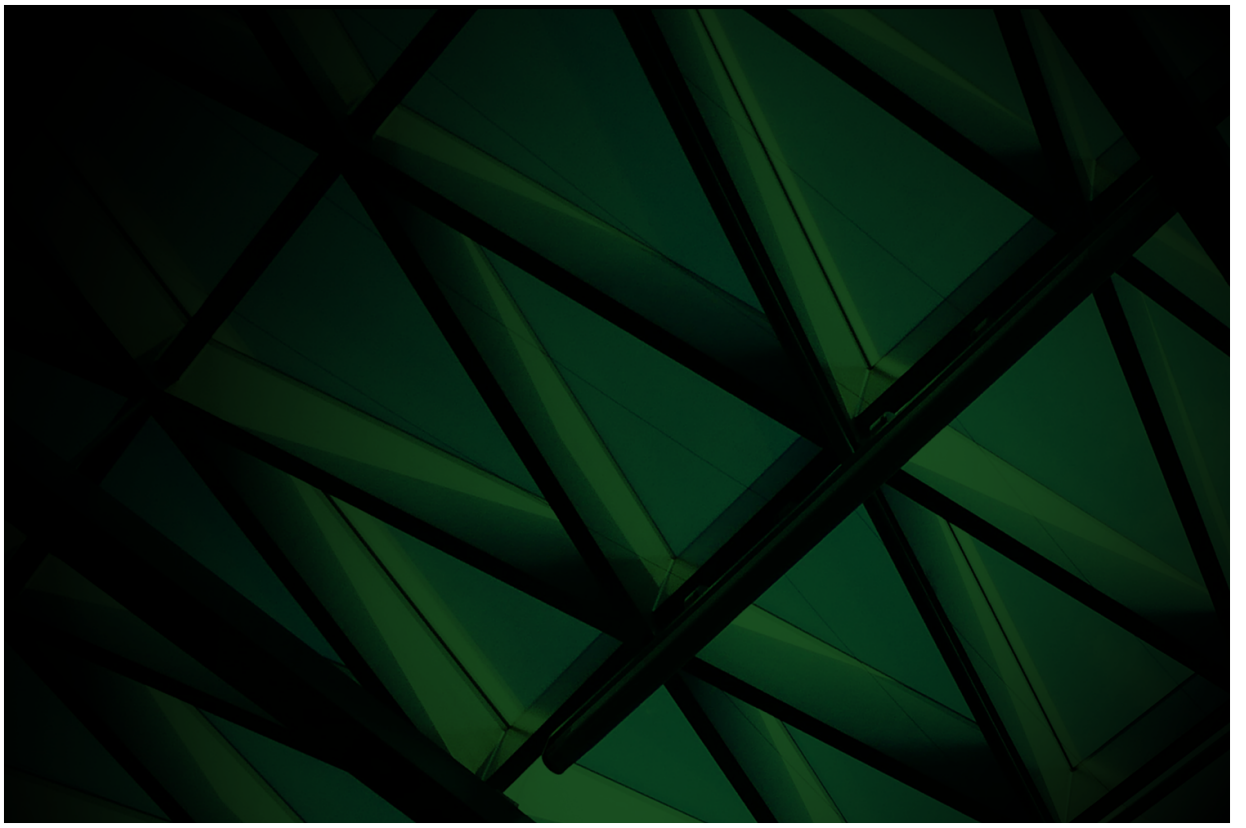


SOAP Web Services White Paper

VERSION 2022

jade platform

Contents

SOAP Web Services

Web services are the fundamental building blocks in the move to distributed computing on the Internet. Open standards and the focus on communication and collaboration among people and applications have created an environment where web services are becoming the platform for application integration.

This white paper provides an overview of Simple Object Access Protocol (SOAP) web services and some of the technologies used in web services today. This paper also covers the Jade implementation of the SOAP-based web service provider and consumer features, along with a detailed example.

For details about RESTful web services, see the *REST Services* white paper (which is also available from the Jade website at https://www.jadeplatform.com/developer-centre/learn/whitepapers).

SOAP web services generally provide the following features.

- Expose useful functionality to web users through a standards-based web protocol.
- Provide a way to describe their interfaces in enough detail to allow a user to build a client application to talk to them. This is usually provided in an Extensible Markup Language (XML) document called a Web Services Description Language (WSDL) document.
- Are registered so that potential users can find them easily. This is done with Universal Discovery Description and Integration (UDDI).

For more details, see the following subsections.

Why Web Services?

One of the primary advantages of the web services architecture is that it allows programs written in different languages on different platforms to communicate with each other in a standards-based way. SOAP is significantly less complex than earlier approaches, so the barrier to entry for a standards-compliant SOAP implementation is significantly lower. The other significant advantage that web services have over previous efforts is that they work with standard Internet protocols - XML, HTTP, and TCP/IP. A significant number of companies already have an Internet infrastructure, and people with knowledge and experience in maintaining it, so again, the cost of entry for web services is significantly less than for previous technologies.

Exposing existing applications as web services allows you to build new, more-powerful applications that use web services as building blocks. For example, you could develop a purchasing application to automatically obtain price information from a variety of vendors, allow the user to select a vendor, submit the order, and then track the shipment until it is received. The vendor application, in addition to exposing its services on the web, could in turn use web services to check the customer's credit, charge the customer's account, and set up the shipment with a shipping company.

SOAP

SOAP is a messaging protocol for web services that defines a specification for how to send messages in a standardized form. If you have a well-formed XML fragment enclosed within a pair of SOAP elements, you have a SOAP message.

There are other parts of the SOAP specification that describe how to represent program data as XML and how to use SOAP to do Remote Procedure Calls (RPCs). These optional parts of the specification are used to implement RPC-style applications where a SOAP message containing a callable function, and the parameters to pass to the function, is sent from the client and the server returns a message with the results of the executed function. Most current implementations of SOAP support RPC applications because programmers who are used to doing distributed applications with other technologies understand the RPC style.

SOAP also supports document-style applications where the SOAP message is just a wrapper around an XML document. Document-style SOAP applications are very flexible and many new web services take advantage of this flexibility to build services that would be difficult to implement using RPC. Jade gives you the freedom to choose between RPC and document styles.

The last optional part of the SOAP specification defines what an HTTP message that contains a SOAP message looks like. This HTTP binding is important because almost all current operating systems support HTTP. The HTTP binding is optional, but almost all SOAP implementations support it because it is the only standardized protocol for SOAP.

By far the most compelling feature of SOAP is that it has been implemented on many different hardware and software platforms. This means that SOAP can be used to link disparate systems.

SOAP is much smaller and simpler to implement than many of the previous protocols. For example, CORBA, one of SOAP's precursors, took years to implement so only a few implementations were ever released. SOAP, however, can use existing XML parsers and HTTP libraries to do most of the hard work, so a SOAP implementation can be completed in a matter of months.

The ubiquity of HTTP and the simplicity of SOAP make them an ideal basis for implementing web services that can be called from almost any environment.

The Jade web services framework provides support for the HTTP protocol. SOAP messaging is transparent, as Jade handles the processing of incoming SOAP messages, creates the appropriate transient objects, and calls the requested method. On return from the method call, a SOAP response message is then generated and sent back to the requesting client. User exits are available via method reimplementation to override the default processing and response.

WSDL

WSDL stands for Web Services Description Language. For our purposes, we can say that a WSDL file is an XML document that describes a set of SOAP messages and how the messages are exchanged. In other words, WSDL is to SOAP what OpenAPI Specification (OAS) is to REST. Since WSDL is XML, it can be read and edited, but in most cases it is generated and consumed by software.

The notation that a WSDL file uses to describe message formats is based on the XML Schema standard, which means it is both programming-language neutral and standards-based, making it suitable for describing web services interfaces that are accessible from a wide variety of platforms and programming languages. In addition to describing message contents, WSDL defines where the service is available and what communications protocol is used to talk to the service. This means that the WSDL file defines everything required to write a program to work with a web service. A WSDL document can be generated by Jade and consumed by another Jade or third-party application. Jade provides the necessary tools to do this quickly and easily.

UDDI

Universal Discovery Description and Integration (UDDI) is the yellow pages of web services. As with traditional yellow pages, you can search for a company that offers the services you need, read about the service offered, and contact someone for more information. You can, of course, offer a web service without registering it in UDDI.

A UDDI directory entry is an XML file that describes a business and the services it offers. There are three parts to an entry in the UDDI directory.

- The "white pages" describe the company offering the service: name, address, contacts, and so on.
- The "yellow pages" include industrial categories based on standard taxonomies such as the North American Industry Classification System and the Standard Industrial Classification.
- The "green pages" describe the interface to the service in enough detail for someone to write an application to use the web service. The way services are defined is through a UDDI document called a Type Model or **tModel**. In many cases, the **tModel** contains a WSDL file that describes a SOAP interface to a web service, but the **tModel** is flexible enough to describe almost any kind of service.

The UDDI directory also includes several ways to search for the services you need to build your applications. For example, you can search for providers of a service in a specified geographic location or for a business of a specified type. The UDDI directory will then supply information, contacts, links, and technical data to allow you to evaluate which services meet your requirements.

Jade may provide UDDI discovery and publication in a future release. However, at present there is little usage of this feature in the community. In fact, IBM, Microsoft, and SAP have now closed their public UDDI nodes.

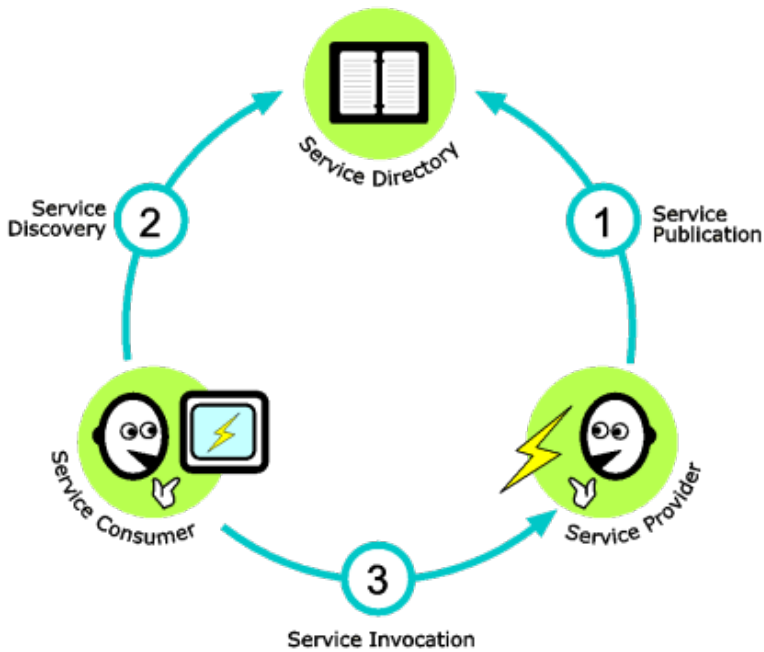
SOAP Web Services in Jade

The Jade Platform has a full implementation of both the web service provider and web service consumer features.

Jade web services currently support the XML 1.0, WSDL 1.1, SOAP 1.1, and SOAP 1.2 standards. In addition, Jade web services are also WS-I 1.0 compliant.

For more details, see the following subsections.

Architecture



Jade SOAP Web Service Provider

A SOAP web service provider is a node on the network (Intranet or Internet) that provides access to a software service that performs a specific set of operations. A service provider node provides access to the services of a business system, a subsystem, or a component.

The Jade SOAP Web service provider framework uses the Jade web application. An understanding of this framework is assumed in the following discussion.

Jade uses the HTTP protocol for communicating with web service clients or a direct connection over TCP for communicating with other Jade systems. By method reimplementation, it is possible to cater for other protocols as well.

The web services framework in Jade shields you from the complexities of working with SOAP messages. As far as you are concerned, you are creating Jade methods. The steps involved in creating a web service in Jade are as follows.

1. Add a web service by creating a subclass of the **JadeWebServiceProvider** class. Each of these subclasses will be a web service. You can define multiple web service classes within a single application.
2. Build the web service by adding methods to this class. Methods that are to be exposed will require the **webservice** option in the method signature. Any method added to a **JadeWebServiceProvider** subclass will by default be a web service method.
3. Define the properties to be exposed for the classes that will be returned by this service. A web service exposure form will list all classes that are required to be exposed and by default, all public and read-only properties will be selected.
4. Create a web-enabled web service application. This application will receive and respond to client requests. Set up the application options such as machine name, virtual directory, web service exposure or exposures, and secure service.
5. Extract the Web Services Description Language (WSDL) file. This file will have all of the necessary information for a web service client to create a SOAP message and communicate with your application. The generated WSDL conforms to WSDL 1.1.

That is it from the development perspective. Of course, in order to successfully execute your application, you will have to set up the virtual directory and the **jadehttp.ini** (IIS) or **jadehttp.conf** (Apache) file correctly for your web server. Once this has been done, you can test your web service by using the built-in web browser test harness or a client application.

Although the development process is relatively simple, careful design of your web service is important for the following reasons.

- Incorrect exposure of properties can lead to large response messages being generated. For example, in the example Erewhon system, if all of the properties were exposed, what looks like a simple call (like getting a single client by name) could end up returning almost all of the information in the database. This could end up generating a response string that is about nineteen (19) megabytes!
- Once the WSDL file is given to customers, it gets harder to change the interface as the interface is now published. Version control will have to be enforced and multiple versions of the service may need to be maintained.
- As with most applications, performance, scalability, and reliability must be considered when designing your web service. This becomes particularly important because a web service can be invoked without human interaction.

For more details, see the following subsections.

Running a Web Service Application in the Direct SOAP Mode

Before the implementation of Direct SOAP functionality, access to SOAP web service applications was available using only IIS. Client applications would make HTTP/1.1 SOAP requests to IIS, and then IIS would communicate with the Jade web application to serve responses to these requests. The communication between the IIS and the Jade web service application was performed by the Internet Server Application Programming Interface (ISAPI) extension Dynamic Link Library (DLL) called **jadehttp.dll**. This DLL implemented a proprietary Jade protocol between IIS and the SOAP web service applications.

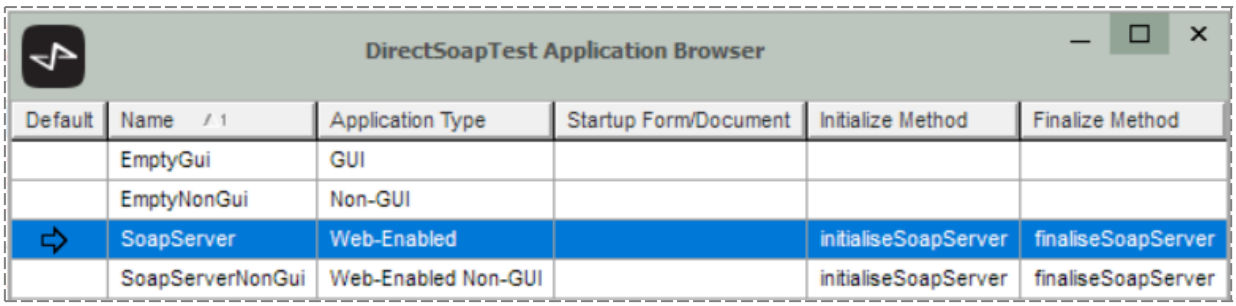
The Direct SOAP functionality adds support for the HTTP/1.1 communication protocol to web service applications. This means that Jade web applications, when run in the Direct SOAP mode, no longer require IIS to enable communication with other HTTP-compliant applications (for example, SOAP clients, web browsers, reverse proxies, web servers, or TLS termination servers). This allows a greater degree of freedom for the deployment of Jade web service applications and it also reduces the system requirements for the development of SOAP applications.

The general steps to define a SOAP web service provider application are specified in "Defining a Web Services Provider Application", in Chapter 11 of the *Developer's Reference*.

The only functionality not provided by the **JadeWebServiceProvider** class when SOAP applications are run in the Direct SOAP mode are the three methods related to the use of the legacy IIS virtual directory feature (that is, **createVirtualDirectoryFile**, **deleteVirtualDirectoryFile**, and **isVDFilePresent**). If these methods are called in a Direct SOAP application, exception 1068 (*Feature not available in this release*) is raised.

As the use of the virtual directory (specified in the **Virtual Directory** text box on the **Web Options** sheet of the Define Application dialog) is normally reserved for serving files and SOAP applications serve only XML-formatted SOAP responses, this loss of functionality is not expected to adversely impact applications.

In the following image, the highlighted entry is for a SOAP web service application called **SoapServer**. There is also another SOAP web service application called **SoapServerNonGui**, but these applications could also be called **Milly**, **Molly**, or **Mandy**.



Default	Name / 1	Application Type	Startup Form/Document	Initialize Method	Finalize Method
	EmptyGui	GUI			
	EmptyNonGui	Non-GUI			
⇨	SoapServer	Web-Enabled		initialiseSoapServer	finaliseSoapServer
	SoapServerNonGui	Web-Enabled Non-GUI		initialiseSoapServer	finaliseSoapServer

To run a SOAP web services application

- If you have the following setting of the **DirectSoap** parameter in the [WebOptions] section of the Jade initialization file, all **Web-Enabled** and **Web-Enabled Non-Gui** applications on your node run in Direct mode.

```
[WebOptions]
DirectSoap=true
```

- If you have the following (default) setting of the **DirectSoap** parameter in the [WebOptions] section of the Jade initialization file, all **Web-Enabled** and **Web-Enabled Non-Gui** applications on your node run in the legacy IIS mode.

```
[WebOptions]
DirectSoap=false
```

- If your node has a **Web-Enabled** application called **SudsyServer** and the following **DirectSoap** parameter settings, the application called **SudsyServer** runs in Direct SOAP mode but all other **Web-Enabled** and **Web-Enabled Non-Gui** web service applications on that node run in legacy IIS mode.

```
[WebOptions]
DirectSoap=false
SudsyServer_DirectSoap=true
```

In addition to the **DirectSoap** parameter, the [WebOptions] section of the Jade initialization file also provides the parameters listed in the following table.

Parameter	Specifies...
KeepAliveTimeout	The maximum time a connection will be held open after a response has been sent.
LogHttpMessages	Whether HTTP messages are logged. If logging of HTTP messages is enabled, they are also displayed in the console of GUI REST applications.
MaxRequestLength	Maximum length (in bytes) of HTTP requests for SOAP web service applications running in Direct SOAP mode.
MinimumReadDataRate	Minimum rate allowed for incoming data (defined in kilobits per second).
QueueDepthMax	Maximum size of the SOAP web service request queue.

Running Existing SOAP Web Service Applications in Direct Mode

The majority of pre-existing SOAP web service applications should be able to run in the legacy or Direct modes, without the need to change any application code.

To run an existing Jade SOAP web service application in Direct SOAP mode, simply:

1. Add the **DirectSoap** parameter with a value of **true** to the [WebOptions] section of the Jade initialization file.
2. Disable IIS.

> **Note** This step is not required if the SOAP web service application is configured to talk on any port other than the standard HTTP and HTTPS ports of **80** and **443**.

Applies to Version: 2022.0.05 and higher

Runtime Deployment

The web service URL settings can be set at run time in the [WebOptions] section of the **jade.ini** file or in the XML-based configuration file.

In the **jade.ini** file, use ***application-name*_WebServicesURL=*url*** to set it for a specific Jade application and use **WebServicesURL=*url*** to set a default value where there is no specific application value set. For example, in the **jade.ini** file for the development environment:

```
[WebOptions]
ErewhonWebService_WebServicesURL=http,developmentServer,jade,jadehttp.dll
```

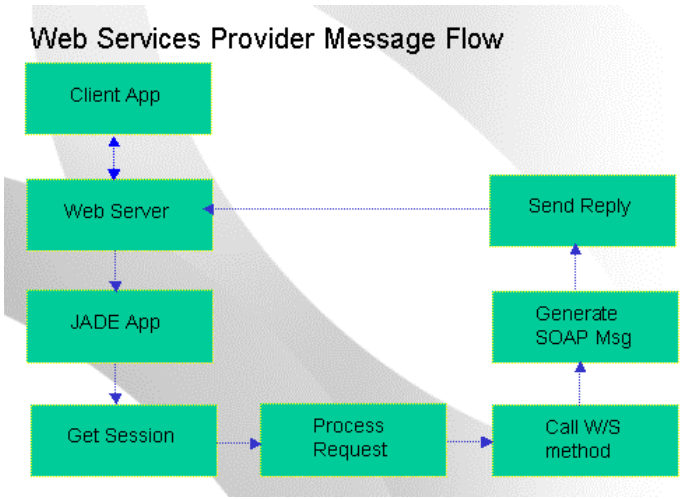
For the **jade.ini** for the production environment:

```
[WebOptions]
ErewhonWebService_WebServicesURL=http,productionServer,jade,jadehttp.dll
```

The XML configuration file allows several runtime configuration options, which can be created using a text editor or the application that is provided with Jade.

For details about configuring web services, refer to the [WebOptions] section in the *Jade Initialization File Reference* or "Configuring Web Applications" in Chapter 3 of the *Web Application Guide* for details about the Web Configuration application and XML-based configuration file settings.

Jade Web Service Provider Message Flow



Consider the following points about the web service provider message flow.

- The web server can be IIS or Apache.
- The Jade application must be a web-enabled or web-enabled non-GUI application.

  A non-GUI application can be run on any of the Jade-supported operating systems.
- The Get Session session handling is optional and if used, will create a persistent instance of the session subclass.
- The Process Request method can be reimplemented. This can be used, for example, to inspect the incoming SOAP message.
- The Send Reply method can be reimplemented. This can be used, for example, to inspect the response message.
- When the Jade application is executing, any exception that is raised will be converted to a SOAP fault and returned to the calling application.
- The message flow depicted in the above image is the default message flow when using the Jade-supplied web framework (which uses the HTTP protocol only).

SOAP Message Formats

WSDL 1.1 distinguishes between two message styles: document and RPC. Here's how each style affects the contents of **<soap:Body>**.

- **Document**

  <soap:Body> contains one or more child elements called *parts*. There are no SOAP formatting rules for what the **<soap:Body>** contains; it contains whatever the sender and the receiver agree upon.

- **RPC**

  RPC implies that **<soap:Body>** contains an element with the name of the method or remote procedure being invoked. This element in turn contains an element for each parameter of that procedure.

For applications that use serialization and deserialization to abstract away the data wire format, there is one more choice to be made: the serialization format. The current two popular serialization formats today are:

- **SOAP Encoding**

  SOAP encoding is a set of serialization rules defined in section 5 of SOAP 1.1 and is sometimes referred to as "section 5 Encoding". The rules specify how objects, structures, arrays, and object graphs should be serialized. Generally speaking, an application using SOAP Encoding is focused on remote procedure calls and will likely use RPC message style.

- **Literal**

  Data is serialized according to a schema. In practice, this schema is usually expressed using W3C XML Schema. Although there are no prescribed rules for serializing objects, structures, graphs, and so on, the service's schema describes the application-level Infoset of each of the service's messages.

There are therefore four possible variations in the message format. By default, Jade uses the document/literal format. If circular references are detected, the only way to currently represent this is in encoded format, so in this case Jade defaults to document/encoded. You have a choice of changing either of these defaults to RPC format, by using the application options. Jade's RPC format is always encoded.

Many people believe that a shift away from SOAP encoding is inevitable. The W3C XML Protocol Working Group's SOAP 1.2 specification makes support for SOAP encoding optional (that is, a toolkit can claim SOAP 1.2 compliance without supporting SOAP encoding), the WS-I Basic Profile Working Group's interoperability guidelines (Basic Profile Version 1.0a) disallows the use of SOAP encoding with SOAP 1.1, and the W3C Web Service Description Working Group has dropped support for encoding from the WSDL 1.2 specification.

Before Jade can drop support for encoded formats and conform to WS-I's Basic Profile, there has to be an XML schema-friendly way to handle circular references. A one-to-one relationship is an example of a circular reference. If both properties in this relationship are exposed, the WSDL that Jade generates will be in encoded format.

Versioning Options

When implementing a new version of a web service, there are some instances in which you can simply enhance the existing class and others where you need to implement a new class that can use the previous version.

The most-common tasks you will face when updating a web service are:

- Adding extra methods. The new methods are conceptually related to the existing web service and should be implemented on the same endpoint.
- Changing method signatures. In this case, the number of parameters, the type of a parameter, or the return type changes.
- Updating the data model. In this case, classes are added, properties are added or deleted, or have their name or type changed.

For details, see the following subsections.

Adding Extra Methods

In this case, the new methods can be added to the current web service class and existing clients will continue to work without any problems. A new WSDL file can be generated from the new definitions for new clients and existing clients who require the new functionality. A new Jade application need not be created.

Changing Method Signatures

There are at least three choices, as follows.

- Create a method with a different name and add it to the existing class. This is the same as adding extra methods.
- Create a new class with the same method name. In Jade, this new class cannot be a subclass of the existing class because the method signatures are different.

  You will therefore need to copy the methods whose signatures have not changed to this new class as well, to retain the existing functionality. Even though you can select multiple web service classes to be defined for an exposure, as the generated WSDL cannot contain two methods with the same name, creating a completely new class and a new application is therefore necessary.
- Change the current method's signature, supply the updated WSDL to your web service clients, and use version control to check and reject invalid requests.

Updating the Data Model

Adding a property but not exposing it will not have any effect on existing client systems. However, exposing this property will cause XML to be generated for it and then whether this works or not will depend on how the client system handles the message. Similarly, if an exposed property is deleted or its name is changed, the client system can ignore the fact that a property value it expects is not there or there is a property value with a name that it does not recognize. However, if the type of a property changes, it is likely that the client system will fail, especially if the change is significant; for example, changing a property type from a primitive type to a class, or the reverse.

In order to ensure compatibility with existing client systems, it is generally safer to assume that structural changes to exposed classes may cause a problem. In this case, you have at least two choices.

- Make the change in your existing system and provide your web service clients with the updated WSDL file and other relevant documentation for them to make the necessary changes in their application. Set up version control in the Jade application so that requests that do not match the correct version can be rejected (Jade does this for you, by returning a SOAP fault).
- Make the changes in a separate copy of the system and set up version control in the Jade application for this system. You may need to set up other options such as the machine name and virtual directory so that the requests are directed to the correct versions of the application.

SOAP Faults

When an error is raised during development, you will want to know where the error originated. Because this information is not useful to consumers of the web service, you won't want to return meaningless line numbers when the service is deployed. Instead, you will want to provide other contextual information about what happened.

The SOAP Fault element has four separate pieces. In the following list, the bold names are for SOAP 1.1 and the SOAP 1.2 names are shown in italics.

- **faultcode** *(Fault)*: Contains a value of **VersionMismatch**, **MustUnderstand**, **Client**, or **Server**.
- **faultstring** *(Reason)*: Provides an explanation of why the fault occurred.
- **faultactor** *(Role)*: Indicates the URI associated with the actor that caused the fault on the message path. In RPC-style messaging, the actor should be the URI of the invoked web service.
- **detail** (*Detail*): Carries information about why the error happened. This element can contain more XML elements or it could just be plain text.

The fault codes fall into the following categories.

- **VersionMismatch**: The SOAP receiver saw a namespace associated with the SOAP envelope that it does not recognize. When this fault code is received, the message should not be re-sent. The SOAP namespace needs to be set to something the receiver *does* understand. Jade returns this code when the incoming namespace does not match the namespace of the web service application.
- **MustUnderstand**: An immediate child of the SOAP header had **MustUnderstand** set to **true**. The receiver of the message did not understand the header. The receiver will need to be updated somehow (new code, new libraries, and so on) in order to make sense of the header. This fault code is currently not supported by Jade.
- **Client** *(Sender)*: Something about the way that the message was formatted or the data it contained was wrong. The client needs to fix its mistake in order for the message to be sent back. When returning this fault code, you should also fill in the details element with some specifics on what needs to happen in order for the message to be processed. This fault code is returned by Jade if the service, method, or parameters are invalid.
- **Server** *(Receiver)*: An error happened at the server. Depending on the nature of the error, you may be able to resend the exact same message to the server and see it processed. Jade returns this fault code if the method execution fails.

When a Jade exception is raised on the web service provider, the fault is converted to a SOAP fault message and returned to the client. SOAP faults are returned as HTTP 500 errors.

Using SOAP Headers

Those familiar with HTTP or MIME headers are probably used to seeing various sorts of metadata included with the main data in the message. In a lot of ways, the SOAP header is similar, with *one* major difference.

HTTP uses the Content-Type header to indicate the MIME type of the data in the body of an HTTP request or response. Similarly, an HTTP client can request what kind of data it wants in the response, by including the HTTP Accept header. From a high level, SOAP messages always contain XML data, so in that sense there is no need to specify a MIME type to describe the data. In fact, the structure of the data in SOAP messages is much better defined through the use of XML schema.

A web service that defines its interface through WSDL defines the schema of its data along with the bindings that specifies what response data types will be generated from which request data types.

The well-defined nature of SOAP messages is what allows them to be so easily used from within applications. Therefore, because the data structure is already defined, using SOAP headers to describe the data structure in a SOAP message is unnecessary.

The focus of the SOAP header should be to help process the data in the body. It makes sense to include information about authentication or transactions, because this information will be involved in identifying the person or company who sent the body and in what context it will be processed. Expiration data could be included in the header, to indicate when the data in the body may need to be refreshed. User account information could be included, to ensure that processing the message is performed only for a request that has been legitimately paid for.

Another factor in determining whether information should be included in SOAP headers is whether that information will have broad application to a wide variety of SOAP messages. If so, it should be included in the header. It makes more sense to define a single schema and insert it into the definition of one header element than to force inclusion of the same data into the body schemas of a large number of message definitions. Authentication and routing are problems common to many web services, so it makes sense that this information lives in the header element.

In Jade, SOAP headers are defined as a subclass of the **JadeWebServiceSoapHeader** class. Properties that are to be included in the header are then defined on this class. These classes can then be included in a web service definition, by adding properties to the service of this type. Individual methods can then be assigned these headers.

When session handling is enabled for a web service application, a SOAP header is automatically generated for every method call. This header carries the current session id. The client system does not need to process this header but it is required to return the header back to the Jade web service provider. The header is defined as input-output so that the client knows to do this.

Documenting Your Web Service

Documentation for a web service needs to contain several different elements.

1. First and foremost, it should provide a Web Services Description Language (WSDL) file that programmatically describes the web service.
2. Secondly, it needs to provide written documentation describing how to use the web service. This should include various items, including an API reference, troubleshooting tips, and usage descriptions.
3. Finally, the documentation should provide sample code for all of the operations, preferably using the fewest lines of code needed to call the specified method. Examples of SOAP messages going back and forth should be included, along with the code. These sample messages will help developers to develop a client in languages other than those outlined by the samples. Ideally, the documentation should also include a sample client that uses the web service, complete with source code.

For more details, see the following subsections.

WSDL Files

When documenting a web service, you must provide a WSDL document. This document provides critical information about the web service that both the developers and programming tools need. In a compact, concrete way, this document describes everything, including:

- Messages that the web service understands and the format of its responses to those messages
- Protocols that the service supports
- Where to send messages

All of this information combines to give the programmer a view of how the system expects outside applications to interact with the web service. The WSDL is therefore the main piece of documentation your users need.

The WSDL file can be generated in Jade.

Usage Documentation

The documentation for your web service should also describe how you expect people to use your web service. Explain how errors will be returned, how to initiate usage, and so on. This information will help get others up and running with your web service. Unless you are doing something simple like retrieving a stock quote based on ticker symbol, people are going to need good documentation.

First, include an overview document. A good overview contains pointers to and summaries of the documentation relating to the web service: WSDL locations, developer guides, API reference, and so on. Within the developer guide, explain how the web service is to be used. Describe typical usage scenarios, as well as error handling.

When describing error handling, list errors that can be returned for *every* web service method. Give the return codes, so that client developers can look up the error number and display a meaningful message to their end-users in either a display message or a log entry. You could add a method to the service that given an error code, will return a message describing the error and how to correct it.

Besides error handling, you will also want to document the various operations in the web service. This should look like any other API documentation.

- Explain what the operation does
- Define the meaning and type of the parameters of the operation
- Provide sample code

Give Helpful Hints

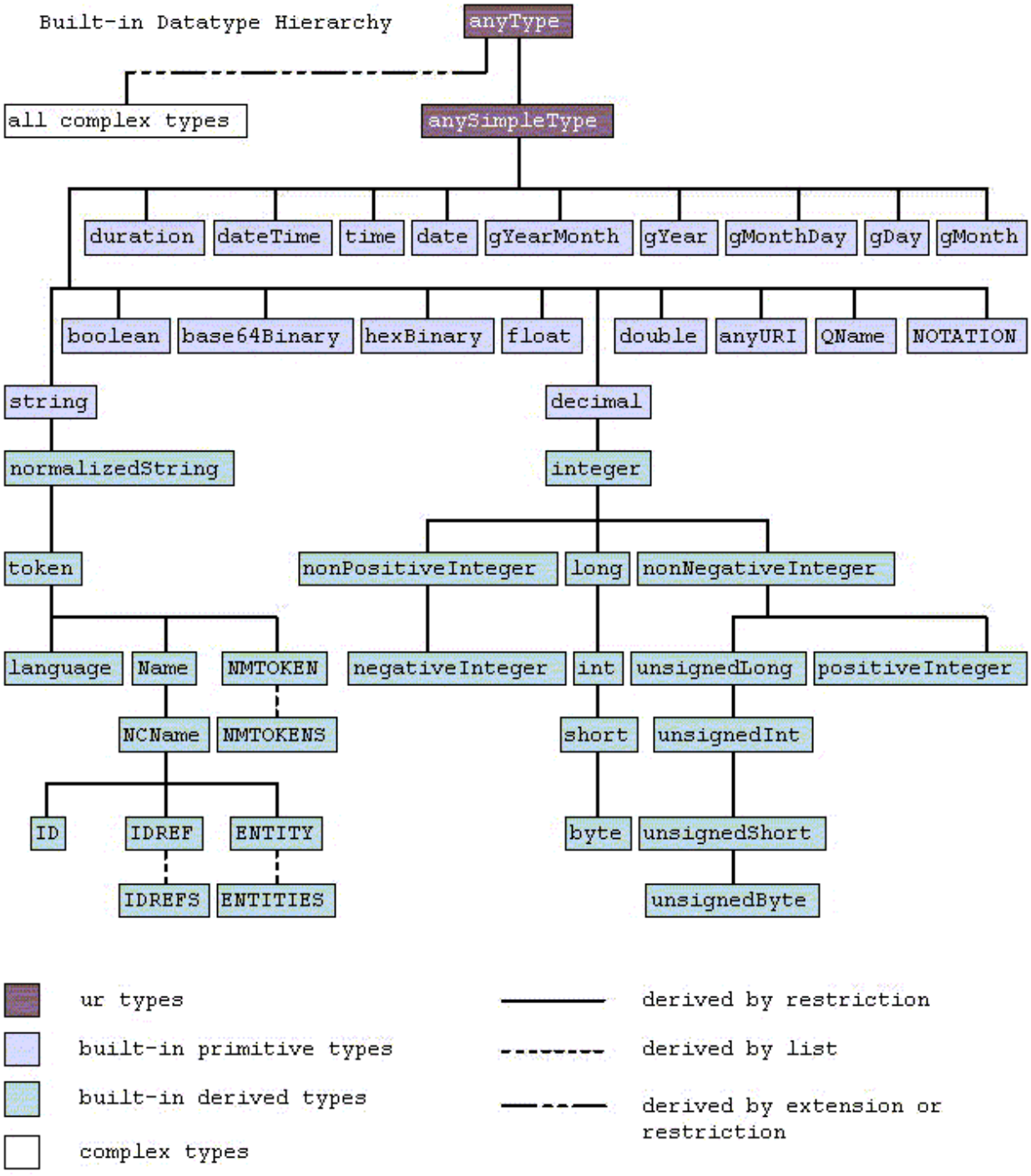
In addition to the above, give a sample SOAP message exchange dependent upon the communication pattern used (one-way, request-response, and so on).

Finally, take some time to develop a sample client that uses most, if not all, of the operations exposed by the web service. Make sure that the sample actually looks like something you expect a client developer might want to build. This reference may prove to be more useful than you think — the developer can use the sample to verify if the problem is in his or her implementation or somewhere with the web service itself.

Make use of the text feature for classes, methods, and properties to document your web service. The text will be extracted as part of the WSDL generation, thereby providing documentation in the WSDL file itself.

Mapping Jade Types to XML Schema Types

The following image represents the XML built-in data type hierarchy.



The mapping of Jade types to XML is as follows.

Jade Attribute	XML Simple Type	XML Examples (Delimited by Commas)
String, StringUtf8	string	Confirm this is electric
Character	unsigned byte	1, 126
Byte	byte	-1, 126
Binary	base64Binary	GpM7
Integer	int	-1, 12678967543233
Integer64	long	-1, 12678967543233
Decimal	decimal	-1.23, 0, 123.4, 1000.00
Real	double	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN
Boolean	boolean	true, false, 1, 0
Time	time	13:20:00.000, 13:20:00.000-05:00
TimeStamp	dateTime	1999-05-31T13:20:00.000-05:00
TimeStampInterval	duration	P5Y2M10D, P5Y2M10DT15H
TimeStampOffset	dateTime	1999-05-31T13:20:00.000-05:00
Date	date	1999-05-31

Note the different format for storing **TimeStamp** information in XML.

A Web Service Provider Example

The SOAP web service provider example in the following subsections is based on the Erewhon Investments example system, available on the Jade public GitHub at https://github.com/jadesoftwarenz/JADE-Erewhon. Load the schemas and generate the data. For details, see the *Erewhon Demonstration System Reference* document.

In this example, we will create a new application called **CustomerService**.

Note The Erewhon Investments example system already includes a web service provider, which may be a useful example as you complete the steps described in the following subsections. was built following the steps described in the following subsections.

In the Erewhon example schema there are two provider applications: **WebServiceOverHttpApp**, which uses normal web services over HTTP, and **WebServiceOverTcpApp**, which uses Jade-to-Jade web services over TCP/IP. Both applications provide the same functionality.

Creating the Web Service Class

In the **ErewhonInvestmentsViewSchema**, we add a subclass to the **JadeWebServiceProvider** class. This class is called **Customer**.

You can add properties to this class, but bear in mind that state information cannot be stored between requests because the transient instance of this class that is created for the request is deleted when the response has been sent.

We also add the following text to this class.

```
This service is used to access client information on the Erewhon system.
A list of clients can be obtained as well as details for an individual client.
```

```
Client details can also be updated using this service.
```

Creating the Web Service Methods

We add four methods, described in the following subsections, to this class.

- getClientNames
- getClient
- updateClientWithProxy
- updateClient

Method 1: getClientNames

The **getClientNames** method will return an array of client names for a company. The company used is the company defined for the application.

```
getClientNames(): StringArray webService;
vars
    names: StringArray;
    client: Client;
begin
    create names transient;
    foreach client in app.myCompany.allClients do
        names.add(client.name);
    endforeach;
    return names;
end;
```

We also add the following text to this method.

```
This method will return a string array of client names.
```

Method 2: getClient

The **getClient** method will return a **Client** object, based on an input parameter that contains the name of the client to search for. If the client does not exist, an error is returned.

```
getClient(clientName: String): Client webService, updating;
vars
    client: Client;
begin
    client := app.myCompany.allClients[clientName];
    if client = null then
        setError(23, clientName, "Client does not exist");
    endif;
    return client;
end;
```

We also add the following text to this method.

```
Given a client name, this method will return a client object. If a client with
the supplied name does not exist, error 23 will be returned.
```

Method 3: updateClientWithProxy

The **updateClientWithProxy** method will update a client object based on an input parameter that is a transient client object.

If the client does not exist or if the update fails, an error is returned.

The following method shows an example of using an input parameter that is not a primitive type.

```
updateClientWithProxy(proxyClient: Client) updating, webService;
vars
    client : Client;
    result : Integer;
begin
    client := app.myCompany.allClients[proxyClient.name];
    if client = null then
        setError(23, proxyClient.name, "Client does not exist");
        return;
    endif;
    result := app.myTA.trxUpdateClient(client, client.edition,
                                       proxyClient.myAddress);
    if result <> 0 then
        setError(result, proxyClient.name, global.getErrorString
                (app.getLastError));
    endif;
end;
```

We also add the following text to this method.

```
This method takes a client proxy object as parameter and updates the persistent
copy of the client object with details from the proxy. If the persistent object
does not exist, error 23 is returned.
```

Method 4: updateClient

The **updateClient** method will update a client object based on several primitive type input parameters. If the client does not exist, an error is returned. The following method is an alternative implementation to the method above.

```
updateClient(name, street, city, country, phone, fax, email,
             website: String) webService, updating;
vars
    client:  Client;
    result:  Integer;
    address: Address;
begin
    client := app.myCompany.allClients[name];
    if client = null then
        setError(23, name, "Client does not exist");
        return;
    endif;
    address := create Address(name, street, city, country, email, fax, phone,
                              website) transient;
    result := app.myTA.trxUpdateClient(client, client.edition, address);
    if result <> 0 then
        setError(result, name, "Client update failed");
    endif;
epilog
```

```
        delete address;
    end;
```

We also add the following text to this method.

```
    This method takes several string parameters and updates the persistent copy of
    the client object with details from the parameters. If the persistent object does
    not exist, error 23 is returned.
```

Note The framework will delete return types that are transients (for example, the **StringArray** in method 1 (**getClientNames**) when processing is complete.

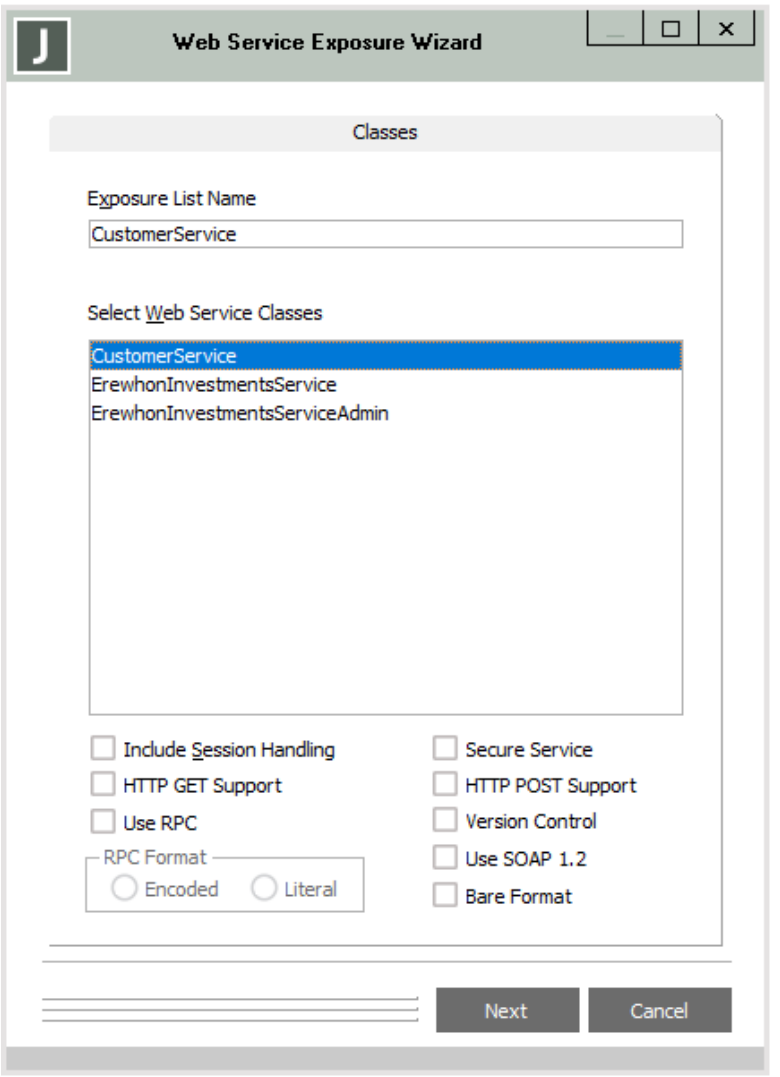
If you do not want this behavior, set the **deleteTransientReturnTypes** property to **false**. You should set this property in the **create** method of your **JadeWebServiceProvider** subclass.

Creating the Exposure List

The classes to be exposed in a web service are deduced from the parameters and return types for every web service method exposed by the service.

When defining this exposure, the properties that are to be exposed for this service must also be defined. Use the exposure wizard to do this. You can define multiple exposures for each application. In addition, you can use the same exposure to define more than one web service application. To achieve this, you must define the exposures separately.

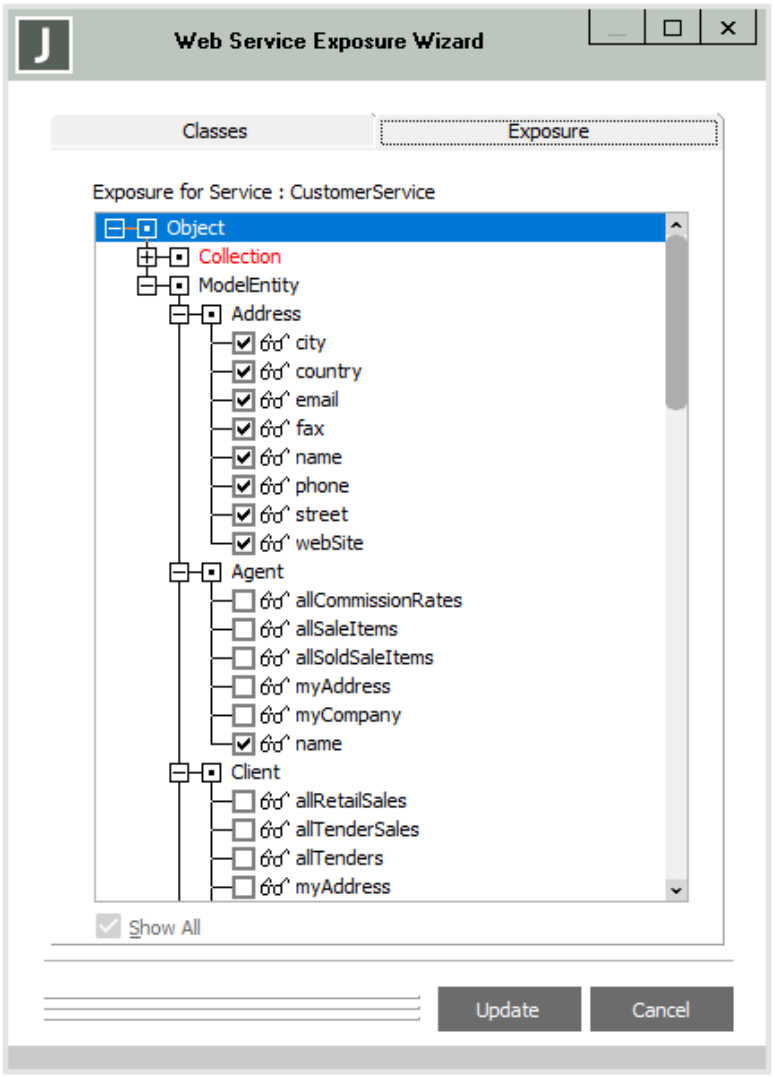
To create a new exposure, use the Exposures Browser **Web Services** sheet, accessed by selecting **Add** from the Exposures command when the Exposure Browser has focus and then selecting the **Web Services** sheet. The following dialog is then displayed.



The list of web service classes defined for your schema will be displayed. You will need to select at least one of these classes.

Other web service options (for example, session handling and version control) are also set up in this dialog. For details about specifying application web services, see "Defining a Web Service Provider Application", in Chapter 11 of the *Developer's Reference*.

Clicking the **Next** button displays the second page of this dialog.

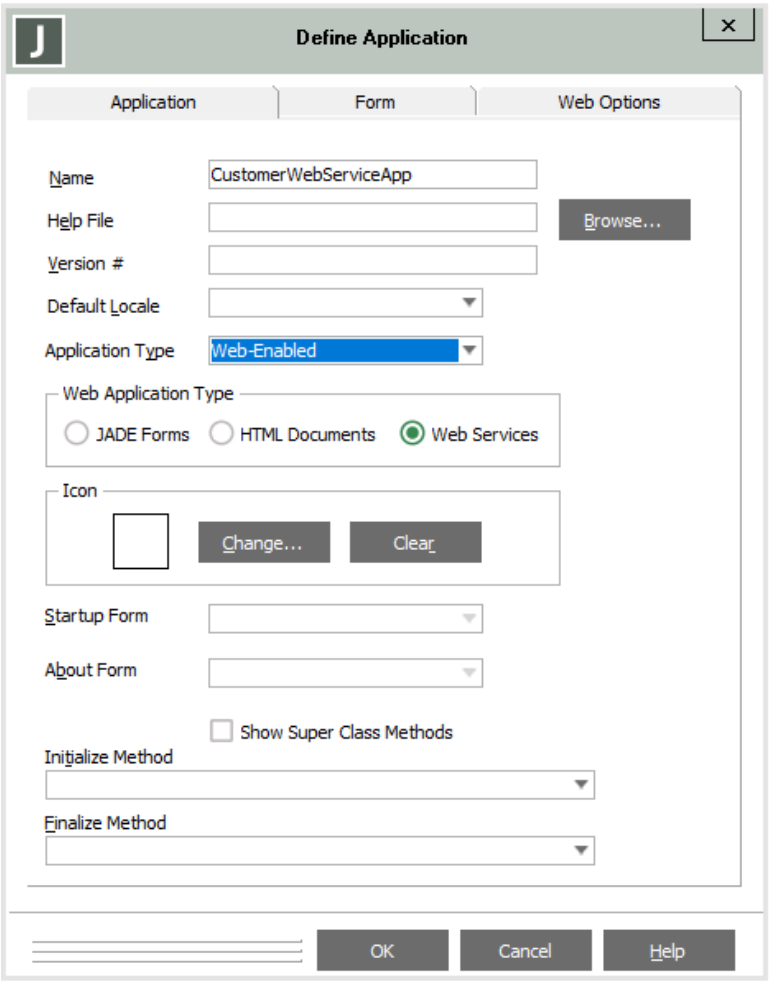


Select the properties that need to be exposed and then click the **Update** button, which will save the exposure.

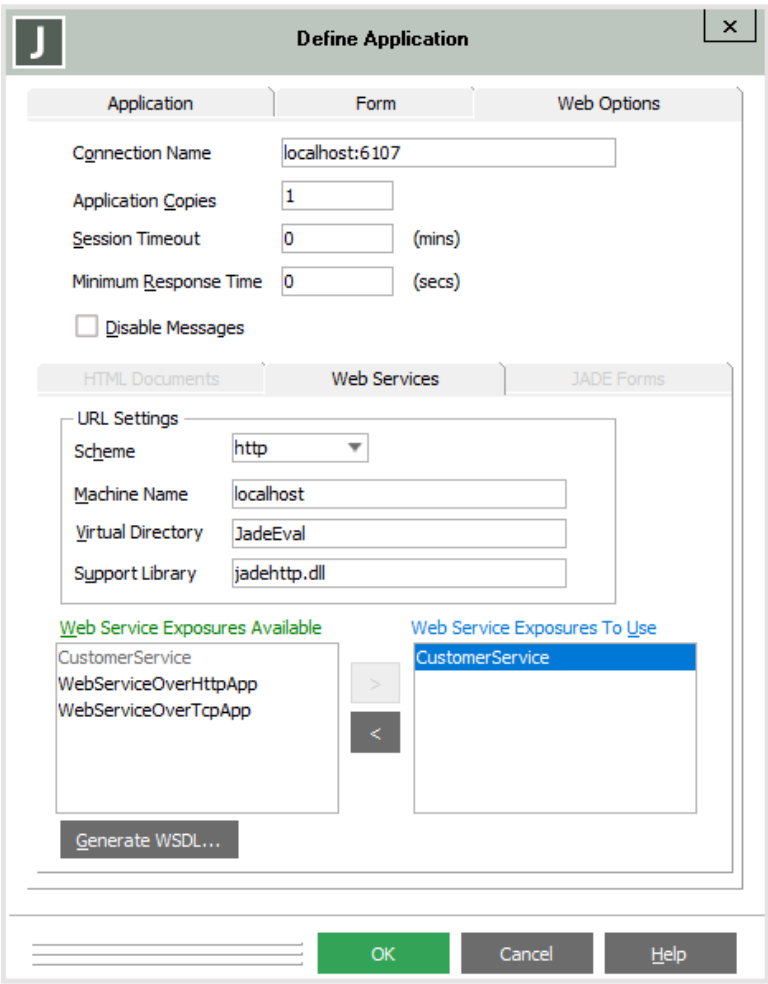
Creating the Web Service Application

When you have added the required methods, you can now set up the application.

From the Application Browser, add a new application and call it **CustomerWebServiceApp**. Set up this application to be web-enabled and the web application type to be a web service.



In the **Web Options** sheet, specify your web service options and then select the **WebServiceOverHttpApp** as your web service exposure. In the example, scheme is kept at the default value of **http**, **localhost** is used as the machine name, **6107** is used as the port, and **JadeEval** as the virtual directory.



You can also use Jade-to-Jade web services, which allow a web service consumer to connect directly to the web service provider without the need for a web server. As the messaging protocol is specific to Jade, the web service consumer must be a Jade system. To use the Jade-to-Jade web service option, change the scheme to **tcp** and enter a machine name of **localhost:1234**, where **localhost** is used as the machine name and **1234** represents the TCP port number to be used.

Nothing else needs to be set to use this option. Note also that nothing needs be set up in the IIS or Apache web server, as the communication between the Jade web service provider and consumer uses TCP directly and bypasses the web server.

Generating the WSDL

You can now generate the WSDL by clicking the **Generate WSDL** button.

Generate and save the WSDL file.

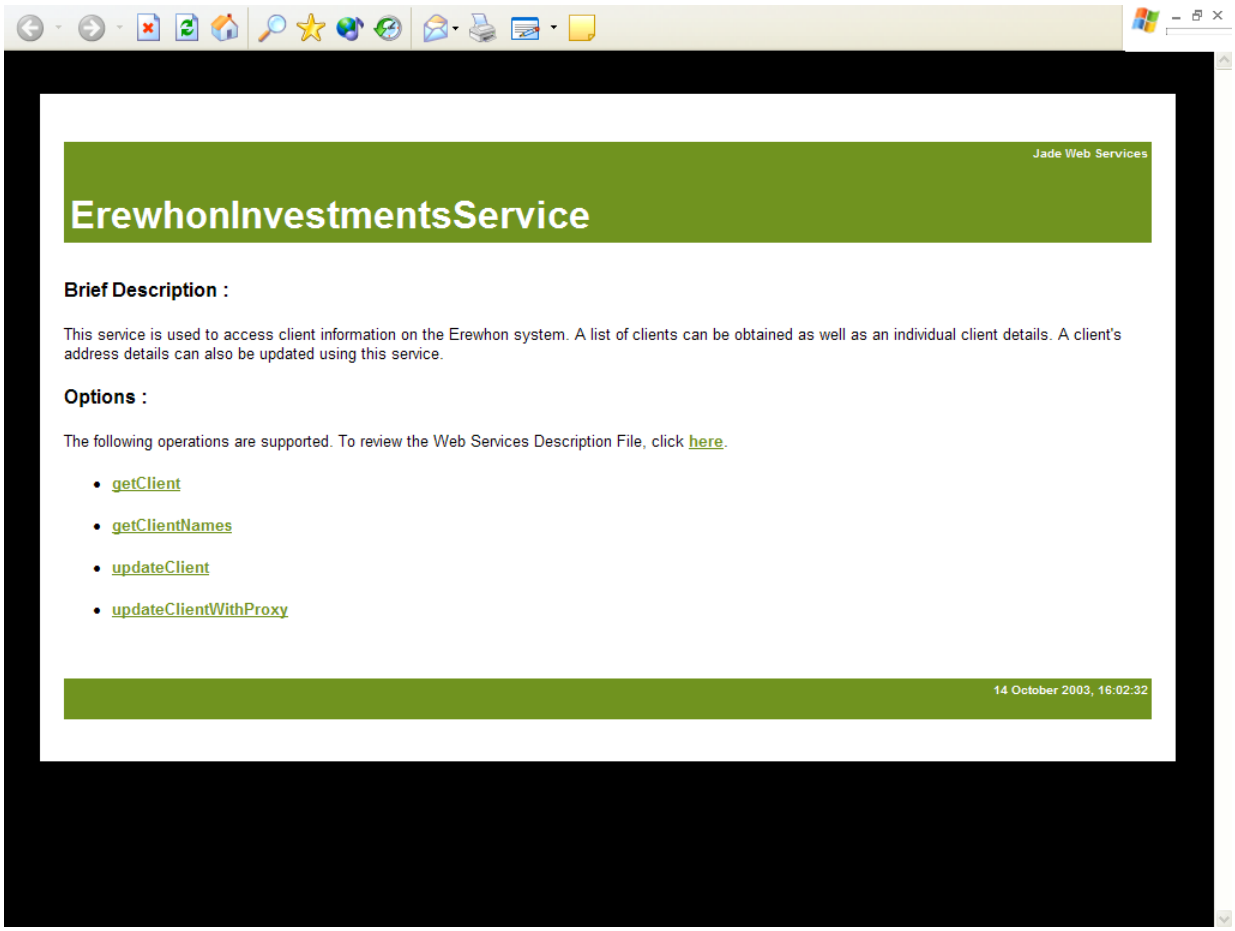
Note that you can select more than one exposure from this dialog. As each exposure needs to be written to a separate WSDL file, selecting multiple exposures will prompt you for multiple file names.

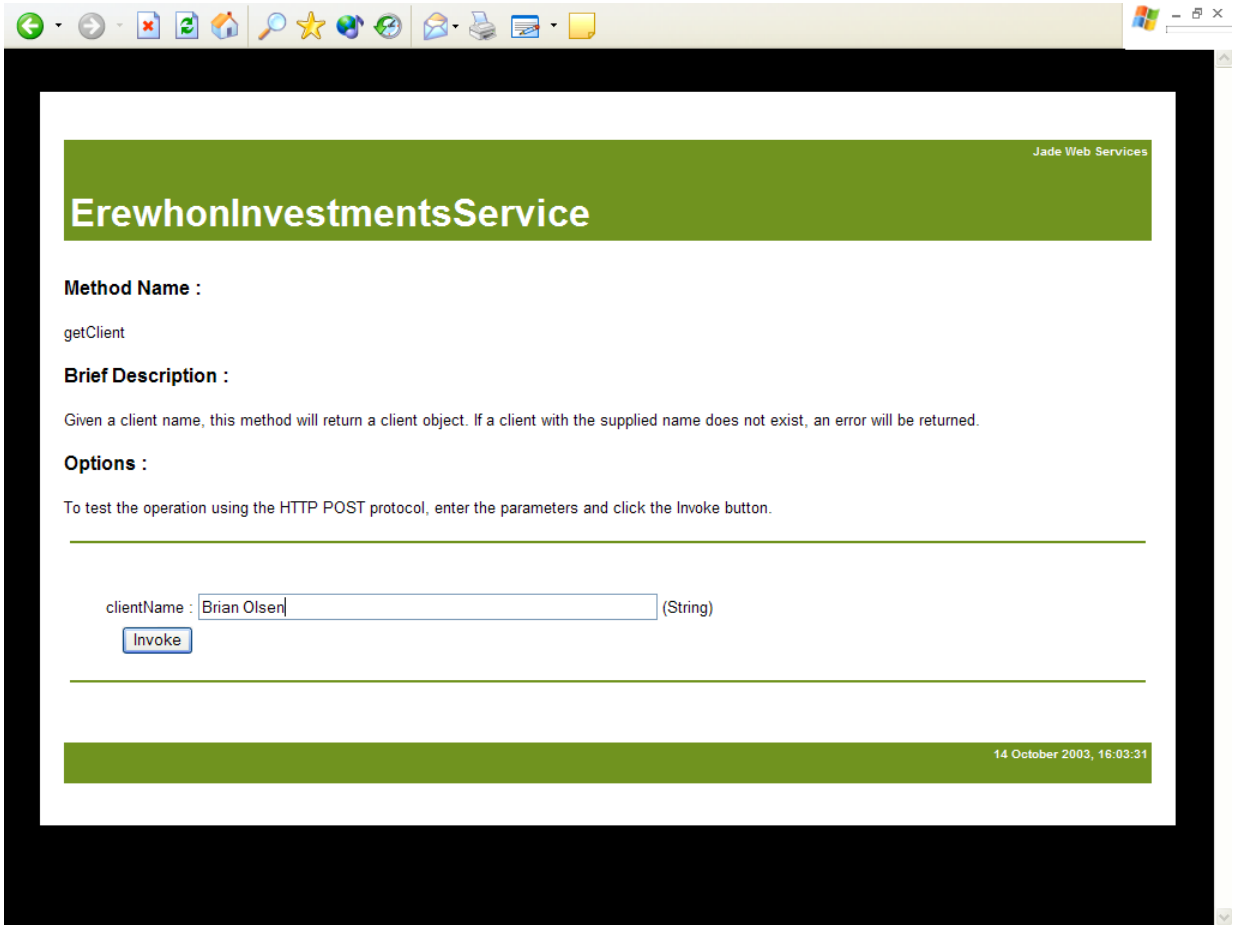
Using the Test Harness

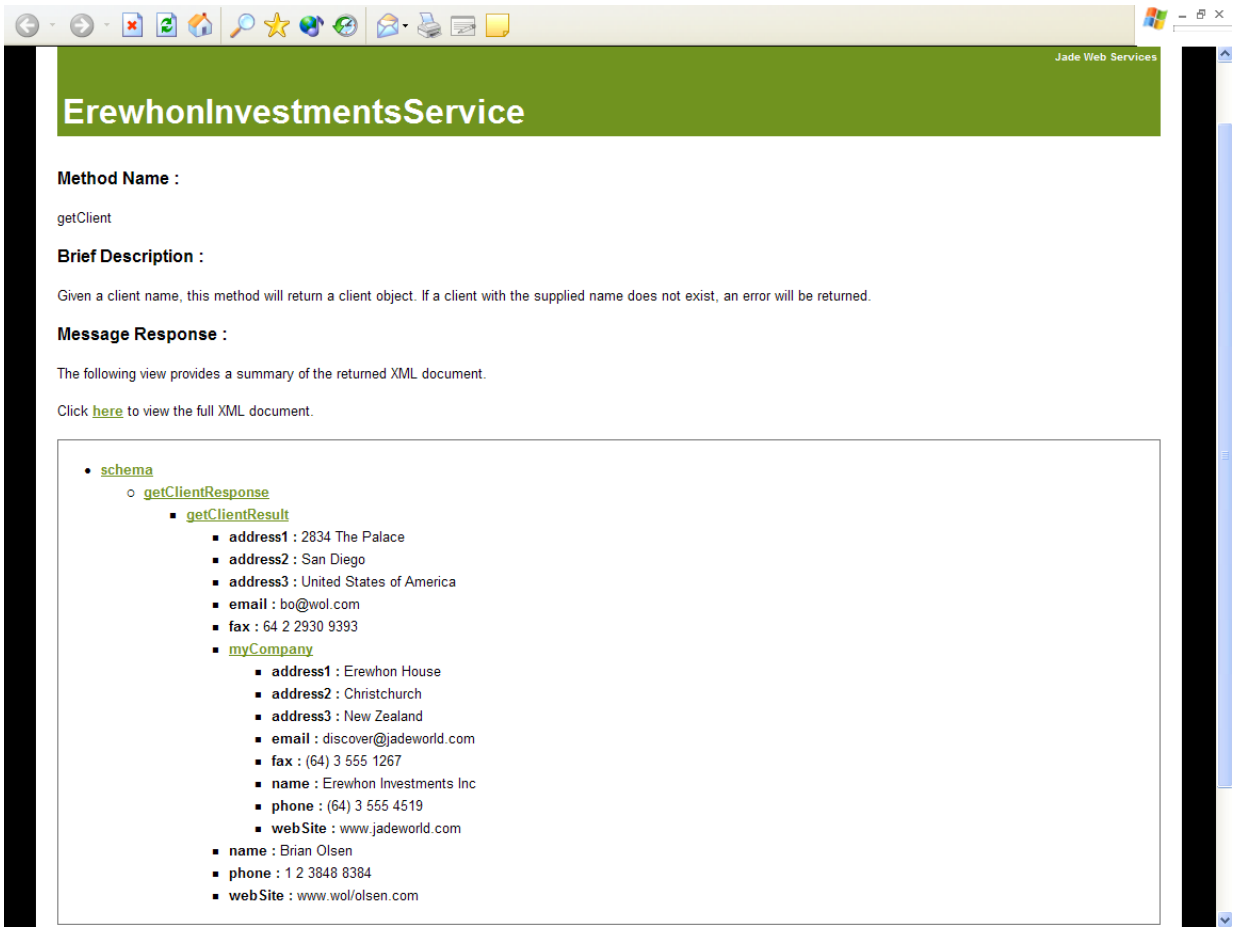
Jade provides an in-built test harness whereby you can enter a URL containing the name of the web service into a browser and test the functionality of your web service.

You will need to set up your virtual directory on your web server and update the initialization file for the **jadehttp** module. For more details, see "Configuring JadeHttp for Remote Connections", in Chapter 2 of your *Installation and Configuration Guide*. Once all of this is done, you can run the web service application and use the **Browser** menu item in the File menu to bring up the test harness on your browser.

The following images illustrate a sample session using the test harness.




Jade Web Services
ErewhonInvestmentsService
Method Name :
getClient
Brief Description :
Given a client name, this method will return a client object. If a client with the supplied name does not exist, an error will be returned.
Options :
To test the operation using the HTTP POST protocol, enter the parameters and click the Invoke button.
clientName : Brian Olsen (String)
Invoke
14 October 2003, 16:03:31



The test harness cannot be used when any of the parameters to the method are not primitive types. For example, the **updateClientWithProxy** method cannot be invoked using the test harness.

Jade Web Services Client

The Jade web services framework shields you from the complexities of working with SOAP messages. As far as you are concerned, you are using Jade methods. For more details, see the following subsections.

There are many tools available today to write a web service client. A WSDL file that was generated in Jade can be imported into a .NET application. Using the Add Web Reference dialog to load a WSDL file into a .NET application creates a file of web service method calls and proxy classes.

Using this generated proxy, you can then create a client application with a GUI front-end.

Creating a Jade Web Services Client

The steps involved in creating a web service client in Jade are as follows.

1. Access the Web Service Consumer Browser from the **Web Service Consumer** menu item in the Browse menu and then add a consumer, by selecting the **Add** menu item in the Consumer menu.

2. From the Web Service Consumer Wizard, enter a WSDL file name if the file is on disk or enter the URL of a WSDL file that is available via the network.

3. Click the **Validate** button. A default consumer name is generated and a list of web service methods with parameters and return types is displayed. The consumer name is used to create a subclass of **JadeWebServiceConsumer** containing these methods.

4. Click the **Next** button. A list of class and property names is displayed. The names from the WSDL are shown on the left and the corresponding Jade names on the right. The Jade names will be different and highlighted in orange if the WSDL names do not conform to the Jade naming rules. You can change any Jade name and add a prefix to all class names, property names, or method names. You can also change the superclass of all created classes from the default of **Object**. The Erewhon sample schema **WebServiceConsumerSchema** has a superclass of **ErewhonAdmin** and a class prefix of **WS_** applied.

5. If you check the **Generate methods for asynchronous calls** check box, methods for consuming the web service asynchronously are generated in addition to the methods for synchronous execution.

   Tip For details about and examples of running web services synchronously and asynchronously, see "Using the Imported Web Service Consumer", in Chapter 11 of the *Developer's Reference*.

6. If you check the **Generate new primitive types** check box, the web service consumer classes and methods generated from the WSDL use the primitive types **Integer64**, **Byte**, and **TimeStampInterval** where appropriate. (These primitive types were not available in earlier implementations of web services.)

7. Click the **Update** button. These classes are then all automatically added to the current schema.

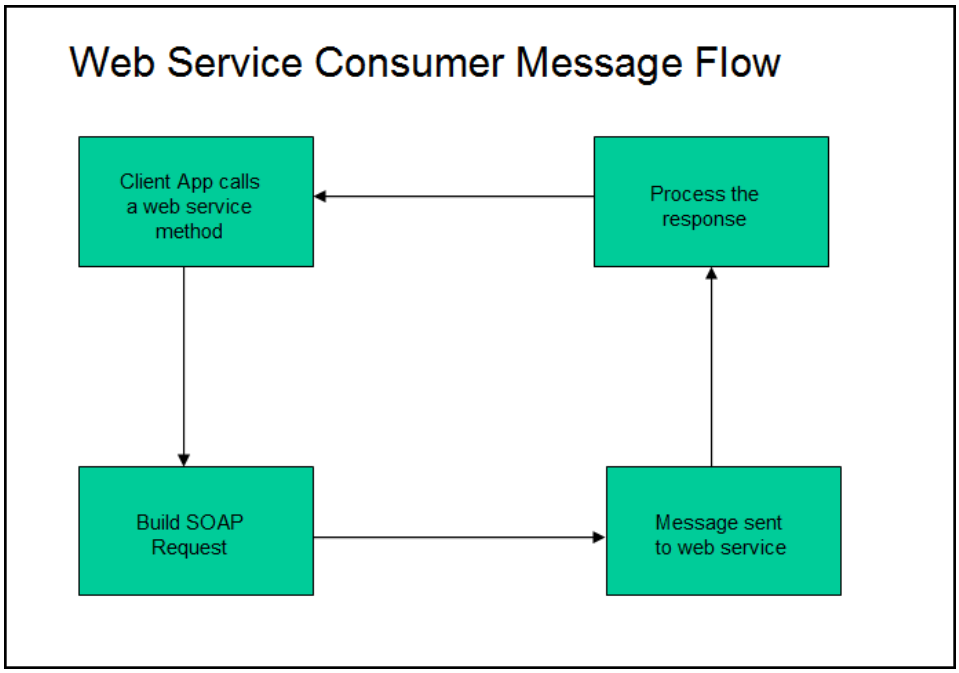
Using a Jade Web Services Client

To use the web service consumer, write user logic to create an instance of the **JadeWebServiceConsumer** subclass and call the required methods with the parameters. Jade will automatically package and send a SOAP message with the method request and parameter values to the web service provider, wait for the SOAP response, and unpackage the values into the web service method return value plus any **io** or **output** method parameters.

After the call to the web service provider method, the return value will be automatically populated. If this is a class with references to and collections of other classes, transient instances of these classes will have been created with data from the incoming SOAP message. The references will be established and the collections populated, including any primitive and object arrays.

The data can then be accessed and used as if the method accessed local Jade resources. If the web service provider is unavailable or there are connection problems, appropriate Jade exceptions are raised.

Message Flow



Starting with the called **JadeWebServiceConsumer** subclass method:

- **sendRequest**, which builds the SOAP request.
- **invoke**, which sends the SOAP request out via HTTP (or TCP) and receives the response.
- **processReply**, which raises a server error exception if the response is not a valid SOAP message, converts a SOAP error message into a Jade exception, and populates the method return values.

You can reimplement the **invoke** method to examine and possibly change the input SOAP message (the value of the **inputMessage** method parameter) before it is processed.

Web Service Styles

As explained for the web service provider earlier in this document, Jade supports both Document- and RPC-style web services. In the consumer, this information is part of the WSDL definition, and Jade will build the web service consumer methods and classes differently for the two styles.

For **RPC-style**, classes are built for each definition in the WSDL that is not a primitive type; that is, all classes used as web service consumer method parameters and return types, plus all other classes to which they refer. The web service consumer parameters and return types are Jade primitive types or the classes used as web service consumer classes. This gives a very natural Jade-like system, and for a simple Jade-to-Jade service, may be the easiest way to code but it requires careful design of the web service provider methods, to avoid frequent reloads and changes in the consumer as method signatures are altered. In addition, the trend in web services is away from RPC-style towards Document-style.

For **Document-style**, classes are built as for RPC-style but two additional classes are built for each web service method: one containing the parameters and the other for the return value. The Erewhon schema **WebServiceConsumerSchema** shows this. To call the web service method, an input parameter object is created, populated, and used as the method parameter, and an output parameter object is automatically created and populated as needed by the method call.

This requires a little more coding, but with careful design can be a much more flexible mechanism, as you can write a few general-purpose web service methods and use one or more of the parameters to determine the actual processing.

Transients

The web service consumer code keeps track of all transients that it creates, and these are deleted when the **reset** method is called or the web service consumer object is deleted. If the web service consumer object is re-used for multiple method calls, you should call the **reset** method before each such use.

Any transient objects that are created in your code for web service consumer method parameters should be deleted in your code when they are no longer required.

SOAP Headers

If the imported WSDL includes details for SOAP headers, they will be automatically built as subclasses of **JadeWebServiceSoapHeader** and references created from the **JadeWebServiceConsumer** subclass to them. To populate them on output, just set the values before calling the web service consumer method.

If the target web service provider returns values in the SOAP headers, they will be automatically updated from the web service consumer method call.

Updating a Consumer

To update a Jade web service consumer from an updated WSDL, follow a similar procedure to the initial consumer creation described earlier in this document, accessing the **Web Service Consumer Browser** from the **Web Service Consumer** menu item in the Browse menu. Select the required consumer and then the **Reload** menu item in the Consumer menu. Click **OK** on the Warning message box that is displayed, then follow the rest of the steps in the earlier description of the initial web service consumer creation.

The existing classes and properties created from the prior consumer creation plus the consumer methods, whether renamed or not, are retained if the new WSDL still includes definitions for these under their original names. Your existing code referencing the created classes will need changing only if there are previous classes, properties, or consumer methods that are no longer in the new WSDL or whose definitions have changed.

Changing the End Point

The URL for a web service is composed of several parts, as follows.

- Scheme

  Within the URL of an object, the first element is the name of the scheme, separated from the rest of the object by a colon. The rest of the URL follows the colon in a format depending on the scheme. Internet protocols are then followed by **//**. In Jade, the Internet protocol can be one of the following values.

  - http

    Use the HTTP protocol (default)

  - https

    Use the secure HTTPS protocol (if the service is marked as secure)

  - tcp2

    Use the Jade-to-Jade protocol

- User name and password

  Optional user name, if required. The password, if present, follows the user name, separated from it by a colon (**:**). The user name and password are followed by an **@** symbol. The use of user name and passwords that are public is discouraged. You can set these values in the consumer, by setting the **username** and **password** properties (not applicable to **tcp2**).

- Domain name

  The Internet domain name of the host or the IP address.

- Port number

  If it is not the default number for the protocol (80 for HTTP, 443 for HTTPS, must be specified for **tcp2**), is specified after a colon.

- Path

  The rest of the locator is known as the *path*. It can define details of how the client should communicate with the server, including information to be passed transparently to the server without any processing by the client. The path is preceded by **/**. For example, a Jade web service path consists of a virtual directory, followed by **/jadehttp.dll?,** followed by the name of the web service application, followed by the service name, and optionally followed by an exposure list name (when using multiple exposures).

The full syntax of the web service URL is as follows.

```
<scheme>://<user-name>:password@<domain>:<port>/<virtual-directory>
/jadehttp.dll?<application-name>&serviceName=<service-name>& listName=<list-name>
```

In this syntax, the required entities are marked in bold. If an exposure list name is not specified, the first exposure in the list of exposures attached to the web service application is selected (to maintain backwards compatibility).

The following are some Jade URL examples.

```
http://wilbur/jade/jadehttp.dll?WilburWebService&serviceName=InventoryService

http://smith:smithpass@wilbur:5695/jade/jadehttp.dll?WilburWebService&
serviceName=InventoryService&listName=FredsInventory

tcp2://wilbur:5700/jade/jadehttp.dll?WilburService&serviceName=InventoryService
```

The URL to which the SOAP request is sent is set in the imported WSDL. It can be subsequently changed in the Jade Platform development environment on the **Web Services** sheet of the Define Classes dialog for the **JadeWebServiceConsumer** subclass.

It can also be changed in the **JadeWebServiceConsumer**::**setEndpointURL** method; for example, to set the end point for a regular web service:

```
setEndPointURL("http://myserver/jade/jadehttp.dll?ErewhonWebServiceApp&
serviceName=ErewhonInvestmentsServiceAdmin")
```

To set the end point for a Jade-to-Jade web service:

```
setEndPointURL("jadehttp.tcp2://myserver:8081/jade/jadehttp.dll?
ErewhonWebServiceAppJ2J&serviceName=ErewhonInvestmentsServiceAdmin")
```

You can also change the end point URL in the XML-based runtime configuration file, by setting the **endpoint** element. For details, see "endpoint element" in Chapter 3 of the *Web Application Guide*.

Jade-to-Jade Web Services

If the WSDL is imported from a Jade-to-Jade web service provider, the end point will reflect this.

You do not need to set or change anything in the consumer; Jade will automatically communicate using the Jade-to-Jade web service instead of the IIS or Apache web server.