operations_enumerator.hasMoreElements()) {
            WOClientOperation operation =
(WOClientOperation)operations_enumerator.nextElement();
            operation_names.addObject((String)operation.name());
        }
        return operation_names;
    }

    /**
     * Invokes the Web service's operations.
     * @param operation      operation to invoke;
     * @param arguments      argument list;
     * @return               value returned by the operation.
     */
    public Double invoke(String operation, Object[] arguments) {                    // 1
        Object result = serviceClient().invoke(serviceName(), operation, arguments);
        return (Double)result;
    }

    /**
     * Obtains the Web service name.
     * Normally one WSDL file describes one Web service,
     * but it could describe one or more services.
     * @return Web service name.
     */
    public String serviceName() {                                                   // 2
        return (String)serviceClient().serviceNames().objectAtIndex(0);
    }

    /**
     * Obtains an WOWebServiceClient through which service operations are invoked.
     * @return Web service-client object.
     */
    private WOWebServiceClient serviceClient() {
        if (_serviceClient == null) {
            _serviceClient = clientFromAddress(_service_address);
        }
        return _serviceClient;
    }

    /**
     * Obtains a Web service-client object through which
     * service operations can be invoked.
     * @return Web service-client object.
     */
    private static WOWebServiceClient clientFromAddress(String address) {
        WOWebServiceClient service_client = null;

        // Create the Web service's URL.
        URL url;
        try {
            url = new URL(address);
```

```
        }
        catch (MalformedURLException e) {
            url = null;
        }

        // Get a service-client object.
        service_client = new WOWebServiceClient(url);                                    // 3

        return service_client;
    }
}
```

The following list highlights some aspects of `CalculatorClient.java`.

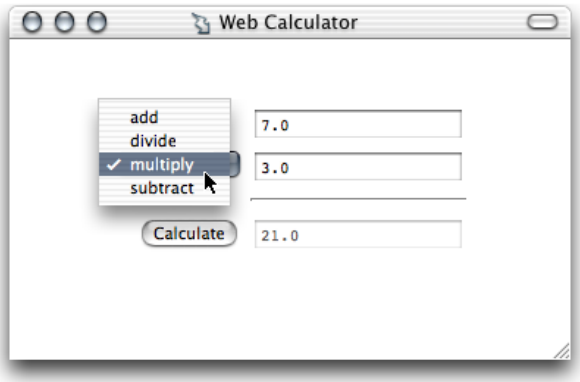
1. The `invoke` method defines as parameters the operation name and its arguments. It uses the `invoke` method of WOWebServiceClient to invoke the Web service operation.

2. The `serviceName` method returns the name of the first Web service in the list of Web services defined by the WSDL document used to create the WOWebServiceClient object. Most WSDL documents define one Web service, but a WSDL document can define more than one Web service.

3. The WOWebServiceClient class (`com.webobjects.webservices.client`) provides a one-argument constructor that takes a URL (`java.net`) object that points to a WSDL document. Therefore, to create a WOWebServiceClient you must create a URL object from the URL (Uniform Resource Locator) of the appropriate WSDL document.

Build and run the Calculator_Client application. Your Web browser should show a page like the one shown in Figure 3-1. If your Web browser didn't launch, launch it and connect to `http://localhost:4210/cgi-bin/WebObjects/Calculator_Client.woa`.

**Figure 3-1** A possible user interface to the Calculator Web service




# Using Sessions in Web Services

Using sessions during Web service consumption is simple. You get a session from one Web service and share it with other Web services served from the same application. For example, you can develop an application that provides several related Web services. A practical way to share information among the services is to store shared data in a session object.

The Session project, in `projects/Session`, showcases a simple Web service application that provides two Web services: LogIn and AccessData. The LogIn service accepts user data and stores in a session. To give the AccessData service access to the information recorded by LogIn, its session is set to the one used by LogIn.

The Session_Client project, in `projects/Session_Client`, implements a Web service client that consumes LogIn and AccessData. It sets user name and password properties through LogIn and retrieves them through AccessData. Listing 3-3 shows the logic behind this process.

**Listing 3-3** Session_Client project—`Application.java` file

```
import com.webobjects.appserver.*;
import com.webobjects.foundation.*;
import com.webobjects.webservices.client.*;

public class Application extends WOApplication {

    public static void main(String argv[]) {
        WOApplication.main(argv, Application.class);
    }

    public Application() {
        super();
        System.out.println("Welcome to " + this.name() + "!");

        // Create the service client used to consume
        // both LogInService and AccessDataService.
        SecurityClient securityClient = new SecurityClient();

        // Log in as Susana with the password anasus.
        securityClient.logIn("Susana", "anasus");

        // Get session from LogInService.
        WOWebService.SessionInfo sessionInfo = securityClient.logInSessionInfo();

        // Set AccessDataService's session to the one obtained from LogInService.
        securityClient.setAccessDataSessionInfo(sessionInfo);

        // Get values of properties stored in session created by LogInService.
        String userName = securityClient.userName();
        String userPassword = securityClient.userPassword();

        // Print the properties' values.
        System.out.println();
        System.out.println("***********************************************");
        System.out.println("User name from AccessDataService: " + userName);
        System.out.println("User password from AccessDataService: "  + userPassword);
        System.out.println("***********************************************");
        System.out.println();
    }
}
```

Listing 3-4 shows the SessionClient class in the Session_Client project.

**Listing 3-4** Session_Client project—`SessionClient.java` file

```
import java.net.*;
```

```
import com.webobjects.appserver.*;
import com.webobjects.foundation.*;
import com.webobjects.webservices.client.*;

/**
 * Used to consume the LogIn and AccessData Web services.
 */
public class SecurityClient extends Object {

    private WOWebServiceClient _logInClient = null;
    private WOWebServiceClient _accessDataClient = null;

    private final String LogInServiceAddress =
"http://localhost:4220/cgi-bin/WebObjects/Security.woa/ws/LogIn?wsdl";
    private final String AccessDataServiceAddress =
"http://localhost:4220/cgi-bin/WebObjects/Security.woa/ws/AccessData?wsdl";

    private final String LogInService = "LogIn";
    private final String AccessDataService = "AccessData";


    public SecurityClient() {
        super();
    }

    /**
     * Invokes the setUserInfo operation of the LogIn Web service.
     */
    public void logIn(String name, String password) {
        Object[] arguments = { name, password };
        logInClient().invoke(LogInService, "setUserInfo", arguments);
    }

    /**
     * Invokes the userName operation of the AccessData Web service.
     * @return user name stored in shared session object.
     */
    public String userName() {
       Object result = accessDataClient().invoke(AccessDataService, "userName", null);
        return (String)result;
    }

    /**
     * Invokes the userPassword operation of the AccessData Web service.
     * @return user password stored in shared session object.
     */
    public String userPassword() {
        Object result = accessDataClient().invoke(AccessDataService, "userPassword",
null);
        return (String)result;
    }

    /**
     * Obtains a Web service client through which LogIn operations are invoked.
     * @return a Web service client for LogIn.
     */
    protected WOWebServiceClient logInClient() {
        if (_logInClient == null) {
```

```
            _logInClient = clientFromAddress(LogInServiceAddress);
        }
        return _logInClient;
    }

    /**
     * Obtains a Web service client through which AccessData operations are invoked.
     * @return a Web service client for AccessData.
     */
    protected WOWebServiceClient accessDataClient() {
        if (_accessDataClient == null) {
            _accessDataClient = clientFromAddress(AccessDataServiceAddress);
        }
        return _accessDataClient;
    }

    /**
     * Obtains session information from LogInService.
     * @return session information from LogInService.
     */
    public WOWebService.SessionInfo logInSessionInfo() {
        return logInClient().sessionInfoForServiceNamed(LogInService);
    }

    /**
     * Sets the session used by AccessDataService.
     */
    public void setAccessDataSessionInfo(WOWebService.SessionInfo sessionInfo) {
       accessDataClient().setSessionInfoForServiceNamed(sessionInfo, AccessDataService);
    }

    /**
     * Obtains a Web service client through which
     * service operations are invoked.
     * @return Web service client object.
     */
    private WOWebServiceClient clientFromAddress(String address) {
        WOWebServiceClient service_client = null;

        // Create the Web service's URL.
        URL url;
        try {
            url = new URL(address);
        }
        catch (MalformedURLException e) {
            url = null;
        }

        // Get a service-client object.
        service_client = new WOWebServiceClient(url);

        return service_client;
    }
}
```

Listing 3-5 shows the LogIn class in the Session project.

**Listing 3-5** Session project—LogIn.java file

```
import org.apache.axis.MessageContext;

import com.webobjects.appserver.*;
import com.webobjects.foundation.*;

/**
 * Implements the LogIn Web service.
 */
public class LogIn {
    public static final String USER_NAME = "userName";
    public static final String USER_PASSWORD = "userPassword";

    /**
     * Sets a user's name and password properties in a session.
     */
    public void setUserInfo(String userName, String userPassword) {
        WOSession session = serviceSession();
        session.setObjectForKey(userName, USER_NAME);
        session.setObjectForKey(userPassword, USER_PASSWORD);
    }

    /**
     * Retrieves the session from the current context.
     * @return current context's session.
     */
    private WOSession serviceSession() {
        WOContext context =
(WOContext)MessageContext.getCurrentContext().getProperty("com.webobjects.appserver.WOContext");
        WOSession session = context.session();
        return session;
    }
}
```

Listing 3-6 shows the AccessData class in the Session project.

**Listing 3-6** Session project—`AccessData.java` file

```
import org.apache.axis.MessageContext;

import com.webobjects.appserver.*;
import com.webobjects.foundation.*;

/**
 * Implements the AccessData Web service.
 */
public class AccessData {

    /**
     * Obtains the value of the USER_NAME property from the session.
     * @return user name.
     */
    public String userName() {
        String userName = null;
        WOSession session = serviceSession();
        if (session != null) {
            userName = (String)session.objectForKey(LogIn.USER_NAME);
```

```
        }
        return userName;
    }

    /**
     * Obtains the value of the USER_PASSWORD property from the session.
     * @return user password.
     */
    public String userPassword() {
        String userPassword = null;
        WOSession session = serviceSession();
        if (session != null) {
            userPassword = (String)session.objectForKey(LogIn.USER_PASSWORD);
        }
        return userPassword;
    }

    /**
     * Retrieves the session from the current context.
     * @return current context's session.
     */
    private WOSession serviceSession() {
        WOContext context = (WOContext)MessageContext.getCurrentContext().getProperty(
                                        "com.webobjects.appserver.WOContext");
        WOSession session = context.session();
        return session;
    }
}
```

# Accessing the WOContext Object

Sometimes you may need to access the context (`com.webobjects.appserver.WOContext` ) of an HTTP request. For example, you can use the WOContext object to store data you want to share between methods or classes as an operation is processed. To access the WOContext object associated with an operation's invocation, use the code in Listing 3-7 .

**Listing 3-7** Accessing the WOContext object from a Web service class

```
import com.webobjects.appserver.WOContext;
import org.apache.axis.MessageContext;
...
MessageContext message_context = MessageContext.getCurrentContext();
WOContext context =
(WOContext)message_context.getProperty("com.webobjects.appserver.WOContext");
```

# Adding Security to Web Services

You can add security processing to Web services by creating a security delegate class that implements methods of the WOSecurityDelegate  interface in the `com.webobjects.webservices.support` package. These methods are

- `processClientRequest` : Invoked by consumer applications before sending a request to the provider.
- `processClientResponse` : Invoked by consumer applications before processing a response from the provider.
- `processServerRequest` : Invoked by provider applications before processing a request from a consumer.
- `processServerResponse` : Invoked by provider applications before sending a response to a consumer.
- `onFaultClientRequest` : Invoked by consumer applications when a processing exception occurs while generating a request.
- `onFaultClientResponse` : Invoked by consumer applications when a processing exception occurs while processing a server response.
- `onFaultServerRequest` : Invoked by provider applications when a processing exception occurs while processing a consumer request.
- `onFaultServerResponse` : Invoked by provider applications when a processing exception occurs while generating a response.

The projects Security and Security_Client in `projects/Security` and `projects/Security_Client`, respectively, show how the methods of a security-delegate class are invoked before and after an operation is executed.

# Web Service Deployment Descriptors

Under Axis, Web services are deployed using XML-based files known as Web service deployment descriptors (WSDD). The Resources group of Web service application projects contains one or two of these files, named `client.wsdd` and `server.wsdd`. Application projects that only provide Web services have the `server.wsdd` file, while projects that consume services contain both.

Listing 3-8 shows the `server.wsdd` file of a Web service provider project.

**Listing 3-8** The `server.wsdd` file of a Web service provider project

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <globalConfiguration>
        <parameter name="sendMultiRefs" value="true"/>
        <parameter name="sendXsiTypes" value="true"/>
        <parameter name="sendXMLDeclaration" value="true"/>
        <requestFlow>
            <handler
type="java:com.webobjects.webservices.support._private.WOSecurityHandler"/>
            <handler
type="java:com.webobjects.appserver._private.WOServerSessionHandler"/>
        </requestFlow>
        <responseFlow>
            <handler
type="java:com.webobjects.appserver._private.WOServerSessionHandler"/>
            <handler
type="java:com.webobjects.webservices.support._private.WOSecurityHandler"/>
```

```
        </responseFlow>
    </globalConfiguration>
    <handler name="URLMapper" type="java:org.apache.axis.handlers.http.URLMapper"/>
    <handler name="HTTPActionHandler"
type="java:org.apache.axis.handlers.http.HTTPActionHandler"/>
   <handler name="RPCDispatcher" type="java:org.apache.axis.providers.java.RPCProvider"/>
   <handler name="MsgDispatcher" type="java:org.apache.axis.providers.java.MsgProvider"/>
    <transport name="http">
        <requestFlow>
            <handler type="HTTPActionHandler"/>
            <handler type="URLMapper"/>
        </requestFlow>
    </transport>
</deployment>
```

Listing 3-9 shows the client.wsdd file of a Web service consumer project.

**Listing 3-9** The `client.wsdd` file of a Web service consumer project

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <globalConfiguration>
        <parameter name="sendMultiRefs" value="true"/>
        <parameter name="sendXsiTypes" value="true"/>
        <parameter name="sendXMLDeclaration" value="true"/>
        <requestFlow>
            <handler
type="java:com.webobjects.webservices.support._private.WOSecurityHandler"/>
            <handler
type="java:com.webobjects.webservices.client._private.WOClientSessionHandler"/>
        </requestFlow>
        <responseFlow>
            <handler
type="java:com.webobjects.webservices.client._private.WOClientSessionHandler"/>
            <handler
type="java:com.webobjects.webservices.support._private.WOSecurityHandler"/>
        </responseFlow>
    </globalConfiguration>
    <transport name="http" pivot="java:org.apache.axis.transport.http.HTTPSender"/>
    <transport name="https" pivot="java:org.apache.axis.transport.http.HTTPSender"/>
    <transport name="local" pivot="java:org.apache.axis.transport.local.LocalSender"/>
</deployment>
```

You can edit the `server.wsdd` and the `client.wsdd` files to add handlers or to add Web services that have static WSDL documents. However, you must not remove any of the handlers defined in those files by default. Also, you should consult the Axis documentation before making any changes to the WSDD files.

# Adding Web Service Support to Existing Projects

To add Web service–provider support to an existing project, you have to add the JavaWebServiceSupport framework to it. To add Web service–client support, you need to add the JavaWebServiceSupport and JavaWebServiceClient frameworks. The frameworks are located in `/System/Library/Frameworks` (`$NEXT_ROOT/Library/Frameworks` on Windows).

# Developing Direct to Web Services Applications

This chapter describes the creation of a Direct to Web Services application. Direct to Web Services allows you to rapidly develop Web service–based applications that provide access to a data store. As other WebObjects rapid-development approaches, Direct to Web Services is a data model–based and rule-based application development approach.

You create a project called `HousesForSale`, which provides a Web service with two operations, one to find information on houses for sale and another to find real-estate agents. If you don't want to create the project by hand, you can find it in `projects/HousesForSale`.

The chapter contains the following sections:



## The Data Model

The HousesForSale project includes the JavaRealEstate framework located in `/Library/Frameworks`. The framework contains the RealEstate data-model file. The data model defines several entities; you work with only two of them: Listing and ListingAddress. Figure 4-1 shows the Listing entity definition and data from its corresponding database table; Figure 4-2 does the same for the ListingAddress entity.

**Figure 4-1** Listing entity defined in the RealEstate data model

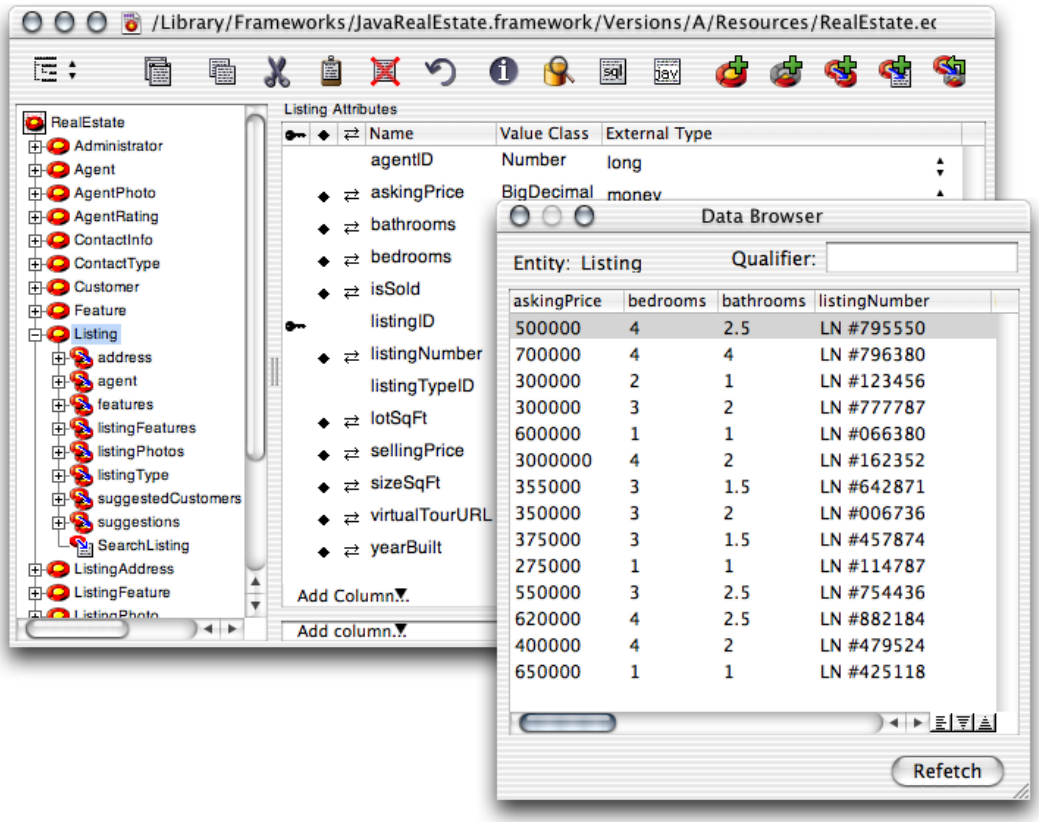
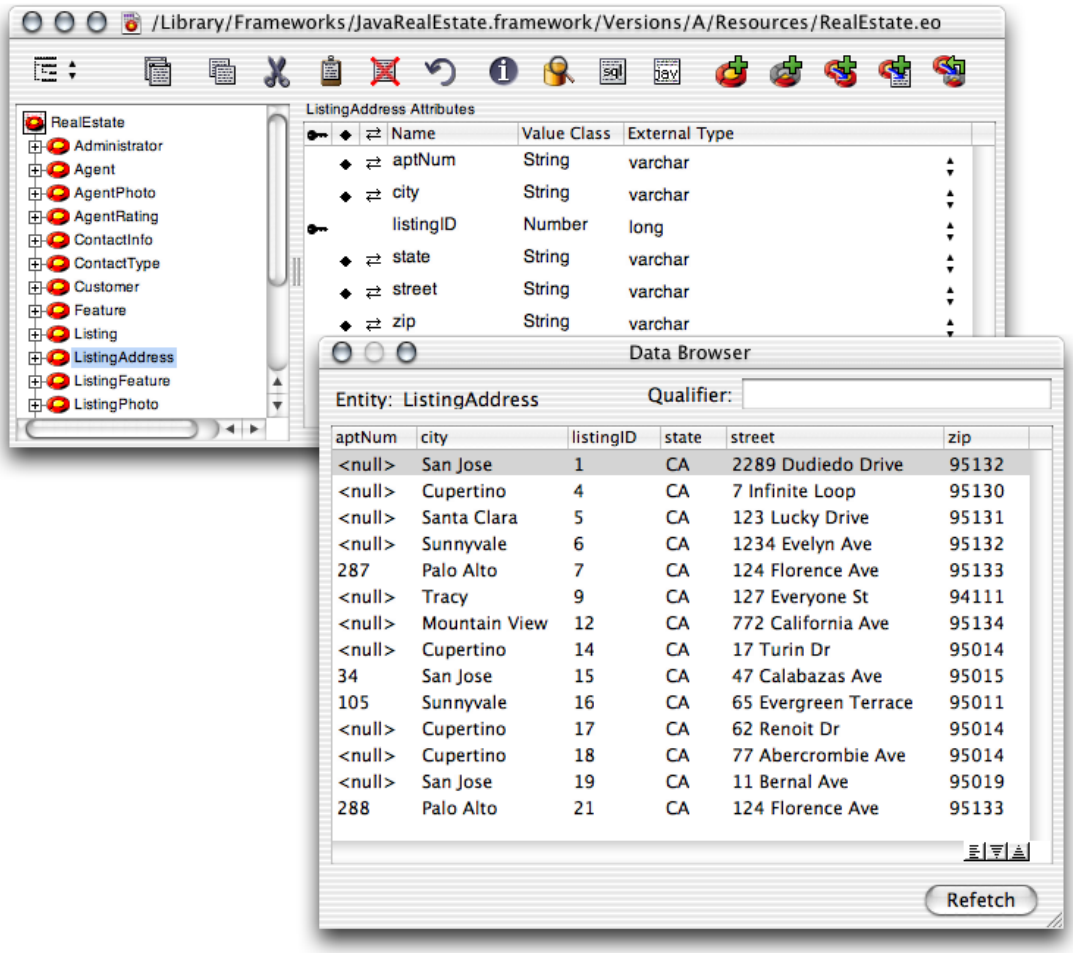


**Figure 4-2** ListingAddress entity defined in the RealEstate data model




# Creating a Direct to Web Services Application Project

Follow these steps to create a Direct to Web Services–based application project:

1. Launch Project Builder, located in `/Developer/Applications`.
2. In the New Project pane of the Project Builder Assistant, select Direct to Web Services Application under WebObjects.

To create the HousesForSale project:

1. Name the project `HousesForSale`.
2. In the Choose EOAdaptors pane, make sure the JDBC adaptor is selected.
3. In the Choose Frameworks pane, add the JavaRealEstate framework located in `/Library/Frameworks`.

4. In the Build and Launch Project pane, deselect "Build and launch project now."
5. Edit the Properties file so that it looks like Listing 4-1 .

**Listing 4-1** Properties file of the HousesForSale project

```
WOAutoOpenInBrowser false
WOPort 5210
```

6. Build and run the application.

# Web Services Assistant

To customize a Direct to Web Services application you use the Web Services Assistant. It's located in `/Developer/Applications` . With it you define an operation's parameters and return values. In addition, you determine whether the operation's result is returned as an array of enterprise-object instances or as a SOAP document, which can be traversed using an NSDictionary.

After you launch the Assistant, the Connect dialog appears (Figure 4-3 ). Enter `http://localhost:<port>` in the text input field and click Connect.

To connect to the HousesForSale application, enter `http://localhost:5210` .

**Figure 4-3** Connect dialog of Web Services Assistant

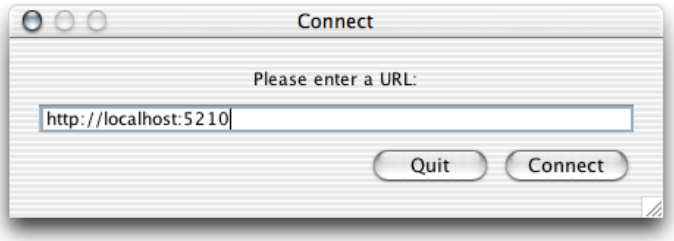
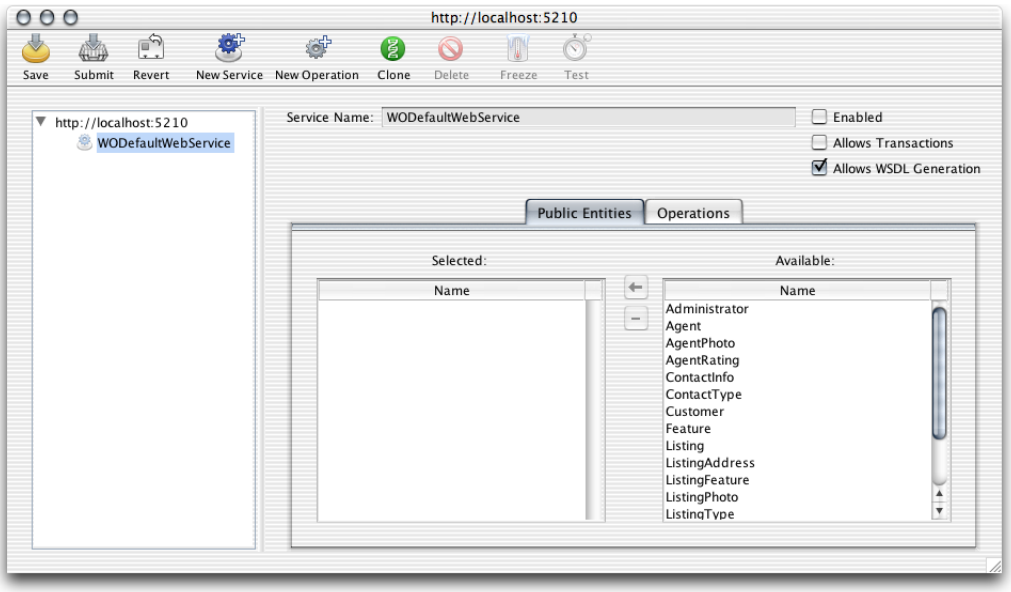



Figure 4-4 shows the Web Services Assistant main window.

**Figure 4-4** The Web Services Assistant main window




Initially, your application contains one Web service named WODefaultWebService, which is disabled by default. You should enable this service during development only. When the service is enabled and you add an entity to the service's Public Entity list, the Assistant creates `insert`, `update`, `search`, and `delete` operations for it, in addition to fetch-specification based operations. You can then copy those operations to a custom service, intended for public consumption. See "Using WODefaultWebService Operations" (page 47) for details.

# Adding a Web Service

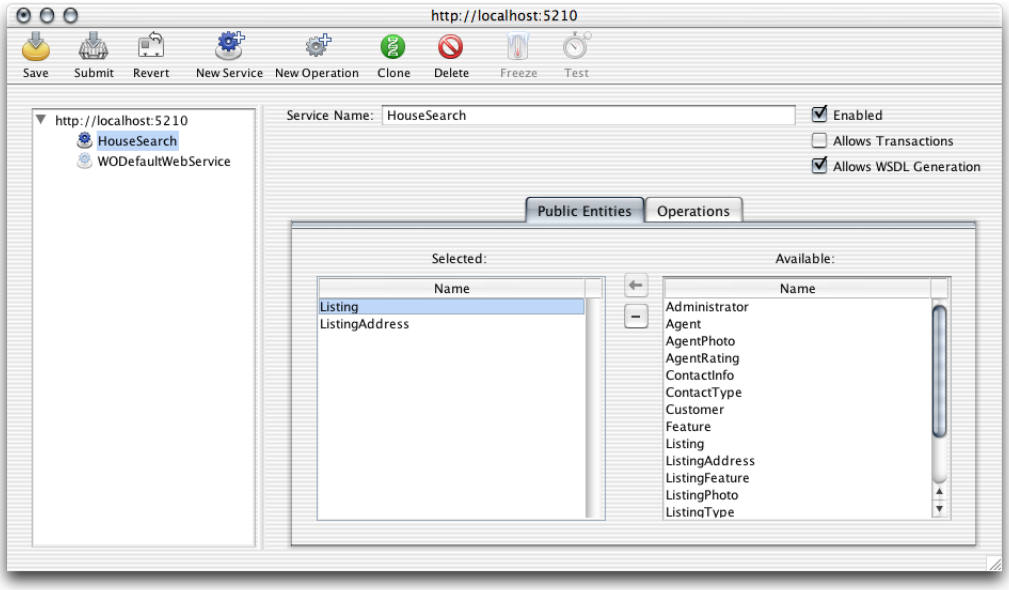
These are the steps you need to perform to add a Web service to a Direct to Web Services application:

1. In the Web Services Assistant main window, select the server application in the left-hand side list.
2. Click the New Service toolbar button.
3. Enter the name of the service in the Service Name text input field.
4. Select the entities you want to use in the Web service.

To add the HouseSearch Web service to the HousesForSale application:

1. Select `http://localhost:5210` in the left-hand side list.
2. Click New Service.
3. Enter HouseSearch in the Service Name text field.
4. Select Listing and ListingAddress in the Available list of the Public Entities pane and click the button with the left-pointing arrow.

5. Make sure Enabled is selected.




# Adding an Operation

These are the steps you need to perform to add an operation to a Web service:

1. In the Web Services Assistant, click the New Operation toolbar button.
2. Set the name and type of the operation in the New Operation dialog:
    a. Enter the name of the operation in the Name text input field.
    b. From the Entity pop-up menu, choose the entity the operation is to act on.
    c. From the Type pop-up menu, choose the type of the operation: search, insert, update, or delete.
    d. Click OK.
3. Define the operation's arguments and return values in the Arguments and Return Values panes, respectively.

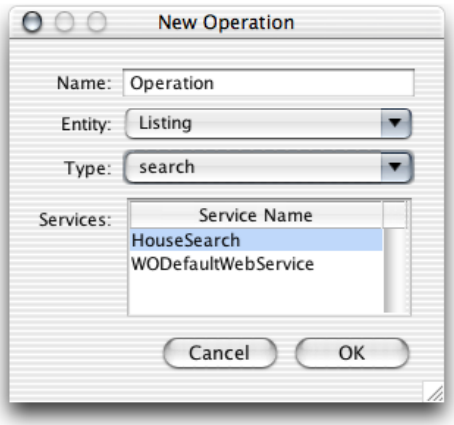
To add the `findHouseByAskingPrice` operation to the HouseSearch Web service of the HousesForSale application, follow these steps:

1. Click New Operation.
2. Enter `findHouseByAskingPrice` in the Name text field.
3. Choose Listing from the Entity pop-up menu.

4. Make sure Type is "search."

5. Make sure HouseSearch is selected in the Services list and click OK.

**Figure 4-5** The New Operation dialog

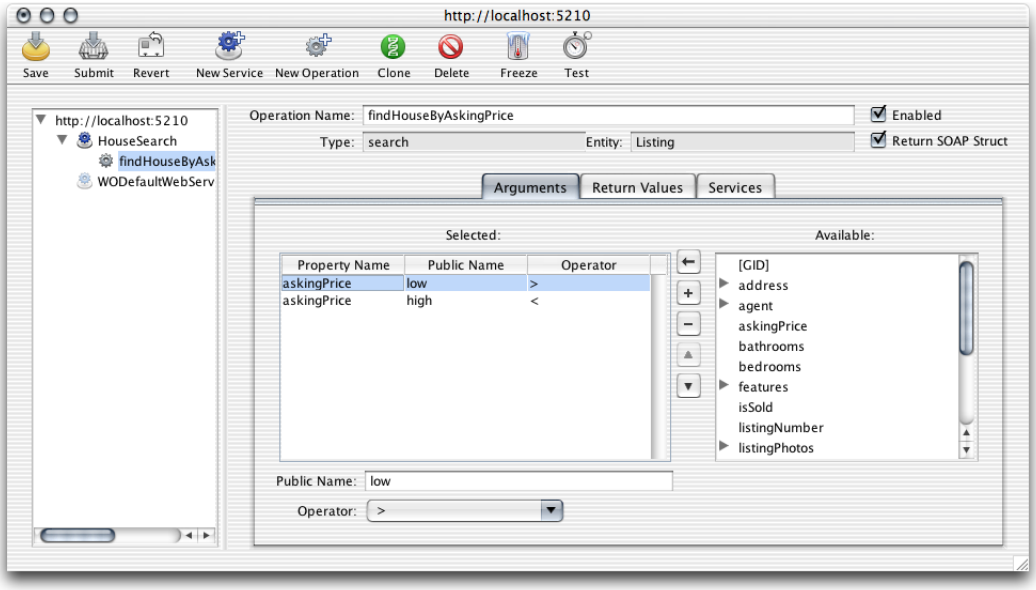



6. In the main window (Figure 4-6 ), select `askingPrice` in the Available list in the Arguments pane and click the button with the left-pointing arrow twice.

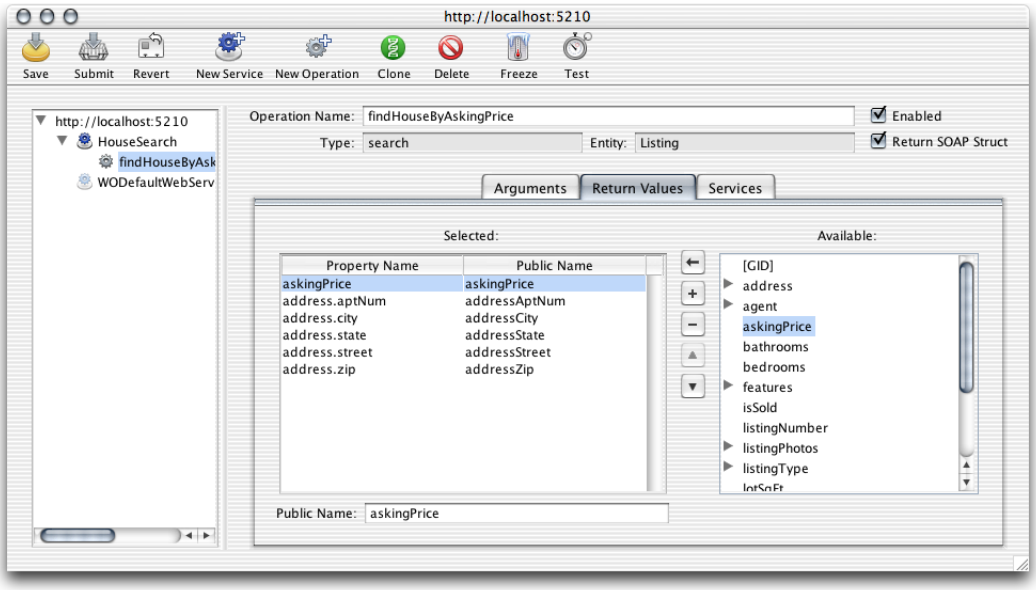
7. Select the first row of the Selected list, enter `low` in the Public Name text input field, and choose ">" from the Operator pop-up menu.

8. Select the second row, enter `high` in the Public Name text field, and choose "<" from the Operator pop-up menu.

**Figure 4-6** The findHouseByAskingPrice operation of the HouseSearch Web service



9. In the Return Values pane, select `askingPrice` from the Available list and click the button with the left-pointing arrow. Repeat for `address.appNum`, `address.street`, `address.city`, `address.state`, and `address.zip`.




Notice that the Selected list of the Return Values pane contains two columns: Property Name and Public Name. The Property Name column shows the fully qualified name of the property as shown in the Available list—including key paths—such as `address.city`. The Public Name column shows the name to be used for operation parameters and return values. You can use different names for those properties, especially since periods cannot be used within the names of operation parameters and return values. The Web Services Assistant removes periods from key paths and capitalizes the first letter of each node of the key path except the first node. So, `address.city` becomes `addressCity`. See "Testing an Operation" (page 46) for an example response to an invocation of the `findHouseByAskingPrice` operation.

# Testing an Operation

To test an existing operation in the Web Services Assistant, perform these steps:

1. Select the operation you want to test from the left-hand side list.

   For testing purposes, it's appropriate to select Return SOAP Struct. That setting is also appropriate when the applications that consume the operation are not WebObjects applications.

2. Click the Test toolbar button.

   The Testing pane of the test window, shown in Figure 4-7 (page 47), has two panes: the Parameters pane and the Result pane. In the Parameters pane you enter the values for the operation's parameters. When you click Test, the Result pane shows the return values of the operation.

3. In the Parameters pane of the Test window, enter values for the operation's parameters and click Test.

To test the `findHouseByAskingPrice` operation of the HouseSearch service of the HousesForSale example, follow these steps:

1. Select `findHouseByAskingPrice` under HouseSearch under `http://localhost:5210`.
2. Select Return SOAP Struct.
3. Click the Test toolbar button.
4. Enter `250000` in the Low text input field, `350000` in the High text field, and click Test.

**Figure 4-7** The test window of the `findHouseByAskingPrice` operation


# Using WODefaultWebService Operations

Although creating operations is made easy by the Web Services Assistant, you may want to get a head start. When you add entities to the WODefaultWebService service, the Assistant adds four operations: `insert`, `update`, `search`, and `delete`. By default only the `search` operation is enabled. In addition, operations are created for all fetch specifications defined in the data model for the entity.

To use an operation from WODefaultWebService in another Web service provided by the same application, follow these steps:

1. In the Web Services Assistant, select WODefaultWebService in the left-hand side list.
2. Add the necessary entities to the Public Entities list of WODefaultWebService.

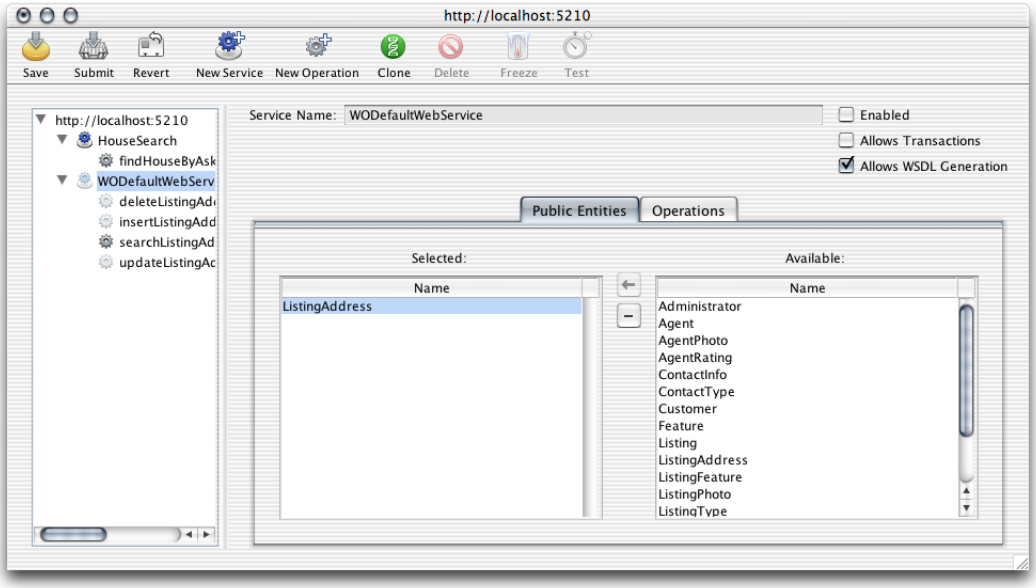
   Note that you cannot delete automatically generated operations from WODefaultWebService. But you can remove an entity from the Public Entities list, which removes all the automatically generated operations based on that entity.
3. Select the operation of WODefaultWebService you want to use in another Web service.
4. Click the Clone toolbar button.
5. In the Clone dialog, enter a name for the new operation and click Clone.
6. Select the new operation in the left-hand side list and display the Services pane.
7. Select the Web service you want to add the operation to in the Available list and click the button with the left-pointing arrow.

   You can remove the operation from WODefaultWebService by selecting WODefaultWebService in the Selected list and clicking the "–" button.

To create the `findHouseByCity` operation for the HouseSearch Web service, based on the automatically generated `searchListingAddress` operation of WODefaultWebService, follow these steps:

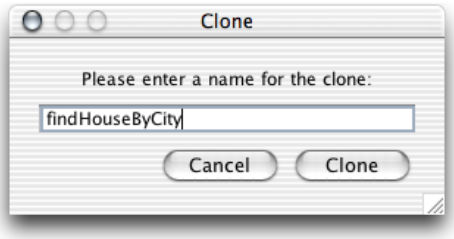
1. In the Web Services Assistant, select WODefaultWebService under `http://localhost:5210`.
2. In the Public Entities pane, select ListingAddress in the Available list and click the button with the left-pointing arrow.

   Web Services Assistant adds four operations to WODefaultWebService: `deleteListingAddress`, `insertListingAddress`, `searchListingAddress`, and `updateListingAddress`. Notice that only the `searchListingAddress` operation is enabled.



3. Select `searchListingAddress` under WODefaultWebService.

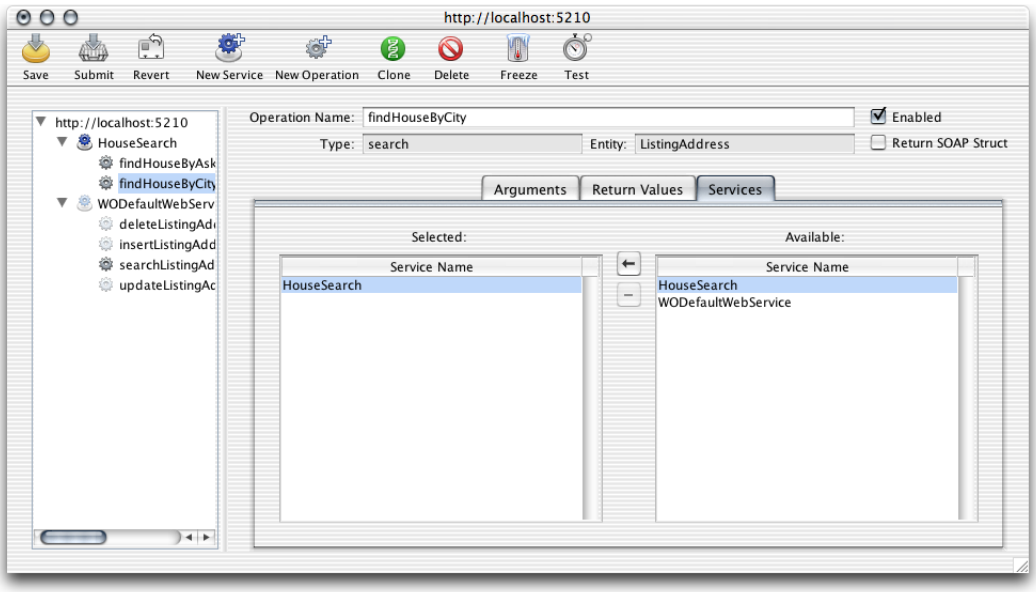
4. Click the Clone toolbar button, enter `findHouseByCity` in the text input field of the Clone dialog, and click Clone.




5. Select `findHouseByCity` under WODefaultWebService.

6. In the Services pane, select HouseSearch in the Available list and click the button with the left-pointing arrow.

7. In the Selected pane, select WODefaultWebService and click the "–" button. The operation is now part of the HouseSearch Web service.

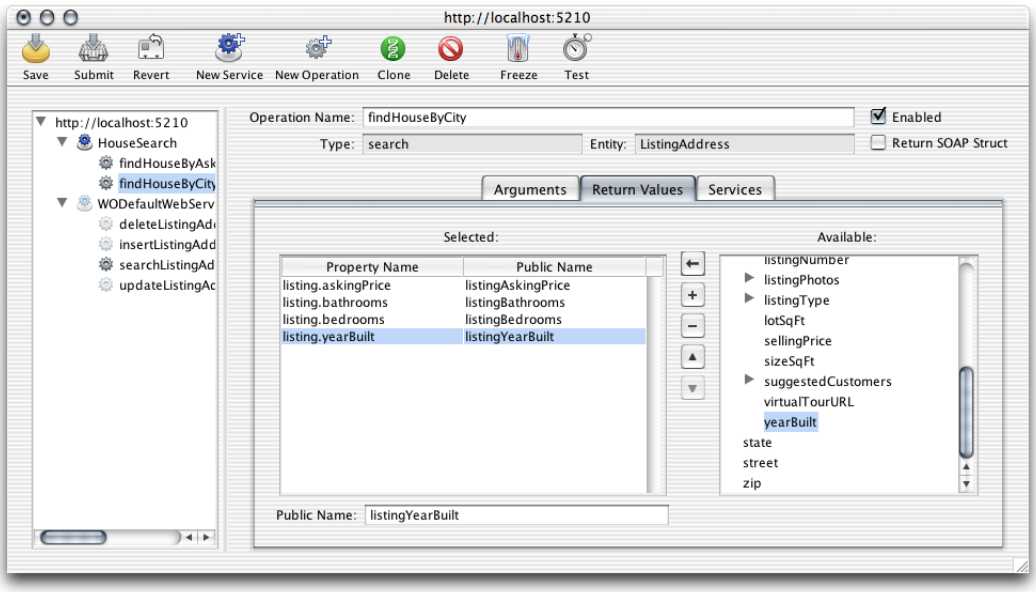



8. Select `findHouseByCity` under HouseSearch and display the Arguments pane.

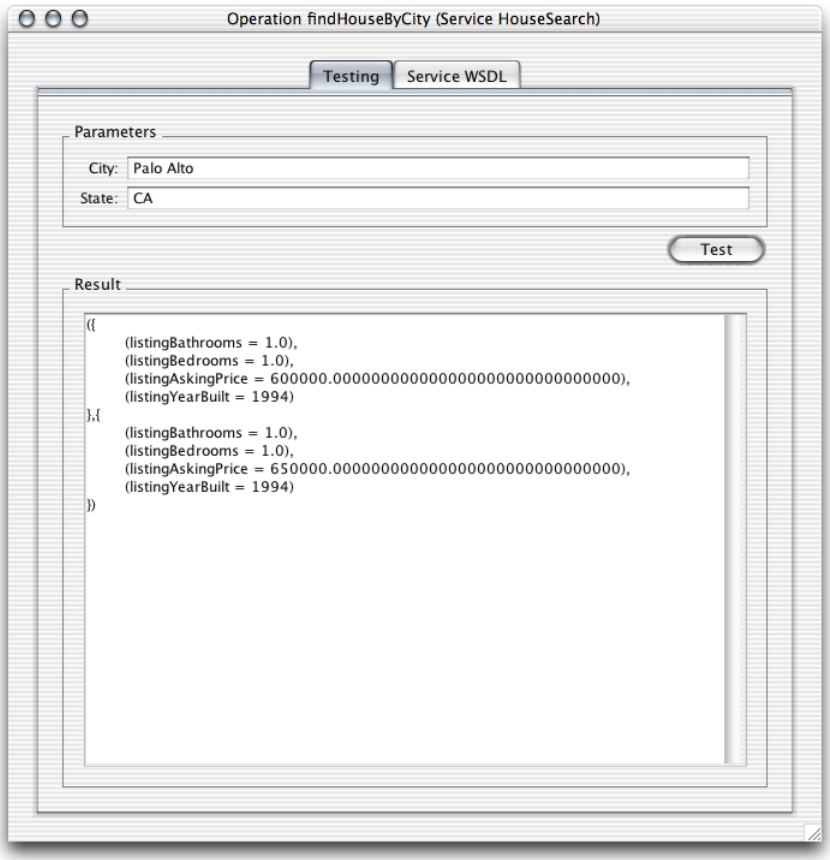
9. Select the `aptNum` property in the Selected list and click the button with the minus sign. Repeat for `street` and `zip`.

10. In the Return Values pane, remove all properties from the Selected list.

11. Select `listing.askingPrice` in the Available list and click the button with the left-pointing arrow. Repeat for `listing.bathrooms`, `listing.bedrooms`, and `listing.yearBuilt`.




12. Select Return SOAP Struct and test the operation.



# Observing SOAP Messages Using TCPMonitor

Sometimes you may find it useful to see the SOAP messages as they are transmitted between Web service consumers and providers. Follow these steps to observe the communication that occurs between the Web Services Assistant and your Direct to Web Services application:

1. Make sure the Web Services Assistant is not connected to the Direct to Web Services application whose communication you want to monitor and that the application itself is not running.
2. In Project Builder, add a custom rule file called `d2w.d2wmodel` (if it doesn't already exist), and assign it to the Application Server target. See "Creating a Custom Rule File" (page 62) for details.
3. Open `d2w.d2wmodel` in Rule Editor.

   Control-click `d2w.d2wmodel` and choose "Open with Finder."
4. In Rule Editor, add the following rule and save the `d2w.d2wmodel` file:

   Left-Hand Side: `(serviceName = '<serviceName>')`.
   Key: `serviceLocationURL`.
   Value:
   `"http://<host>:<TCPMonitorListenPort>/cgi-bin/WebObjects/<applicationName>.woa/ws/<serviceName>"`
   .
   Priority: `50`.
5. Launch TCPMonitor by double-clicking TCPMonitor in `/Developer/Examples/JavaWebObjects`, and enter the appropriate values in the Listen Port and Target Port text input fields, the port that TCPMonitor monitors and your application's port, respectively. Click Add.
6. In the Web Services Assistant, connect to your application through the port that TCPMonitor monitors.

To observe the communication that happens between the Web Services Assistant and HousesForSale, follow these steps:

1. In the Web Services Assistant, save the application's Web service configuration and close the Web Services Assistant window.
2. In Project Builder, add a custom rule file named `d2w.d2wmodel` to the application project if it doesn't already exist.
3. Open the `d2w.d2wmodel` file in Rule Editor and add the following rule:

   Left-Hand Side: `(serviceName = 'HouseSearch')`.
   Key: `serviceLocationURL`.
   Value:
   `"http://localhost:5299/cgi-bin/WebObjects/HousesForSale.woa/ws/HouseSearch"`
   .
   Priority: `50`.
4. Save the `d2w.d2wmodel` file and build and run the application.
5. Launch TCPMonitor.

6. Enter `5299` in the Listen Port text input field and `5210` in the Target Port text field and click Add.

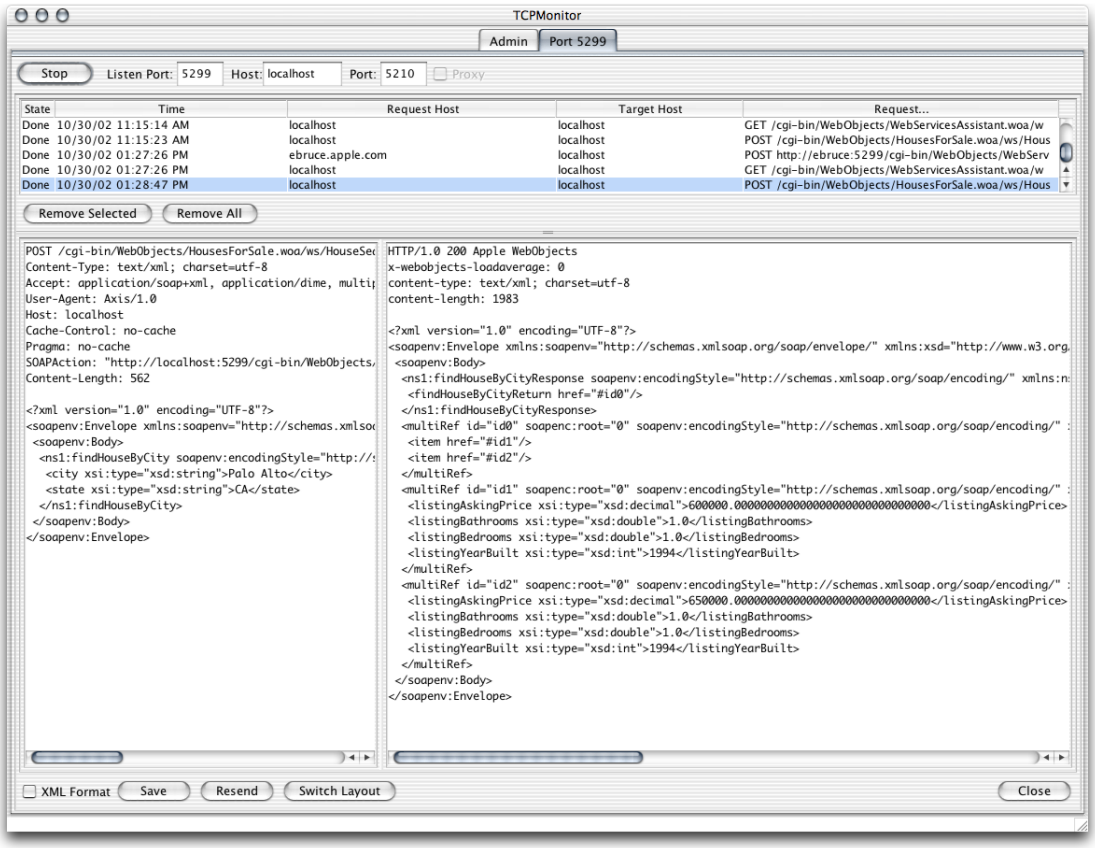
7. Display the Port `5299` pane of TCPMonitor.

8. In the Web Services Assistant, enter `http://localhost:5299` in the text input field of the Connect dialog and click Connect.

   Notice that TCPMonitor shows you the request and response documents as the Web Services Assistant communicates with the HousesForSale application.

9. If you test the `findByCity` or `findByAskingPrice` operations, TCPMonitor logs the SOAP request the Assistant sends to the application as well as the response sent by the application as shown in Figure 4-8 .

**Figure 4-8** TCPMonitor window



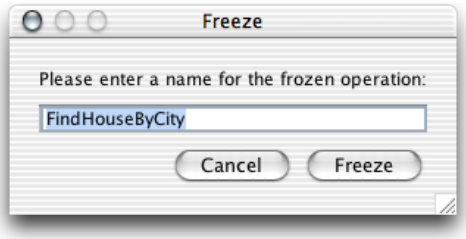

# Freezing Operations

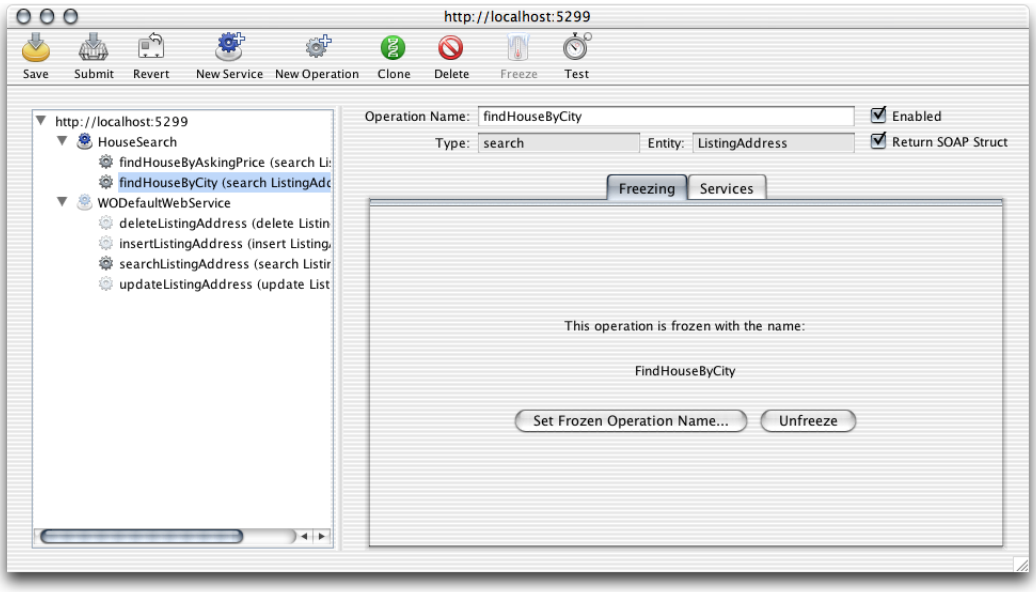
You can freeze operations when you need to customize their workings. Frozen operations take the form of Web components in your application project. When you freeze an operation, the parts of the Web service's WSDL document that correspond to the operation are frozen as well. In addition, you cannot use the Web

Services Assistant to customize further a frozen operation; for example, you cannot add or remove arguments or return values with the Assistant. If you need to do so, you have to edit the Java file and WSDL document manually.

The following list itemizes the steps needed to freeze an operation.

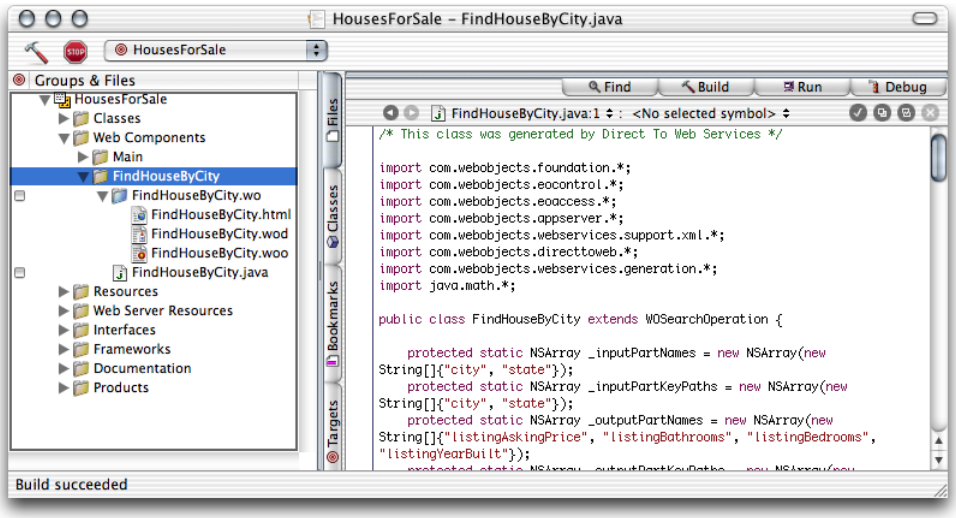
1. In the Web Services Assistant, select the operation you want to freeze and click the Freeze toolbar button. In the Freeze dialog, enter the name of the frozen-operation component and click Freeze.



The Assistant adds the FindHouseByCity component to the HouseForSale project, as shown in Figure 4-9 .

**Figure 4-9** The FindHouseByCity component—the frozen version of the `findHouseByCity` operation




2. Save the Web service configuration and close the Web Services Assistant window.
3. Restart the application.

The WSDL document corresponding to a frozen operation is stored in the HTML file of the corresponding component. Listing 4-2 shows the WSDL document for the frozen `findHouseByCity` operation.

**Listing 4-2** The WSDL document of the frozen findHouseByCity operation—the HTML file of the FindHouseByCity component

```
<?xml version="1.0"?>
<definitions name="[AnyService]Definition"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://17.203.33.19/cgi-bin/WebObjects/HousesForSale.woa/ws/
[AnyService]/wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:lang="http://lang.java/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:webobjects="http://www.apple.com/webobjects/webservices/soap/"
    targetNamespace="http://17.203.33.19/cgi-bin/WebObjects/HousesForSale.woa/ws/
[AnyService]/wsdl">
    <types>
    </types>
    <message name="findHouseByCityInput">
      <part type="xsd:string" name="city"/>
      <part type="xsd:string" name="state"/>
    </message>
    <message name="findHouseByCityOutput">
      <part type="xsd:anyType" name="return"/>
```

```
    </message>
    <message name="WSDLInput">
    </message>
    <message name="WSDLOutput">
      <part type="xsd:anyType" name="return"/>
    </message>
    <message name="beginTransactionInput">
    </message>
    <message name="beginTransactionOutput">
      <part type="xsd:anyType" name="return"/>
    </message>
    <message name="commitTransactionInput">
    </message>
    <message name="commitTransactionOutput">
      <part type="xsd:anyType" name="return"/>
    </message>
    <message name="rollbackTransactionInput">
    </message>
    <message name="rollbackTransactionOutput">
      <part type="xsd:anyType" name="return"/>
    </message>
    <portType name="[AnyService]PortType">
        <operation name="findHouseByCity" parameterOrder="city state">
            <input message="tns:findHouseByCityInput"/>
            <output message="tns:findHouseByCityOutput"/>
        </operation>
        <operation name="WSDL">
            <input message="tns:WSDLInput"/>
            <output message="tns:WSDLOutput"/>
        </operation>
        <operation name="beginTransaction">
            <input message="tns:beginTransactionInput"/>
            <output message="tns:beginTransactionOutput"/>
        </operation>
        <operation name="commitTransaction">
            <input message="tns:commitTransactionInput"/>
            <output message="tns:commitTransactionOutput"/>
        </operation>
        <operation name="rollbackTransaction">
            <input message="tns:rollbackTransactionInput"/>
            <output message="tns:rollbackTransactionOutput"/>
        </operation>
    </portType>
    <binding type="tns:[AnyService]PortType"
 name="[AnyService]SoapBinding"><soap:binding style="rpc"
 transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="findHouseByCity">
            <soap:operation soapAction="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl"/>
            <input>
                <soap:body use="encoded"
namespace="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
                <soap:body use="encoded"
namespace="http://17.203.33.19/cgi-bin/WebObjects/
```

```
HousesForSale.woa/ws/[AnyService]/wsdl" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
        <operation name="WSDL">
            <soap:operation soapAction="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl"/>
            <input>
                <soap:body use="encoded"
namespace="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
                <soap:body use="encoded"
namespace="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
        <operation name="beginTransaction">
            <soap:operation soapAction="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl"/>
            <input>
                <soap:body use="encoded"
namespace="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
                <soap:body use="encoded"
namespace="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
        <operation name="commitTransaction">
            <soap:operation soapAction="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl"/>
            <input>
                <soap:body use="encoded"
namespace="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
                <soap:body use="encoded"
namespace="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
        <operation name="rollbackTransaction">
            <soap:operation soapAction="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl"/>
            <input>
                <soap:body use="encoded"
namespace="http://17.203.33.19/cgi-bin/WebObjects/
```

```
HousesForSale.woa/ws/[AnyService]/wsdl" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
                <soap:body use="encoded"
namespace="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl" encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
    </binding>
    <service name="[AnyService]">
        <port name="[AnyService]Port" binding="tns:[AnyService]SoapBinding">
            <soap:address location="http://17.203.33.19/cgi-bin/WebObjects/
HousesForSale.woa/ws/[AnyService]/wsdl"/>
        </port>
    </service>
</definitions>
```

In addition to the `findHouseByCity` operation, the frozen WSDL document defines four additional operations: `WSDL`, `beginTransaction`, `commitTransaction`, and `rollbackTransaction`. The WSDL method can returns the WSDL document for the Web service. Consumers can use the rest of the additional operations to implement rudimentary transaction processing. For more information, see "Using Transactions" (page 59).

# Unfreezing Operations

To unfreeze a frozen operation follow these steps:

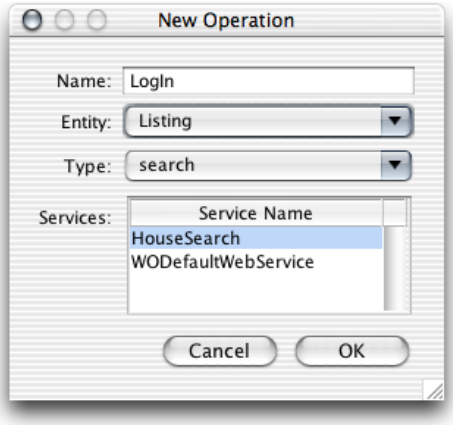
1. In the Web Services Assistant, select the operation you want to unfreeze.
2. Click Unfreeze in the Freezing pane.
3. Save the Web service configuration.
4. In Project Builder, delete the corresponding component.
    a. Select the component under the Web Components group.
    b. Choose Edit > Delete.
    c. In the Delete References dialog, click Delete References & Files.

# Component-Based Operations

"Freezing Operations" (page 52) indicated that when you freeze an operation, the operation's WSDL document as well as its parameters and return values cannot be customized using the Web Services Assistant. However, you can create custom operations whose WSDL document is dynamically generated. For that you need to copy the `templates/Direct to Web Services Operation.pbfiletemplate.pbfiletemplate` to `/Developer/ProjectBuilder Extras/File Templates/WebObjects`.

Follow these steps to create an operation with a dynamic WSDL document for the HousesForSale application:

1. Create an operation using the Web Services Assistant.




2. Save the Web service configuration and close the Web Services Assistant window.

3. In Project Builder, select the Web Components group and add a Direct to Web Services Operation component with the same name of the operation added in the Web Services Assistant.

   To confirm that the component's `invoke` method is invoked, edit the `invoke` method of its Java file so that it like this:

```
public Object invoke() {
    System.out.println("LogIn operation invoked.");
    return super.invoke();
}
```

4. Add the following rule to the `d2w.d2wmodel` file:

   Left-Hand Side: `(operationName = 'LogIn')`.
   Key: `operationClassName`.
   Value: `"LogIn"`.
   Priority: `50`.

5. Save `d2w.d2wmodel`.

6. Rebuild and run the application.

7. Connect to the application through the Web Services Assistant.

8. Test the operation. Make sure that you see the test output in the console (Project Builder's Run pane).

## Operations Derived From Fetch Specifications

When you make public an entity with fetch specifications in the Web Services Assistant, the Assistant creates operations corresponding to those fetch specifications. You can see the arguments the operations require and the return values in the Arguments pane and the Return Values panes, respectively.

To modify an operation that is based on a fetch specification from a data model, you must edit the fetch specification in the model and rebuild the application.

## Using Transactions

Direct to Web Services supports light-weight transactions. These transactions are editing context–based and cannot be nested. See the reference documentation for `com.webobjects.eocontrol.EOEditingContext` for more information on editing contexts.

When you select Allows Transactions in the Web Services Assistant for a given Web service and restart the service-provider application, three additional operations become available: `beginTransaction`, `commitTransaction`, and `rollbackTransaction`. These transactions are not visible in the Assistant, but you can view them in the Web service's WSDL document. The following list provides an overview of each operation.

- `beginTransaction`: You invoke this operation before invoking operations that modify data in a data store. After invoking the operation, calls to `EOEditingContext.saveChangesInEditingContext` have no effect. To learn more, see the reference documentation for WOBeginTransactionOperation in `com.webobjects.webservices.generation`.
- `commitTransaction`: Saves the changes made to the editing context (`com.webobjects.eocontrol.EOEditingContext`) used by the Web service. To learn more, see the reference documentation for WOCommitTransactionOperation in `com.webobjects.webservices.generation`.
- `rollbackTransaction`: Resets the editing context used by the Web service. To learn more, see the reference documentation for WORollbackTransactionOperation in `com.webobjects.webservices.generation`.

## Using Global IDs

Using the Return SOAP Struct option in the Web Services Assistant you can determine whether an operation returns a `SOAP-struct` element or one or more enterprise objects. When an operation is invoked from other WebObjects applications, you should not select Return SOAP Struct. That way the client application can access data, such as global IDs with relative ease.

Sometimes you may need to create operations that create enterprise objects that are part of a relationship. For example, Figure 4-10 shows the relationships between two data entities: the `books` relationship of Authors and the `author` relationship of Book.

**Figure 4-10** Relationship between Author entity and Book entity


Also notice that the `authorId` attribute of Book is part of the entity's primary key. This means that a Book enterprise object cannot be added to the data store without a value for the `authorId` attribute. (You cannot add a book to the data store without a corresponding author.) Creating an operation in Web Services Assistant that creates an author record and a book record is not possible. However, a method in a client of an Author Web service can invoke the `addAuthor` and `addBook` operations to perform the procedure. For that, you have to use global IDs (`com.webobjects.eocontrol.EOKeyGlobalID`). Listing 4-3 shows a possible implementation of such a method, named `addBookForAuthor` , while Listing 4-4 shows the methods that invoke the Web service operations that interact with the data store.

**Listing 4-3** `addBookForAuthor` method

```
/**
 * Adds a book and an author to the data store.
 * @param title            book's title;
 * @param authorLastName   last name of the author;
 * @param authorFirstName  first name of the author;
 * @return <code>true</code> when successful; <code>false</code> otherwise.
 */
public boolean addBookForAuthor(String title, String authorLastName,
                                String authorFirstName) {
    EOKeyGlobalID author_global_id = addAuthor(authorLastName, authorFirstName);
    addBook(title, author_global_id);
    return true;
}
```

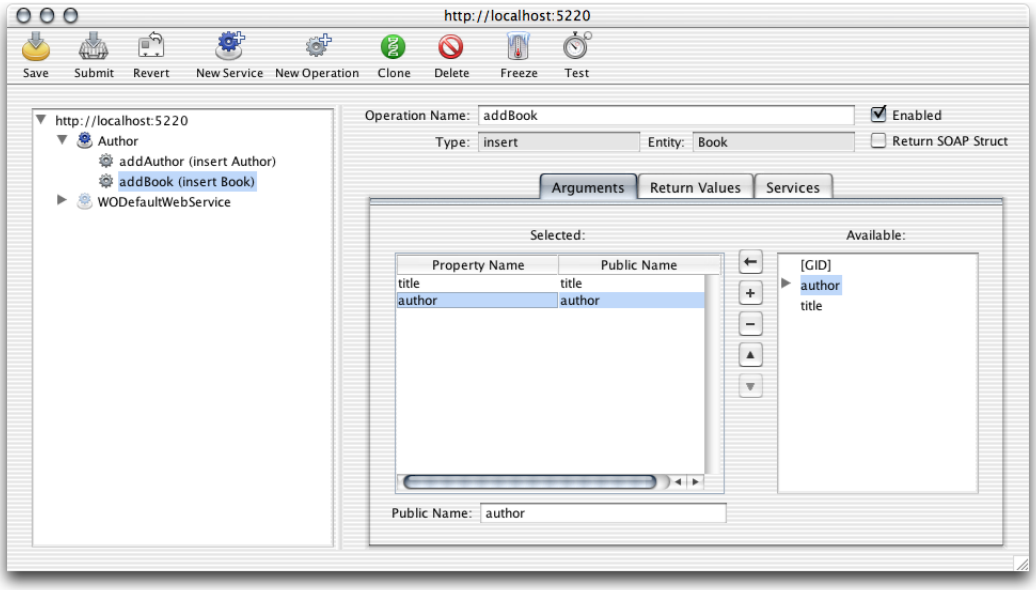
**Listing 4-4** `addAuthor` and `addBooks` methods

```
/**
 * Adds an author to the data store.
 * @param authorLastName   last name of the book's author;
 * @param authorFirstName  first name of the book's author;
 * @return global ID of the corresponding enterprise object.
 */
private EOKeyGlobalID addAuthor(String lastName, String firstName) {
    Object arguments[] = {lastName, firstName};
    Object[] result = (Object[])serviceClient().invoke(serviceName(),
                                                       "addAuthor", arguments);
    WOStringKeyMap key_map = (WOStringKeyMap)result[0];
    EOKeyGlobalID author_global_id = (EOKeyGlobalID)key_map.valueForKey("globalID");
    return author_global_id;
}
```

```
/**
 * Adds a book to the data store.
 * @param title              book's title;
 * @param author_global_id   global ID of the enterprise object representing
 *                           the book's author;
 * @return <code>true</code> when successful; <code>false</code> otherwise.
 */
private boolean addBook(String title, EOKeyGlobalID author_global_id) {
    Object arguments[] = {title, author_global_id};
    Object result = serviceClient().invoke(serviceName(), "addBook", arguments);
    return true;
}
```

Figure 4-11 shows the definition of the addBook operation. Notice the `author` argument.

**Figure 4-11** Definition of the `addBook` operation




The `projects` directory includes two projects Authors and Authors_Client that demonstrate the concepts described earlier. To run them you must install the Authors database, located in the `databases` directory.

# Default Return Values of Operations

Table 4-1 shows the return values of operations for which you don't enter return values in the Return Values pane.

**Table 4-1** Default return values of operations

| Type of operation | Return value |
|---|---|
| insert | Global ID of enterprise object created. |

| Type of operation | Return value |
|---|---|
| update | Array of updated enterprise objects. |
| search | Array of found enterprise objects. |
| delete | Empty array. |

# Creating a Custom Rule File

When you need to perform advanced customization in a Direct to Web Services application, you may have to create a custom rule file named `d2w.d2wmodel` in which you place custom rules. The other rule file, `user.d2wmodel`, is for the exclusive use of the Web Services Assistant. You must not edit `user.d2wmodel`.

To add the `d2w.d2wmodel` file to a Direct to Web Services application project, add a new, empty file to the Resources group, name the file `d2w.d2wmodel`, and assign it to the Application Server target

# Rule Editor Keys

Table 4-2 lists the keys you can use in the `d2w.d2wmodel` file of a project to customize a Direct to Web Services application. To learn more, see the reference documentation for WOServiceUtilities.RuleSystemConstants in the `com.webobjects.webservices.generation` package.

**Table 4-2** Direct to Web Services rule keys

| Key | Description |
|---|---|
| `AllOperationNames` | The names of all the operations available in a Web service. |
| `AllServiceNames` | The names of all the Web services available in an application. |
| `ClassForPropertyKey` | The class of a property. |
| `ClassNameForPropertyKey` | The class name of a property. |
| `ComparisonKey` | The EOKeyComparisonQualifier used to build an EOQualifier for a property. |
| `EntityName` | The entity name for an operation. |
| `FetchLimit` | The `fetchLimit` of a WOSearchOperation. |
| `FetchSpecificationName` | The name of the fetch specification used in a WOFetchSpecSearchOperation. |
| `GidArgumentKey` | Determines whether an enterprise object is to be serialized as an EOGlobalID. |

| Key | Description |
|---|---|
| `InputPartNames` | The XML element names of an operation's input-message parts. |
| `InputPartValues` | The arguments in an operation invocation. |
| `IsDefaultService` | Determines whether a Web service is the default Web Service in a Direct to Web Services application. |
| `IsOperationEnabled` | Determines whether an operation is enabled. |
| `IsOperationFrozen` | Determines whether an operation is frozen. |
| `IsServiceEnabled` | Determines whether a Web service is enabled. |
| `IsTransactionEnabled` | Determines whether a Web services supports transactions. |
| `ModelGroup` | The EOModelGroup of the current operation. |
| `OperationClassName` | The class name of an operation. |
| `OperationName` | The name of an operation. |
| `OperationNames` | The names of the operations of a Web service. |
| `OutputPartNames` | The XML element names of the properties an operation returns. |
| `PropertyKey` | A property key. |
| `PublicEntityNames` | The names of the public entities available in a Direct to Web Services application. |
| `ReturnSOAPStruct` | Determines whether an operation returns EOEnterpriseObjects as serialized enterprise objects or a SOAP structures. |
| `ServiceName` | The name of a Web service. |
| `TranslatedAttributeName` | The EOProperty key path for an argument or result value. |
| `UnspecifiedArgumentKey` | The key used to determine whether a parameter is unspecified. |
| `UsesNamedFetchSpecification` | Determines whether an operation uses a named fetch specification. |
| `WOContext` | The WOContext. |
| `WODefaultWebService` | The name of the default Web service. |
| `WSDLComponentName` | The name of the component containing the WSDL document for an operation. |

# Document Revision History

This table describes the changes to *WebObjects Web Services Programming Guide*.

| Date | Notes |
| --- | --- |
| 2007-07-11 | Updated for WebObjects 5.4. |
| 2005-12-06 | Removed references to projects and companion files that are no longer part of the WebObjects release. |
| 2005-08-11 | Changed the title from "Web Services." |
| 2002-11-01 | First version of *Inside WebObjects: Web Services* . |

Document Revision History

# Glossary

**consumer, Web service** An application that executes a Web service operation by sending a SOAP message to a Web service provider.

**Direct to Web Services** A WebObjects development technology that can generate a Web service application from a model.

**editing context** Object that stores and manages a group of enterprise-object instances. An editing context, which is an instance of the EOEditingContext class, provides an in-memory view of data in a data store. Changes made to enterprise-object instances in an editing context are pushed to the data store by invoking a specific method. In adition, those changes can be undone, even after they have been committed to the corresponding data store.

**handler** A Java class used by Axis to process a SOAP message or a part of it in a specific way. For example, a handler can be implemented to perform authentication on the message's sender before allowing it to be processed by the receiver.

**handler chain** A group of handlers that can be viewed as a unit.

**hash** Number derived from a string such that any change to the string produces a different number.

**operation** A specific process or task that a Web service implements. Much like Java methods, a Web service operation can define an arbitrary number of parameters and return values. Operations are invoked by Web service consumers and executed by Web service providers.

**PKI (Public Key Infrastructure)** Authorization technology that uses a combination of private key cryptography and public key cryptography. It provides key management, data integrity, and data confidentiality.

**SOAP (Simple Object Access Protocol)** XML-based, lightweight, platform-agnostic protocol used to exchange information in a decentralized, distributed environment. The protocol defines the XML elements that must be used to compose a message and how the data in a message should be processed.

**SOAP engine** Application or framework used by Web service providers and consumers to process SOAP messages.

**provider, Web service** An application that executes the logic that implements a Web service operation.

**SSL (Secure Sockets Layer)** Protocol used to provide encrypted communication on the Internet.

**UDDI (Universal Description, Discovery and Integration)** Searchable directory of Web services that Web service requestors can use to search for Web services and obtain their WSDL documents.

**Web service** A network-based repository of processes or tasks that can be used by applications to access data or execute operations across disparate platforms.

**WSDL (Web Services Description Language)** XML-based language used to describe Web services. Web service consumers can dynamically parse a WSDL document to determine the operations a Web service provides and how to execute them.

**WSS-Core (Web Services Security Core Specification)** Specification that defines a set of SOAP extensions that can be used to provide message-level data integrity and confidentiality.