IA81 WMQI and Web Services Version 1.0

31 December, 2002

Helen Wylie

helenw@uk.ibm.com

Take Note!

Before using this report be sure to read the general information under "Notices".

First Edition, December 2002

This edition applies to Version 1.0 of S*upportPac title* and to all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Notices

Trademarks and service marks

Acknowledgments

This SupportPac was produced during a Residency run by the Product Introduction Centre at Hursley. The Residency team included

Chris Brown
Andy Humphreys
John Jones
Paul Pollard
Shahryar Sedghi
Helen Wylie

Summary of Amendments

Date	Changes
31 December 2002	Initial release

Preface

This Support Pac examines the role that WebSphere MQ Integrator can play in a Web Services Environment. The SupportPac identifies components which are required for WMQI to operate in a web services context and provides new components in two of these areas.
The SupportPac provides subflows and Plug in node which together manage the SOAP aspects of incoming, outgoing and fault messages. These subflows enable WMQI developers to provide a Web Service interface to any functionality which can be invoked through a WMQI message flow and to invoke Web Services; these developers need to consider only the data required by the service not the SOAP components.
This SupportPac also provides a JMS transport for Axis. This enables SOAP requestors communicating over http to access the WMQI Web Services by transforming incoming SOAP requests over http to SOAP requests over JMS and forwarding these to the WMQI message flow. This transport also receives responses from WMQI over JMS and returns these to the requestor over JMS.

Bibliography

Web Services Essentials, Cerami, Ethan, O'Reilly & Associates Inc., 2002, ISBN: 0-596-00224-6

XML Pocket Reference, Eckstein, Robert with Michael Casabianca, O'Reilly & Associates Inc., 2001, ISBN: 0-596-00133-9

AXIS: The Next Generation of Java SOAP, Irani, Romin and S. Jeelani Basha, Wrox Press Ltd, 2002. ISBN: 1-861007-15-9

Understanding Web Services, Newcomer, Eric, Addison-Wesley, 2002, ISBN: 0-201-75081-3

Web Links

Eclipse Organisation Eclipse is an open platform for tool integration built by an open community of tool providers. Operating under a open source paradigm, with a common public license that provides royalty free source code and world wide redistribution rights, the eclipse platform provides tool developers with ultimate flexibility and control over their software technology.: http://www.eclipse.org
RosettaNet A self-funded, non-profit organization, RosettaNet is a consortium of major Information Technology, Electronic Components and Semiconductor Manufacturing companies working to create and implement industry-wide, open e-business process standards. These standards form a common e-business language, aligning processes between supply chain partners on a global basis. RosettaNet is a subsidiary of the Uniform Code Council (UCC) (http://www.uc-council.net/): http://www.rosettanet.org
The World Wide Web Consortium (W3C) (http://www.w3.org/) develops interoperable technologies (specifications, guidelines, software, and tools) to lead the Web to its full potential. W3C is a forum for information, commerce, communication, and collective understanding.

- SOAP 1.1: http://www.w3.org/TR/SOAP
- SOAP 1.2 references: http://www.w3.org/2000/xp/Group/ contains various pointers to SOAP 1.2 documents, including
- SOAP 1.2 Primer: http://www.w3.org/TR/soap12-part0
- WSDL 1.1: http://www.w3.org/TR/WSDL.html

OASIS (Organization for the Advancement of Structured Information Standards). OASIS is a not-for-profit, global consortium that drives the development, convergence and adoption of e-business standards. Members themselves set the OASIS technical agenda, using a lightweight, open process expressly designed to promote industry consensus and unite disparate efforts. OASIS produces worldwide standards for security, Web services, XML conformance, business transactions, electronic publishing, topic maps and interoperability within and between marketplaces. The OASIS Network includes UDDI, CGM Open, LegalXML and PKI.): http://www.oasis-open.org
UDDI: http://www.uddi.org
XML-RPC.Com: http://xml-rpc.com/
UCC The mission of the Uniform Code Council, Inc., is to take a global leadership role in establishing and promoting multi-industry standards for product identification and related electronic communication. The goal is to enhance supply chain management thus contributing added value to the customer. http://www.uc-council.net/

1 Introduction

For some years organizations have been offering, and using, business services over the Internet. These services range from relatively simple request/response services such as credit inquiries or package tracking, to full business-to-business interactions such as those defined for RosettaNet (http://www.rosettanet.org). In general, these have been implemented in a relatively ad hoc manner, with the service provider or industry consortium defining the protocols (e.g. HTTP GET/POST) and parameters (e.g. name/value pairs or XML data) required to use the service.

The implementation of an emerging set of standards for advertising and accessing business operations and activities is primarily, but not exclusively, aimed at an internet environment. Typically Web Services use an underlying messaging paradigm to facilitate communication between distinct business components. The role of WebSphere MQ Integrator in the Web Services world is primarily that of an intermediary providing a Web Services interfaces to downstream applications rather than as a provider of end-point implementations of Web Services.

This SupportPac describes the role WebSphere MQ Integrator (WMQI) can play in a Web Services environment. The contents include:

- examples of how Web Services may be used with WMQI;
- sample scenarios;
- a cookbook explaining how such scenarios are implemented;
- supporting assets in the form of message sets, message flows, custom nodes, and test messages.

1.1.1 XML-RPC

XML-RPC was an early attempt at providing a standard protocol to allow computers to make function calls over the web. Details can be found at http://xml-rpc.com/ and many of the ideas have been taken into SOAP.

1.2 Web Services

The term ‘SOAP’ is generically used to cover several emerging standards: SOAP itself; UDDI (Universal Description, Discovery and Integration); and WSDL (Web Services Description Language). These standards are intended to make Web Services much easier to use, especially in an application to application environment. The following definitions are gathered from organisations such as W3 and OASIS:

1.2.1 SOAP

SOAP is a lightweight protocol for the exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it; a set of encoding rules for expressing instances of application-defined data types; and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols.

1.2.2 UDDI

UDDI is an XML-based registry for businesses worldwide to list themselves on the Internet. Its ultimate goal is to streamline online transactions by enabling companies to find one another on the Web and make their systems interoperable for e-commerce. UDDI is often compared to a telephone book's white, yellow, and green pages. The project allows businesses to list themselves by name, product, location, or the Web services they offer.

1.2.3 WSDL

WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate.

Assume an application needs to discover the location and details of a business service. It issues a request to search the registry using UDDI. The UDDI specification covers both the data model of the data held in the registry, as well as the API used to search it. The reply will be a WSDL message, describing various aspects of the service, including where it is, the protocol used to access it, how to invoke it, and the format of the request and reply. All data that flows will be in XML, in a format defined as part of the SOAP standard. The following diagram illustrates how the various parts might be used:

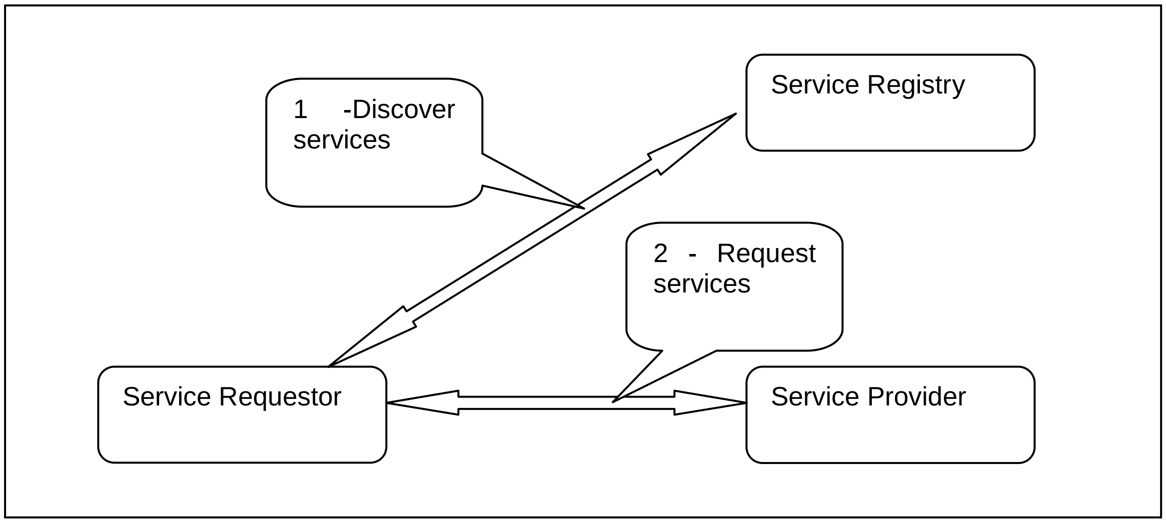


Figure 1 Web Services Context

During early implementations, this scenario is likely to be implemented within an organisation (Intranet rather than Internet). This would certainly be a very powerful method of publishing business services; for example during a merger, it could be an invaluable tool to ease the integration of the IT processes of the organisations involved.

1.3 Scope of SupportPac

This SupportPac concentrates primarily on WMQI support for SOAP1.1 and related WSDL aspects, with little or no mention of UDDI. This reflects the likely role of WMQI in a web services environment.

1.3.1 Examples

In this document we will look at four areas that can be supported through the use of SOAP with WMQI:

- The first is where a business wishes to offer a Web Service using SOAP and uses WMQI to provide a Web Service interface fronting other capabilities, in particular functionality in systems which have already been integrated via WMQI or MQ.

- The second is where a function provided by an existing Web Service needs to be accessed by a WMQI message flow so that the function can be incorporated within the overall service offered by the message flow.

- The third is where applications are MQ enabled and expect to invoke services or request information from other applications using MQ requests. If some of these applications are available only as Web Services WMQI can provide access by processing MQ requests, creating SOAP requests from them and invoking the Web Service on behalf of the MQ client.

- The final area is where the WMQI message flow acts as an intermediary between a SOAP sender and a SOAP receiver. From a purely technical point of view this can be seen as a combination of other scenarios with WMQI acting as both SOAP provider and SOAP Client, however there are some additional considerations when, WMQI is acting as a SOAP intermediate node.

In order to meet these requirements, there are a number of facilities that are needed within WMQI, which are the main technical requirements addressed in this SupportPac.

1.4 Technical Requirements

In order for a WMQI message flow to interface to SOAP services, whether as a provider or a requestor, there are a number of technical requirements to be met. These are investigated as part of this SupportPac. First those required for the actual running of the message flow:

- The ability to build SOAP messages
- The ability to understand (parse) SOAP messages
- The ability to detect errors and to create appropriate SOAP faults.
- The ability to send SOAP messages and receive a reply
- The ability for a WMQI message flow to be driven by a SOAP message.

Other requirements are those which arise primarily at build time:

- Importing schemas from WSDL definitions into the WMQI MRM
- Exporting WMQI MRM definitions to build WSDL.

2 Business Scenarios

2.1 Extension of existing EAI solution

This first scenario assumes that an enterprise has already invested significantly in EAI based on an MQ / WMQI infrastructure. Many of these companies have already adopted a service oriented architecture where clients access services using MQ or JMS to submit requests to the services provided by via a WMQI messaging hub.

The services provided are frequently based on legacy systems, which have either been MQ enabled or have existing adapters or gateways. The formats required by the legacy service providers may be XML but are more likely to include a number of other custom wire formats. The work of modelling the formats required by the services will have already been carried out and defined in the MRM.

Message formats from the clients requesting services are more likely to be in XML and again these have either been defined in the MRM or are parsed using the generic XML parser.

The mediation carried out by the broker between client requests and service invocation can include any of the functionality provided by the broker but this is not of significance to this scenario. The significant factor is the accessibility of the legacy systems through WMQI. A number of companies in this position considered Web Services but did not feel that the approach was mature enough at the time. They are now looking again and regard the use of Web Services as a way of easily extending access to these services to a wider community within their own enterprise.

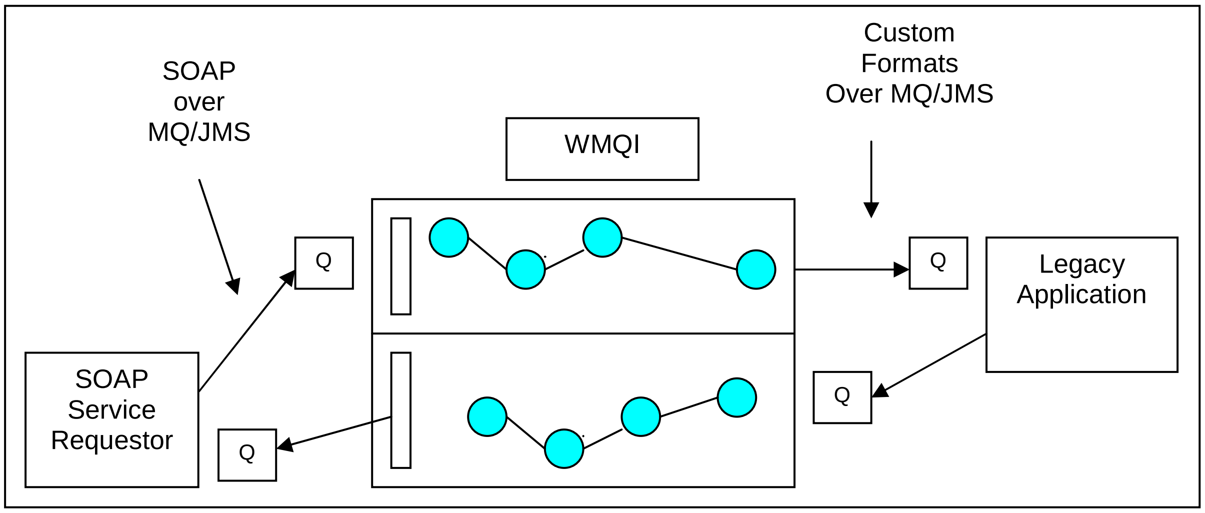


Figure 2 SOAP to Legacy

As shown in Figure 2 above the introduction of a Web Services interface in WMQI allows WMQI to provide a Web Services interface giving extended access to SOAP requesters. In this context we can eliminate some of the complexities inherent in a more general Web Services architecture and should strive to avoid additional infrastructure.

We can assume that there will be no need for dynamic look up of services, i.e. no private or public UDDI will need to be accessed by the WMQI hub when processing a request. In this initial scenario the published Web Services, implemented as WMQI message flows, may conceal use of alternative back end solutions depending on message content but WMQI will control this using typical WMQI routing capabilities and will not invoke UDDI.

It is reasonable to assume in this scenario that all communications can be carried over MQ/JMS and that all clients can make Web Services requests over MQ/JMS. In practice clients may invoke services directly over MQ/JMS or over http with conversion to JMS through a gateway. Clients using MQ or JMS as the communication protocol may use encapsulation within http to pass across the internet and through firewalls.

A significant feature of this scenario is that a single enterprise has control of the Web services clients, the transport, the WMQI implementation and the back end services. Standards can be applied to ensure interoperability of clients and service providers.

Let us consider a couple of examples where this might reasonably operate. Any customer oriented organisation which has and continues to be involved in a series of acquisitions and mergers may use a WMQI hub to aggregate customer information retrieved from a number of applications each holding the customer data for previously separate organisations. Requests can be supported using MQ formats but the company wishes to move to Web Services interfaces through out as they see this as offering benefits in a dynamically changing business environment.

Alternatively we might have a company which has an MQ infrastructure as a reliable communications infrastructure serving its current integration needs across many geographies. Many applications are being added to the infrastructure and the enterprise decides that there is a need to standardise on service interfaces. In particular they are keen to support the use of WSDL and UDDI as a means of advertising and cataloguing the growing number of service oriented offerings within the company. Here WMQI would support the standards based interface definitions.

2.2 Extension of EAI Solution outside an Enterprise

In business terms the second scenario starts from the same point as the first, that is an existing EAI solution based on MQ / WMQI, which already has well defined service offerings.

However the Enterprise wishes to extend access to the service offerings, based on Web Services standards, to service clients over which they have no control in terms of infrastructure. In particular they wish to offer access to their services without requiring clients to have MQ infrastructure and therefore intend to introduce http as the client transport for requesting SOAP services. The great majority of potential Web Services clients will have http (or https) access and they view this as the most widely available transport.

In the most general case Web Services clients will be making requests from outside the Enterprise, these clients may be partners, or customers to whom self service operations are being extended. In many cases the requests will be made across the internet and security becomes an issue as consideration has to be given to the protocols passing through firewalls. We therefore require capability within a DMZ to receive http requests and transmit them to WMQI. WMQI does not currently have an http listener and security constraints would not normally allow it to operate within the DMZ. This scenario therefore assumes that there will be a component offering protocol conversion in the DMZ. This component will listen for http requests and will transmit the requests to the relevant WMQI queue over JMS as shown in Figure 3 below.

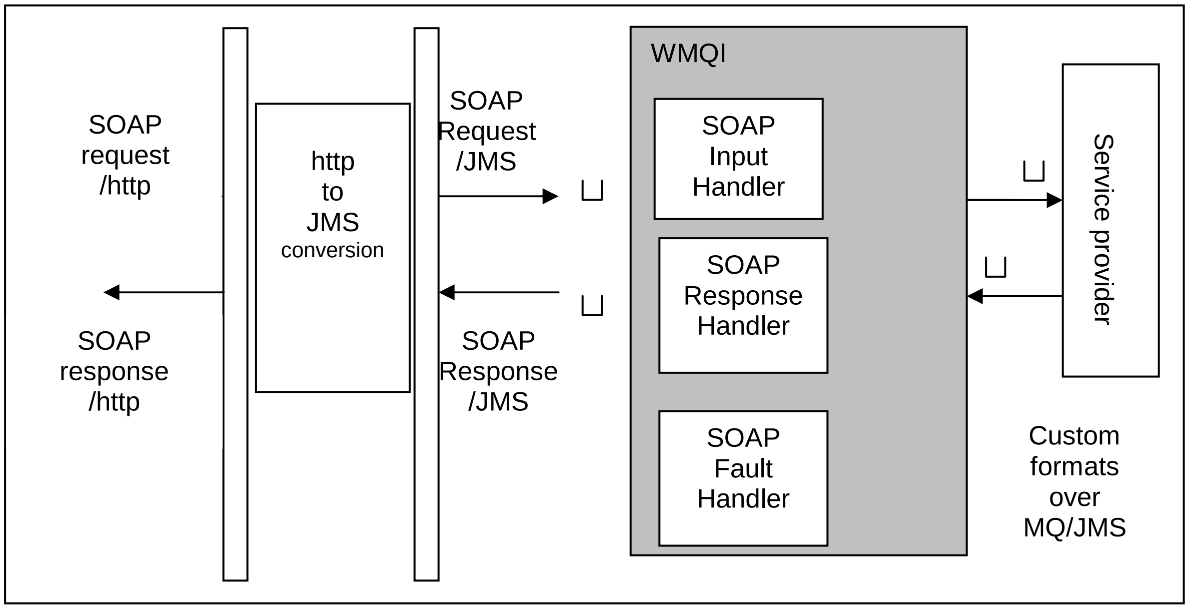


Figure 3 - http transport inside inner firewall

The capabilities required within WMQI are almost identical to those of the first scenario. However in the first scenario we gave no consideration to the service types to be supported. In this scenario the introduction of a transport without the asynchronous guaranteed delivery of MQ means that attention must be given to requests which invoke any legacy service which implements a transaction causing change to enterprise data. The legacy services may not have 24 * 7 availability or may not be able to handle peak loads in a timely manner.

The introduction of Web Services in WMQI which implement a “Fire and Forget” approach addresses this issue. Such a Web Service can accept a request for a transaction and after necessary manipulation place a non SOAP persistent message on a queue for the legacy application implementing the transaction. The WMQI service then generates an immediate SOAP response to the requestor that the request has been submitted.

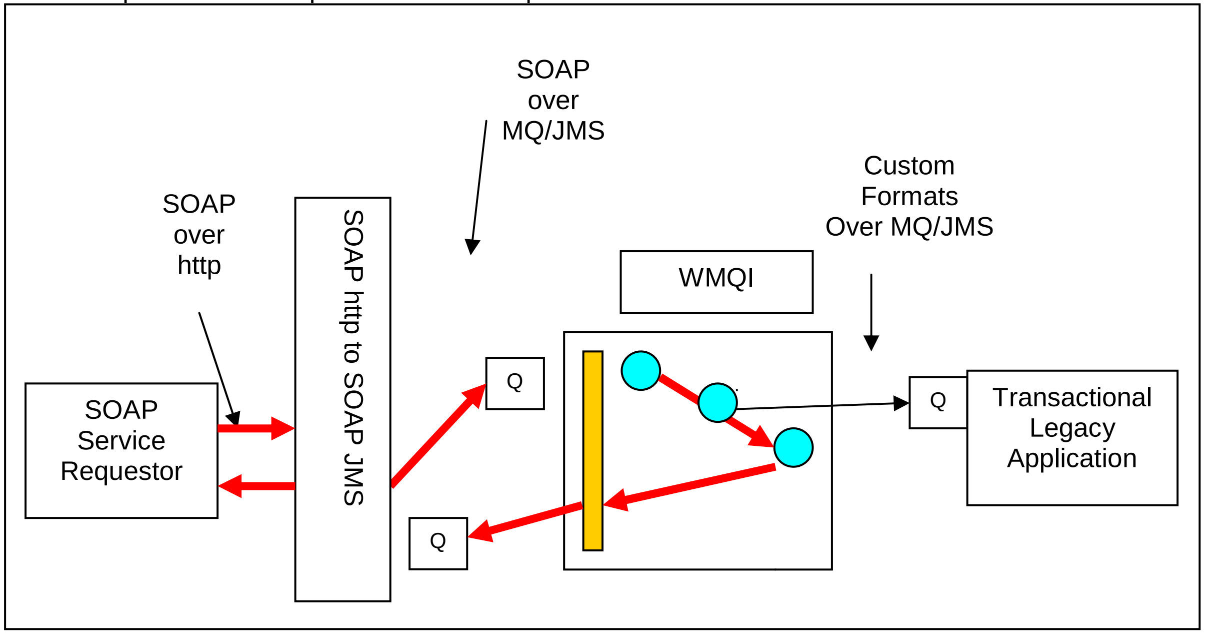


Figure 4 SOAP to Legacy, Fire and Forget

Such a service request, often termed a “Fire and Forget” request, would be very typical in an EAI environment where a loosely coupled, asynchronous approach has been taken to integration and back end applications provide the necessary guarantees to process the transactional requests sent to them. Requesters of “Fire and Forget” services do not normally require information returned after the transaction but they do need to know that the request has been securely passed to the service. In an MQ environment the messaging system assures this once the request has been submitted and acknowledged.

A second type of service request is a Request / Reply, for which the message flow must process a response from the service into a SOAP response. This type of service is non-transactional and timeouts must be supported both within the service and at the client.

These two service types, when combined to a “Fire and Forget” request followed by repeatable “Request/Reply” requests will support the integration requirements of many organisations. It is dependent upon the ability of the back end systems to process requests with the same degree of reliability as the message delivery. However many enterprises dealing with transactions of significant value do have such reliability in their legacy services; in many cases that is why the legacy services are still there supporting the business.

The general concept of services with http (synchronous) to messaging (Asynchronous) protocol conversion applies in very many contexts and is likely to be the most frequent use of WMQI in the Web Services context.

2.3 Access to Web Services from WMQI Message Flows

The first two scenarios look at WMQI as a provider of Web Services. In this scenario we look at the case where a WMQI message flow needs to access a Web Service as part of its business logic. This may occur in any WMQI message flow not only for those which are themselves providers of Web Services.

As an enterprise moves towards the adoption of Web Services as a standard for internal integration there should come a point where all new services must be implemented with a Web Services interface and existing services are either being extended to offer a Web Service interface or being replaced by new web enabled service offerings. The introduction of WMQI to provide Web Service interfaces for many legacy systems would be a part of this process.

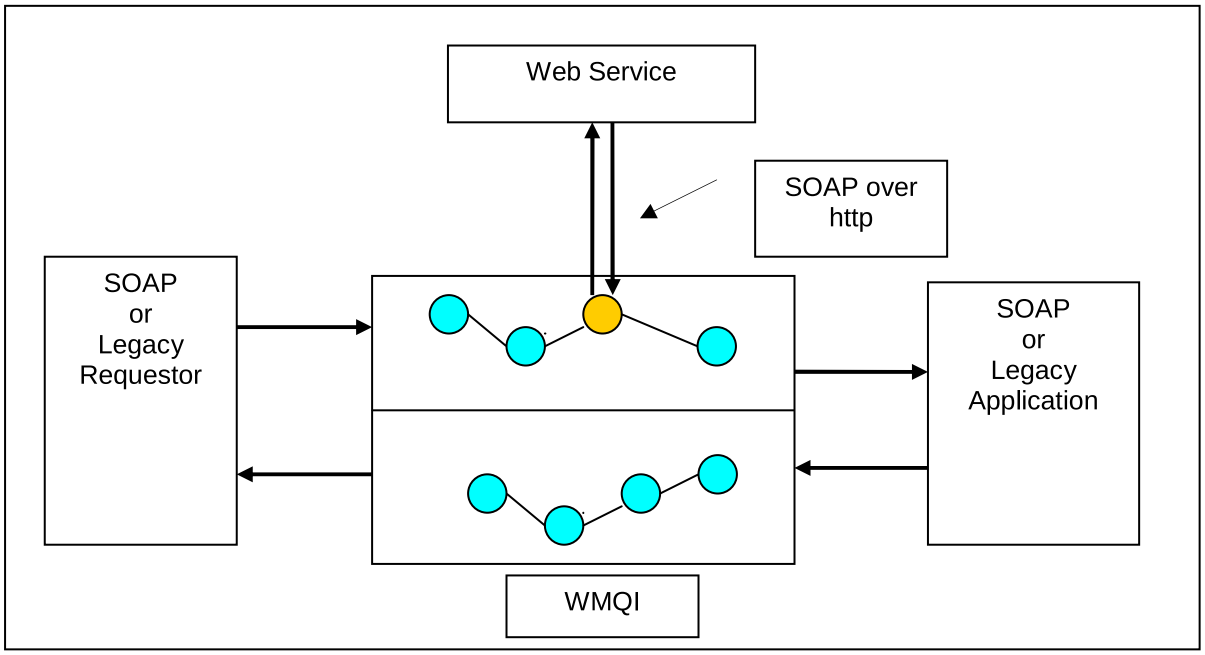


Figure 5 SOAP to Legacy

As the use of Web Services grows within an organisation it may become necessary for WMQI to invoke a Web Service as part of its business logic. Examples of the need for use of a Web Service might include:

- Calls to a logging service
- Calls to external services to provide authorisation / credit checks
- Calls to a service supporting a business process for registration
- Calls to retrieve additional information needed to enrich the incoming message

This scenario assumes that the message flow requires the information in its own processing logic and the request will therefore be made as a pseudo synchronous request / reply mode with timeout if the response is not returned within a given time. This will apply whether access to a Web Service is over http or JMS, both of which must be supported.

2.4 Providing Web Services Access to Legacy Clients

The previous scenario looked at how an organisation might implement Web Services from the viewpoint of the services. Let us now look at a similar migration towards Web Services from the perspective of the client. It is essential to consider how an enterprise can migrate to a Web Services approach by taking intermediate steps which avoid the inherent risks (if not total infeasibility) of a “big bang” switch to a new enterprise integration approach.

As part of this migration new Web Services will be introduced into the enterprise and new clients will use these via the intended SOAP interface. Also existing services will be replaced or given a new Web Services front end. Some existing clients may be easily modifiable to use new interfaces but this will not be true of all, some may be legacy applications or packages which have previously interacted with old service providers using MQ interfaces.

The challenge for WMQI in this scenario is to receive MQ messages from existing clients and to call the new services via their Web Services interface. The goal here is to enable the legacy interfaces to be removed and in many cases for the replacement Web Services interface to allow the easy deployment of new service implementations. WMQI is acting here, as it does in many situations, as a buffer point easing the transition from one deployment state to another.

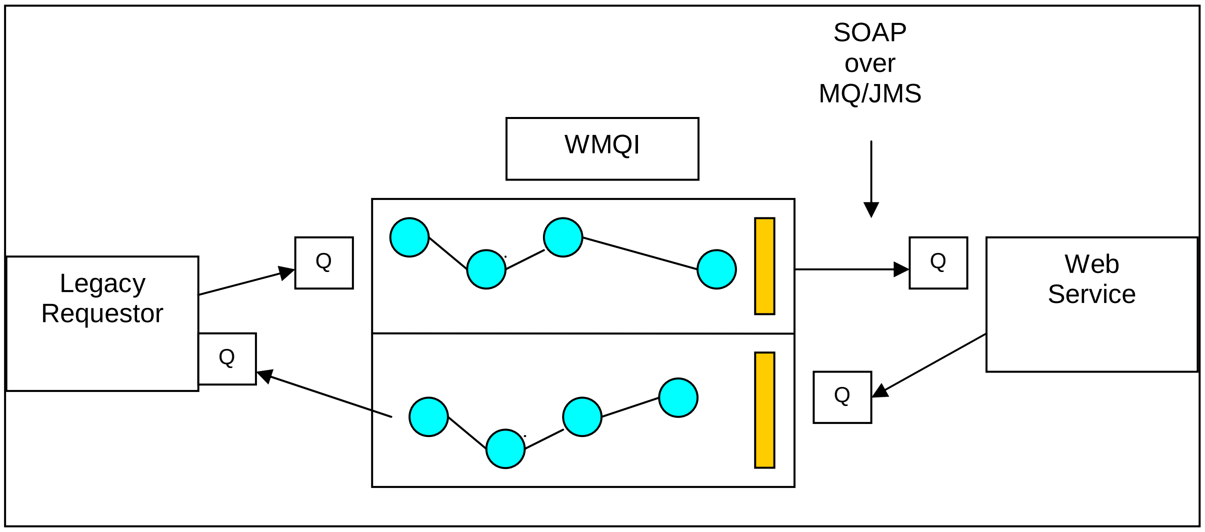


Figure 6 Legacy Clients accessing Web Services

A variation on this approach is one where a decision has been taken to use MQ and WMQI as the standard for integration on the client side. WMQI provides a buffer between clients and a changing set of systems providers, which may include both legacy services and Web Services. This is a likely scenario in organisations operating in an environment of acquisitions and mergers, providing aggregation of requests to a changing set of service providers, translation to the variety of formats required and a range of other facilities.

2.5 Service Intermediary between two Web Services

The final scenario combines all of the technical requirements of previous scenarios but supports a rather different business scenario. In this case there are both Web Services and Web Services clients operating outside the WMQI environment but WMQI acts as a SOAP intermediary between them, in much the same way as it can act as an intermediary between MQ enabled clients and services.

WMQI provides a point at which clients can be protected from change of service provider for given services and a point where a range of functions can be inserted. These may include basic transformation capability between two WSDL definitions, enrichment of the data in the request, and monitoring / data warehousing / auditing of messages or perhaps the introduction of a charging mechanism for services used.

Alternatively it might provide a point at which business decisions can be made on routing (based on message content such as value of transaction), quality of service provided, and location of services provided.

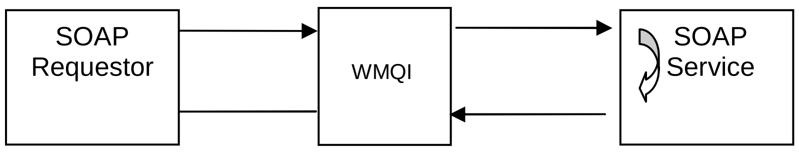


Figure 7 WMQI as a SOAP Intermediary

This scenario is very important in terms of what WMQI can offer in the Web Services world although it does not in itself introduce any new technical requirements. Previous scenarios have considered the input and output of Web Services requests and responses over various combination of http and JMS protocols covering operations within an enterprise and extending beyond it. WMQI as a services intermediary applies in all these cases.

3 WMQI Web Services Solution Architecture

3.1 Overview of Components

Figure 8 below shows all the components which would be involved in implementing the business scenarios described in the previous section. For some of these, for example the SOAP http Client, there are many existing solutions and we will do no more than describe the role in the overall picture. For others, such as the SOAP processing nodes within WMQI, there are new solutions provided in this SupportPac. These are described in detail in section 4, WMQI Component Details and examples of use given in section 6 Sample Implementations.

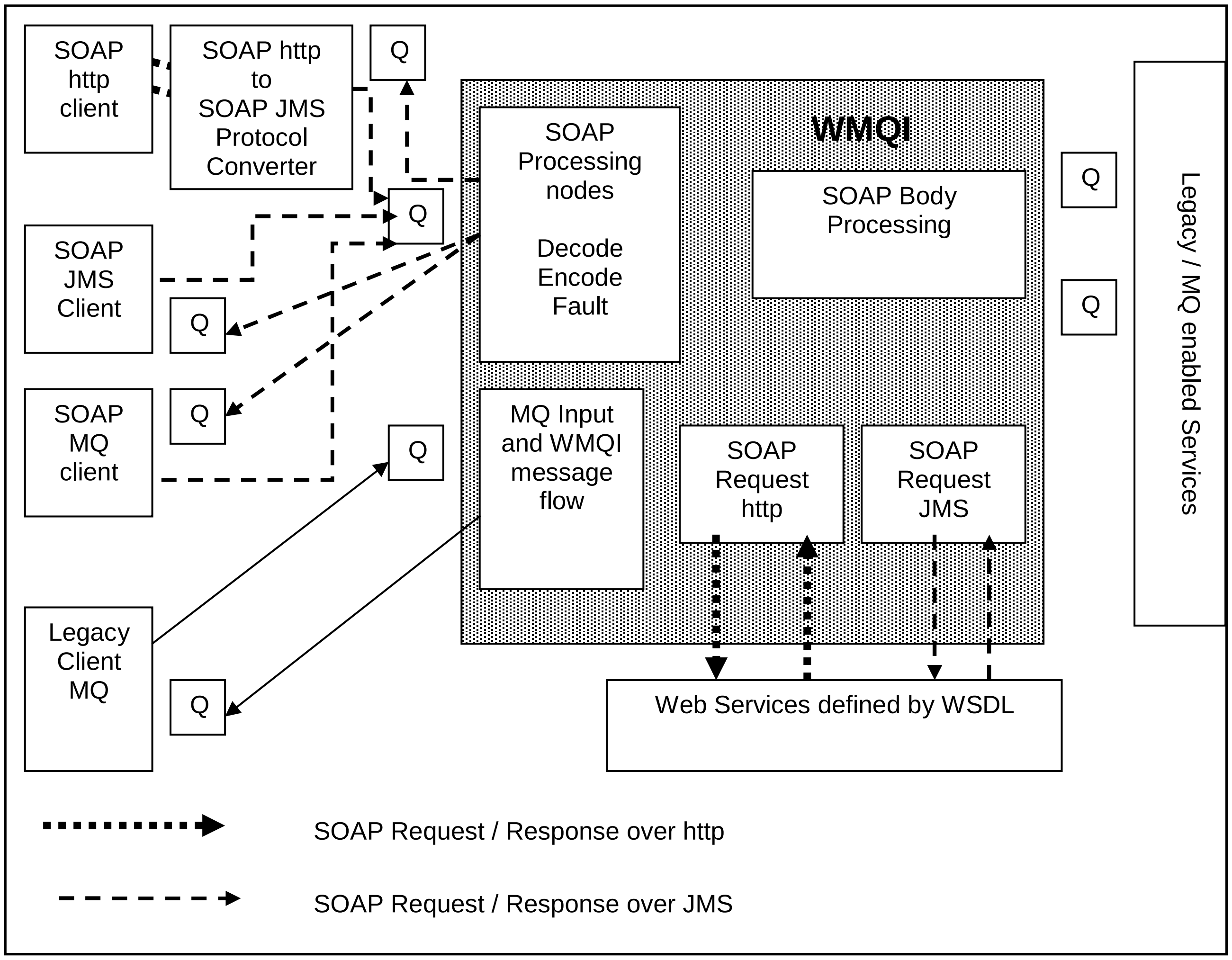

SOAP http client
SOAP http to SOAP JMS Protocol Converter
Q
WMQI
SOAP Processing nodes
Decode Encode Fault
SOAP Body Processing
SOAP JMS Client
SOAP MQ client
MQ Input and WMQI message flow
SOAP Request http
SOAP Request JMS
Legacy Client MQ
Web Services defined by WSDL
Legacy / MQ enabled Services
SOAP Request / Response over http
SOAP Request / Response over JMS

Figure 8 Overview of WMQI in a Web Services Context

Using Figure 8 we can now identify the main components of a Web Services solution.

3.1.1 Clients

First the clients which will despatch requests to the WMQI Server. We have identified four transports which may be used:

- SOAP over JMS
- SOAP over MQ
- SOAP over http
- Non SOAP requests for SOAP services over any supported WMQI protocol (including JMS, MQ, MQe or other custom built Input node)

The WMQI server handles directly all requests other than SOAP over http. Where a client generates SOAP messages over http, these are converted to SOAP over JMS by one of the recommended protocol conversion solutions.

The non SOAP requests are included in the overall architecture as these will be the means by which legacy clients can call Web Services via the WMQI which acts as a SOAP client on the clients behalf.

The client environment for generation of SOAP requests is not mandated by the WMQI Web Services solution.   This SupportPac does not address the Web Services clients directly and we recommend that those interested review the capabilities of various Web Services client environments.

However we do recommend that SupportPacs MA0R and MA7N be reviewed if there is a decision to use MQ messaging as the Web Services transport from the client. MA0R provides plug in a JMS plug in capability for the Axis servers while MA7N offers a plug in providing a SOAP MQ transport for Microsoft Visual Studio SOAP toolkit. Information on these and other SupportPacs are available at http://www-3.ibm.com/software/ts/mqseries/txppacs.

Also recommended would be WebSphere and its associated development environment, WebSphere Application developer Integration edition (WSADIE), which give full support for the development and run time environments of both SOAP servers and SOAP clients. Information on WSADIE is available at http://www.ibm.com/software/ad/studiointegration.

For an initial investigation, without an immediate requirement for serious implementations, we recommend a look at Web Services Toolkit V3.3. This addresses all aspects of Web Services, including client support, and is available at www.alphaworks.ibm.com/tech/webservicestoolkit.

3.1.2 WMQI as a Provider of SOAP services

The most significant element of the WMQI Web Services solution is the WMQI's ability to act as a provider of SOAP services, receiving and processing Web Service requests over JMS or MQ transport.

When WMQI acts as a provider of SOAP services it does so by providing message flows which are capable of processing a SOAP request received as a JMS or MQ message and invoking capabilities which are not normally accessible as a Web Service. A message flow is initiated when a message is read using the SOAPInputHandler, which includes the MQInput capability configured to use the generic XML WMQI parser. The SOAPInputHandler also includes SOAP decode capability which validates the incoming message and massages the information to give a message structure to be processed by the WMQI message flow and any down stream applications to which it passes request data. The services can be implemented by WMQI and associated down stream systems without any need to understand the SOAP envelope as the SOAP nodes will pass service data to the message flow .

When a call to a down stream application is made from within a message flow and a response returned, the response is processed by the SOAPOutputHandler to create a SOAP Response message which is then returned to the original SOAP requestor. When the down stream application is invoked over an asynchronous mechanism , for example MQ to a legacy system, it is necessary to maintain context about the original SOAP service request in order to return a SOAP response to the original requestor.

If errors occur during the processing of the message control is passed to a SOAPFaultHandler generates a SOAP fault message to return to the requestor.

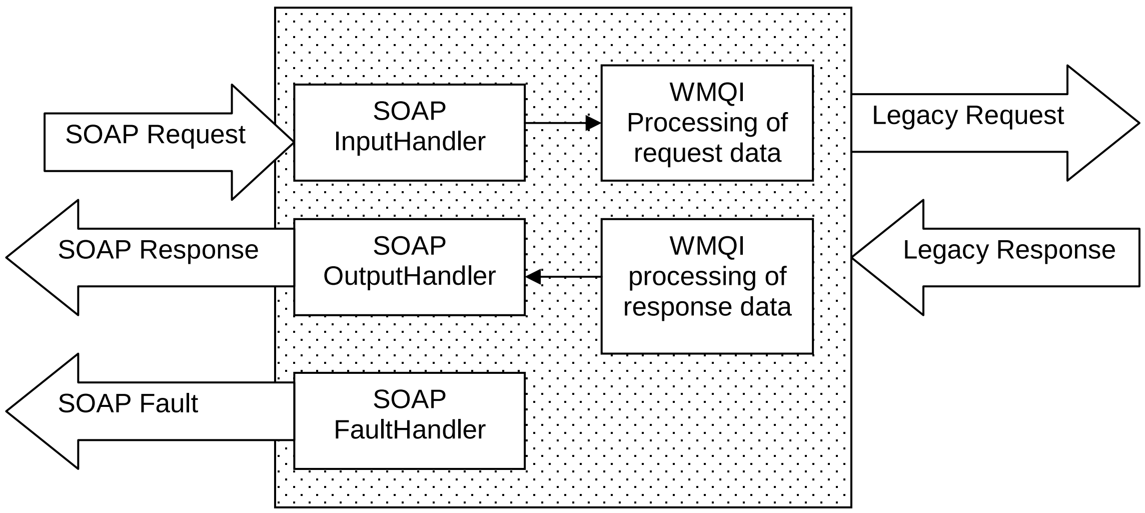


Figure 9 WMQI SOAP Handlers used to provide SOAP services

3.1.3 SOAP Body Processing

Once the incoming message has been processed by the SOAP decoding sub flow the service input data is available within the XML structure. Service data can be processed purely by ESQL but it is also possible to have an MRM definition of the message format to give additional capability. WMQI operations, such as translation to back end service formats, enrichment, monitoring are applied before invoking any legacy / MQ service which is fronted by the message flow. SOAP skills are not necessary to process the service data and the message flow developers will need to be aware only of certain data elements, such as header entries, which are extracted from the SOAP message and placed in the Environment tree and which they may need to process.

3.1.4 WMQI as a SOAP client

The overall WMQI Web Services solution provides support for WMQI to act as a requestor as well as provider.

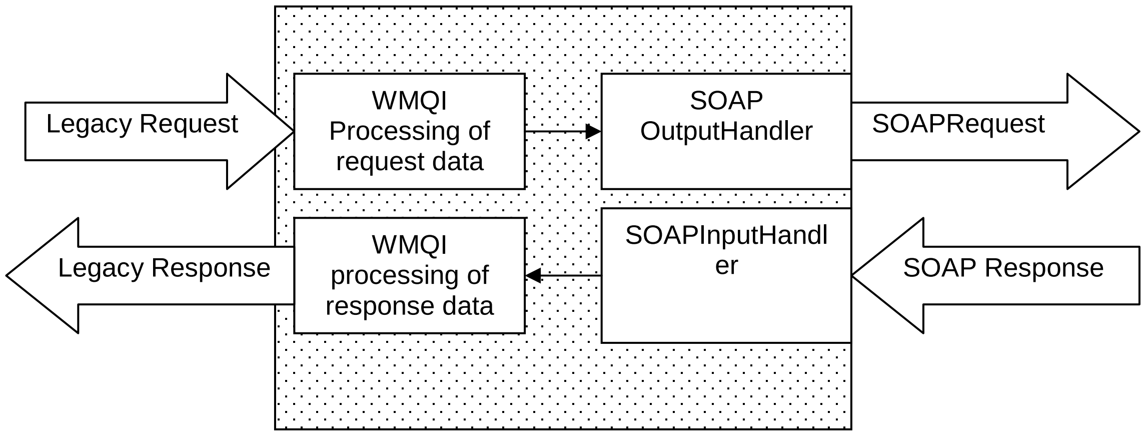


Figure 10 WMQI SOAPHandlers used to act as SOAP Client

WMQI could act as a SOAP client over various transports. We have looked at SOAP as a Web Services client over JMS and over http.

3.1.4.1 WMQI as a SOAP client over MQ / JMS

The SOAPOutputHandler, which generates responses to SOAP clients after service processing is complete, is also able to generate a SOAP request and dispatch this to a SOAP service. The response will be received asynchronously using an SOAPInputHandler and the message flow will be responsible for recovering any state, such as the Reply destination from the original client which started the request sequence.

When the SOAPInputHandler acts as a receiver of SOAP responses from a Web Services client it handles the additional case of the SOAP response returning a SOAP fault.

3.1.4.2 WMQI as a SOAP client over http

The total set of Web Services solutions for WMQI includes access by SOAP over http to Web Services outside WMQI. Because a synchronous protocol is being used the request can be sent and the response received within a single plug in node. The plug in would be configured with properties to determine the address of the Web Service and any necessary proxy information. It would also take a serialised SOAP Request which it would pass to the Web Service the service and receive a response.

Although the design for this plug in has been considered an implementation has not been included in this version of the SupportPac. An interim approach might be the consideration of the WSSOAPNode in SupportPac IA76.

3.1.5 WMQI as a SOAP Intermediary

The ability of WMQI to act as both SOAP Server and as SOAP Client allows it to act as a SOAP intermediary. This does not require additional capabilities in terms of request / response processing. The additional requirements are in terms of context handling and processing of header elements with attributes such as “mustUnderstand” and “actor” and these are built into the SOAPInput and Output Handlers.

3.1.6 Protocol Conversion

In order to make the WMQI Web Services available over http the WMQI solution set needs options for protocol conversion from http to JMS and back. Some alternative ways of accomplishing this are covered in section 5 Other Component Details.

3.1.7 Build Time and Data Format Management

Although not shown as a component in the overview diagram the management of data formats is likely to be of significance in some contexts. Data formats are required at a number of points within the WMQI Web Services solution. These include:

- WSDL to define the way in which a WMQI Web Service must be called. This includes a definition of the SOAP Body which holds the data to be processed by the service.
- MRM formats to define input formats to WMQI for legacy requesters of Web Services
- MRM formats to define the message body after SOAP decoding
- MRM formats to define the message body of SOAP messages created within WMQI for responses or requests to other Web Services
- MRM formats to define the messages for legacy service providers
- Published WSDL for requests coming in to the protocol translator over http and the related WSDL for the same request over JMS.
- Published WSDL for services which are called by WMQI

For any particular service request and response there will be multiple formats in use which are nevertheless related to each other. Neither the client environments for generation of requests nor the build time environment are mandated by the WMQI Web Services solution. WebSphere Application Developer Integration Edition (WSADIE) was used while developing the Support Pac for generation of schemas for Request messages and WSDL. The Schema Importer of WMQI is also a useful tool to carry the schema definitions into the MRM repository. In addition to these tools there are many others which can support the building of XML schemas, which can then be used to build MRM formats and WSDL.

3.2 External Components and Limitations

Web Services is based on open standards and a WMQI Web Services solution may be required to operate in an environment where there are components not tested within the WMQI solution set. These are shown in figure Figure 11 External Components below.

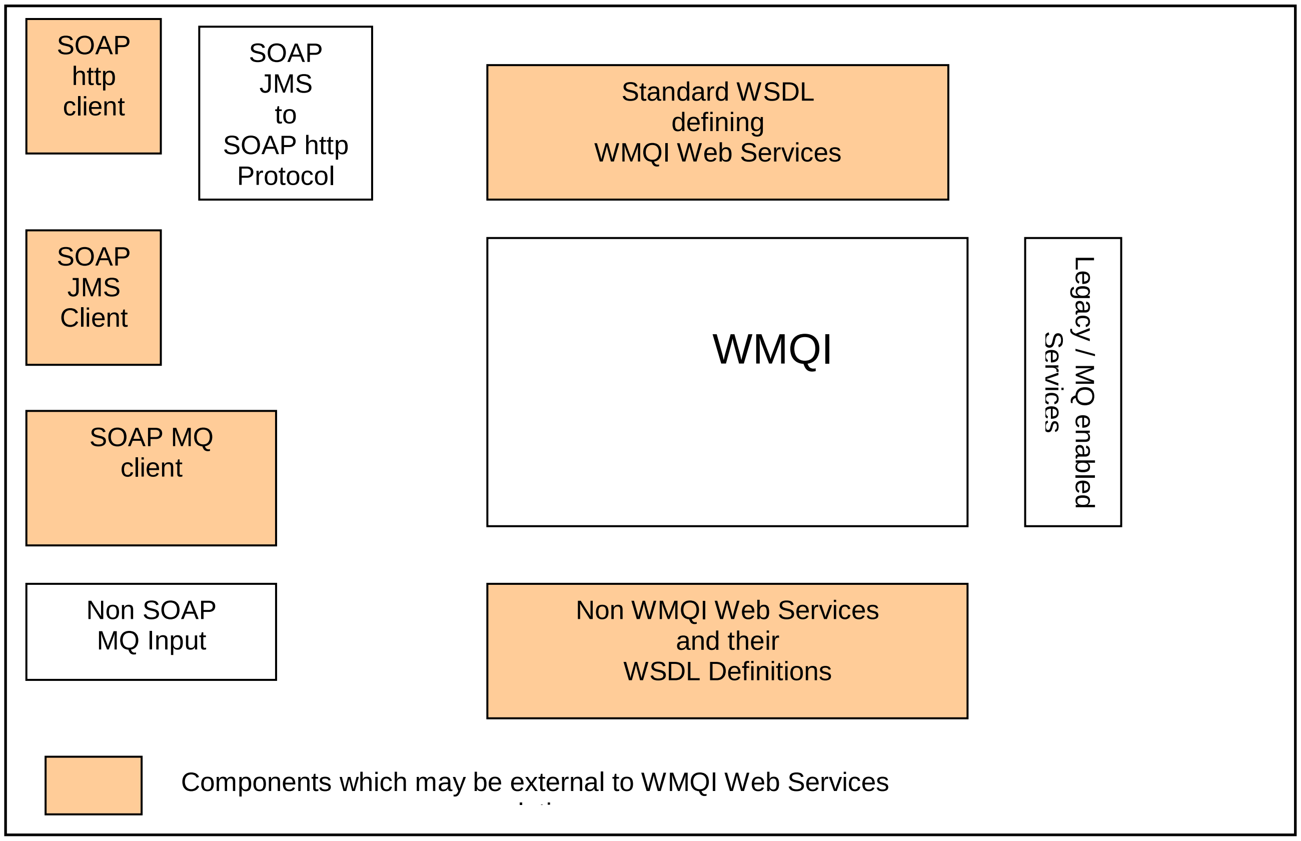

SOAP http client
SOAP JMS to SOAP http Protocol
Standard WSDL defining WMQI Web Services
SOAP JMS Client
WMQI
Legacy / MQ enabled Services
SOAP MQ client
Non SOAP MQ Input
Non WMQI Web Services and their WSDL Definitions
Components which may be external to WMQI Web Services

Figure 11 External Components

SOAP clients, provided they support the versions of the standards adopted by the WMQI Web Services solution, i.e. SOAP 1.1 and WSDL 1.1, will be able to make requests directly to WMQI over MQ or JMS or via the http to JMS protocol converter.

Web services invoked from WMQI message flows over http or JMS may be implemented in frameworks beyond those tested with the WMQI solution set. Again, provided that they conform to the standard it should be possible to understand the WSDL for such services, to invoke the service and to handle any responses returned.

The goal of this SupportPac is to support Web Services interactions which conform to SOAP 1.1. However there are some limitations. The main constraint is that the SOAP support in WMQI does not support multireference values. This can be avoided by using “literal” encoding rather than SOAP encoding or in many cases by configuration at the SOAP requestors to prevent the use multireference.

Full WSDL support is not included in the SOAP Handlers but a configuration plug in within the Handlers allows a number of elements from the WSDL to be defined and used to validate incoming SOAP messages.

There is also very limited support for namespaces in the body of the SOAP messages. All prefixes to element names are stripped before the message is passed into the Message flow for service processing. This makes the handling of data very much easier as parsing within WMQI would treat each unexpanded prefix as part of the element name.

The removal of namespace prefixes is based on the assumption that lack of namespace prefixes within the body will not lead to name conflicts within the message flow. Full and correct handling of namespaces requires changes to existing parsers within WMQI and will not be included before the next release.

In all cases the restrictions can be avoided in situations where there is control over the end to end flow, that is where SOAP clients and SOAP services lie within a single enterprise or where clients can be constrained to work within this limitations.

3.3 Future Directions

3.3.1 SupportPac Enhancements

Future enhancements planned for this SupportPac will include the plug in node to interact with other SOAP services over http and the extension of the JMS transport for Axis to support clients invoking Web Services over JMS.

Further examples on the use of tools to develop WSDL service definitions and MRM formats in parallel is also anticipated.

3.3.2 Product Enhancements

Future releases of WMQI are expected to include product enhancements that will provide improved support for the implementation of Web Services infrastructures and applications. This section is provided for guidance only to give an indication of the directions the product may take. IBM's plans are subject to review and change and this document does not represent a commitment by IBM to deliver these functions.

There are two main areas of focus for currently planned extensions to the product. The first of these areas is the addition of integrated support for http protocol connection to/from WMQI. For inbound connections an embedded http service would direct requests directly to message flows that utilise a new httpInput node. The httpInput node would be used in place of an MQInput node and would be configured with a uri selector to indicate which requests it wished to process. A companion httpReply node will provide the capability for a message flow to provide a (synchronous) response back to the originating request (as required by the http protocol). This may simply be an acknowledgement that the request has been received and is being held (or has been forwarded) for further processing or it may be a response message generated on completion of the processing required for this request. For outbound connections a new httpRequest node will be provided. This node will build and execute an http request using a serialization of all or part (a specified sub-tree) of the message tree passed to it. The body of the response (if any) will be used to build the propagated message tree. In order to provide a reasonable representation of the http "messages" in WMQI a new set of parsers will also be provided. This will consist of new root parser that can correctly manage the message sub-trees associated with an http message and a parser which owns the http header of a request (inbound or outbound) or response.

The second major area of enhancement focuses on extensions to the message modelling and parsing facilities, specifically in the area of XML Schema related constructs and concepts.   One key feature here will be to provide support for XML Namespaces, both in the MRM and in ESQL. This will simplify some of the work required to handle the namespace usage in SOAP messages. Namespace prefixes will be handled directly by the parser and will not be directly visible in the parsed message tree. Direct support for additional XML rendering options such as the ability to automatically recognize or generate xsi:type attributes is also being investigated. Tools enhancements would include the ability to “annotate” an MRM message model by indicating that particular message formats are to be grouped as parameters for a set of WSDL portType definitions and then using this information to generate WSDL files from the message model.

4 WMQI Component Details

4.1 WMQI as a Provider of SOAP services

4.1.1 Requirements

In order for WMQI to act as a provider of SOAP services it must be capable of accepting and processing a SOAP request over JMS where the request conforms to the SOAP 1.1 standard and to the published WSDL definition of the service offered.

It is beyond the scope of this SupportPac to enable WMQI to be able to receive SOAP requests directly over http.

It is beyond the scope of this SupportPac to be able to provide a single message flow which acts as a SOAP router for a number of different services. It takes the approach that a single that a single message flow should support requests for a given service including requests for a number of different operations where that service expects RPC style requests.

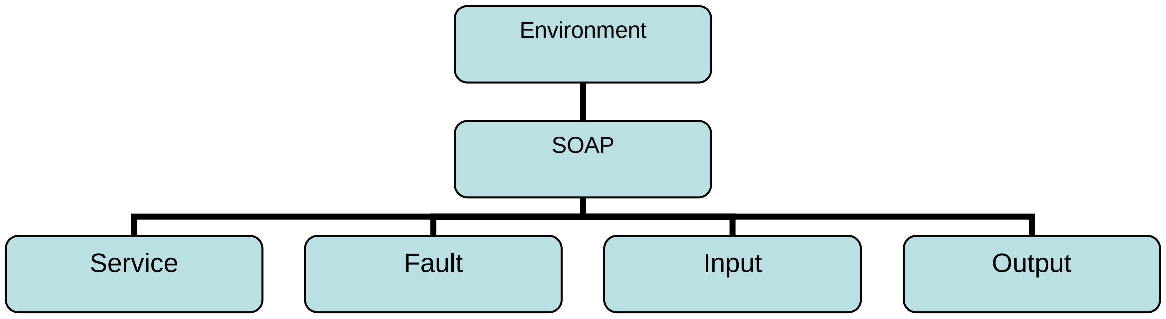
For any given message flow which is a provider of a Web Service, WSDL will be generated to define the service and publish it to interested clients. Each such Web Service will require a message flow to process the SOAP requests for the service, and to generate SOAP faults or SOAP responses as required.

In order to achieve this the SupportPac includes three nodes:

SOAPInputHandler
SOAP Response Handler
SOAP Fault Handler

4.2 SOAP data in the WMQI Environment

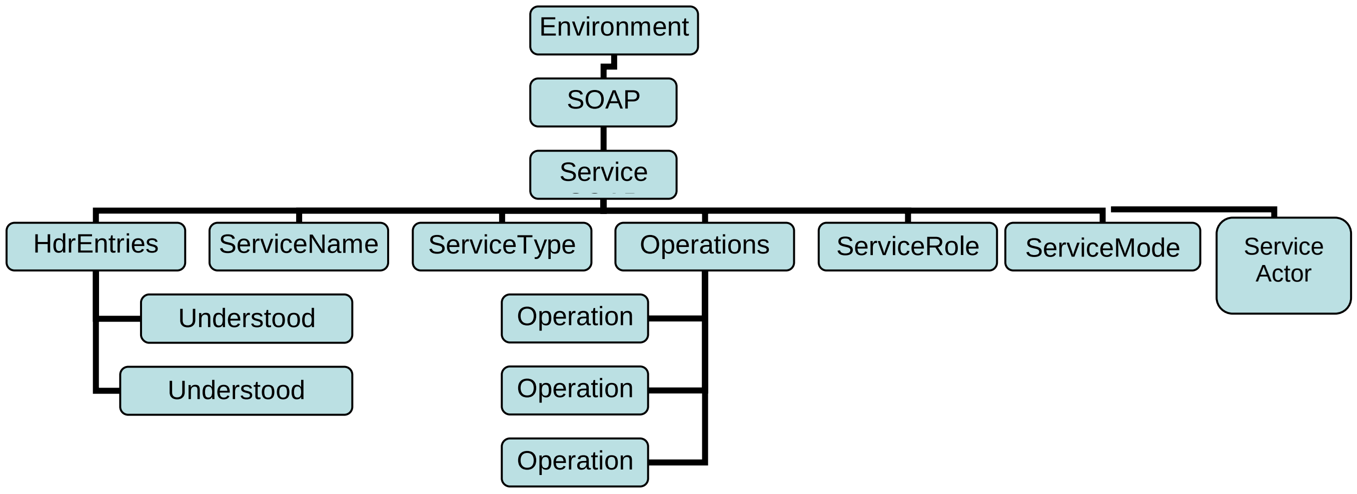
The SOAP nodes communicate using information in a SOAP subtree of the global Environment tree. The configuration properties of the SOAPInputHandler are used to provide the service definition held in the environment and each of the SOAP nodes sets data for each message instance as described in the following sections. The SOAP subtree has four top level entries



4.2.1 Service Sub tree

The Service sub tree within Environment.SOAP holds configuration parameters defined in the Service Definition PROPERTY tab of a SOAP Input or Output Handler. All these properties are required for SOAPInputHandler and a subset for the Output Handler.

The properties define relevant characteristics of the service implemented by the message flow. They combine elements which have been published in the WSDL for the service(for example service name and type) and elements which characterise the implementation of the service (for example headers understood by the message flow)



Environment.SOAP.Service.HdrEntries[].Understood - defines all the valid header entry names which the message flow understands.

Environment.SOAP.Service.ServiceName - is set to the name of the service and is used to identify the service when SOAP faults are generated.

Environment.SOAP.Service.ServiceType - is set to either 'RPC' or 'DOC' depending on the expected interaction style within the SOAP Body

Environment.SOAP.Service.Operations.Operation - this is used for RPC style services and must define all the valid operations which will be supported for inbound SOAP response or request messages (excluding Fault)

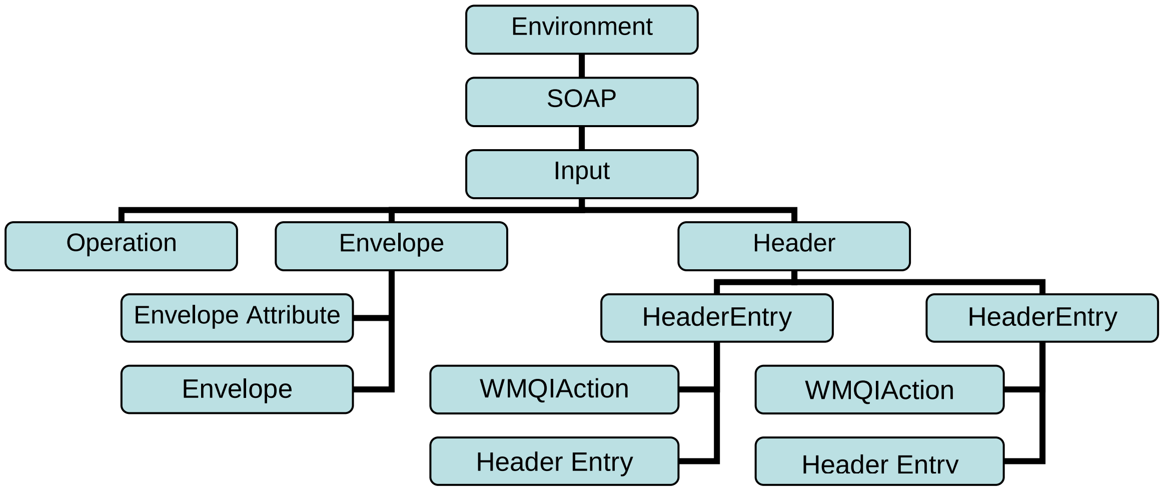
Environment.SOAP.Service.ServiceRole - should be set to the EndPoint or Intermediary to indicate, this is used in determining which headers should be processed.

Environment.SOAP.Service.ServiceMode - must be set to either 'Request' or 'Response' depending on whether the message is a request or a response.

Environment.SOAP.Service.ServiceActor – set to a fully qualified URI to identify the WMQI message flow as a SOAP server. This is used for determination of which headers to process when an actor is given and for definition of the FaultActor when the message flow detects an error.

4.2.2 Input Sub tree

The Input sub tree is generated from SOAP information in an incoming request; it does not hold the service data which will be processed within the message flow but rather subsidiary information which may be required to control processing.



Environment.SOAP.Input.Operation - contains the operation found within the SOAP Body where the incoming request or response is RPC style and a valid operation supported by the service has been found. This is essential to determine the processing path to be taken by the Message Flow.

Environment.SOAP.Input.Envelope contains envelope attributes defining the name space definitions used in the SOAP envelope.

Environment.SOAP.Input.Header contains any header entries found in the incoming message. All header entries are classified as

- those which the message flow must process
- those which the message flow may process
- those which must be forwarded if the message flow is acting as an intermediary.

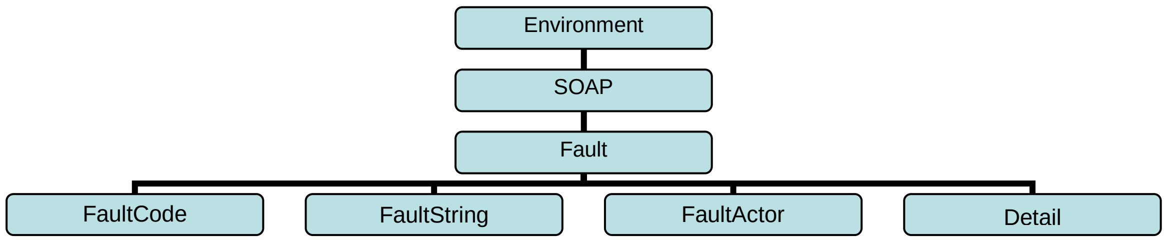
Each header entry consists of an XML structure which is a copy of the header entry in the incoming request. One additional attribute is added to each header entry, this is the WMQIAction which takes one of the values “MustProcess”, “MayProcess” or “Forward”

Headers contained in the MustProcess sub tree will only be those which have been declared in the service definition properties as being understood by the message flow. Any header with a mustUnderstand for the message flow is either in this list or will have already caused a SOAP fault earlier in the SOAPInputHandler.

4.2.3 Fault Sub tree

When a SOAP fault is detected entries in the Fault sub tree must be correctly defined and control passed to the SOAP Fault node which uses the information to send a correctly formatted SOAP fault message to the requester.

SOAP Input and Output Handlers define these entries when errors are found in requests or responses. The service processing within the message flow may also set these entries if they detect a fault while processing the service data.

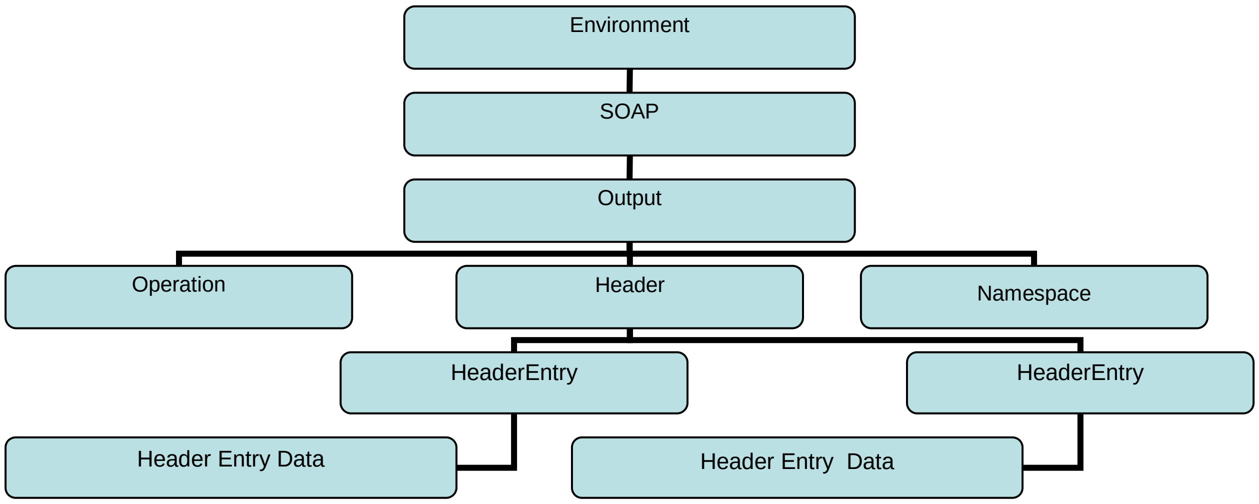


Environment.SOAP.Fault.FaultCode	- must be a valid SOAP 1.1 fault code
Environment.SOAP.Fault.FaultString	- is set to an appropriate error message
Environment.SOAP.Fault.FaultActor	- is either the SOAPActor from the Service Configuration or the SOAP Actor returned in a Fault element in a response message
Environment.SOAP.Fault.Detail	- is created by the SOAP Fault node for service errors

4.2.4 Output Tree

The output tree is used to set up information needed for the SOAPOutputHandler. This is almost a mirror image of the Input sub tree. However it is the responsibility of the Message Flow to build this tree before calling the SOAP Output Handler to send a request or response.

Header entries will be defined as for the Input Tree but no WMQI action will be present as it is assumed that all header entries are to be forwarded. The message flow must ensure that only header entries intended for the outgoing message are included.



The output subtree holds information which is used by a SOAPOutputHandler to build some of the SOAP components.

Output.Operation	Specifies the operation name which will be used for an outgoing RPC message
Output.Header	contains a header which will be added as first child of the SOAP envelope
Output.Namespace	Namespace used to qualify the operation.

4.3 SOAPInputHandler

4.3.1 Overview

The function of the SOAPInputHandler is to parse and validate an incoming SOAP message, which may be a SOAP request messages, a SOAP response message or a SOAP Fault message. The small differences in validation are handled within the SOAPInputHandler based on the configuration which indicates which is expected.

The SOAPInputHandler contains an embedded MQInput node to read messages from a queue. For a successfully validated message it propagates the service request data to the XML message tree with all namespace prefixes stripped. This data will include the parameters for the operation or the XML document depending on request style. Other information is added to the Environment, see previous section.

The validation of a message within the SOAPInputHandler checks the SOAP structure and ensures that all required SOAP components of the message are present and that all SOAP components of the message are in the correct position.

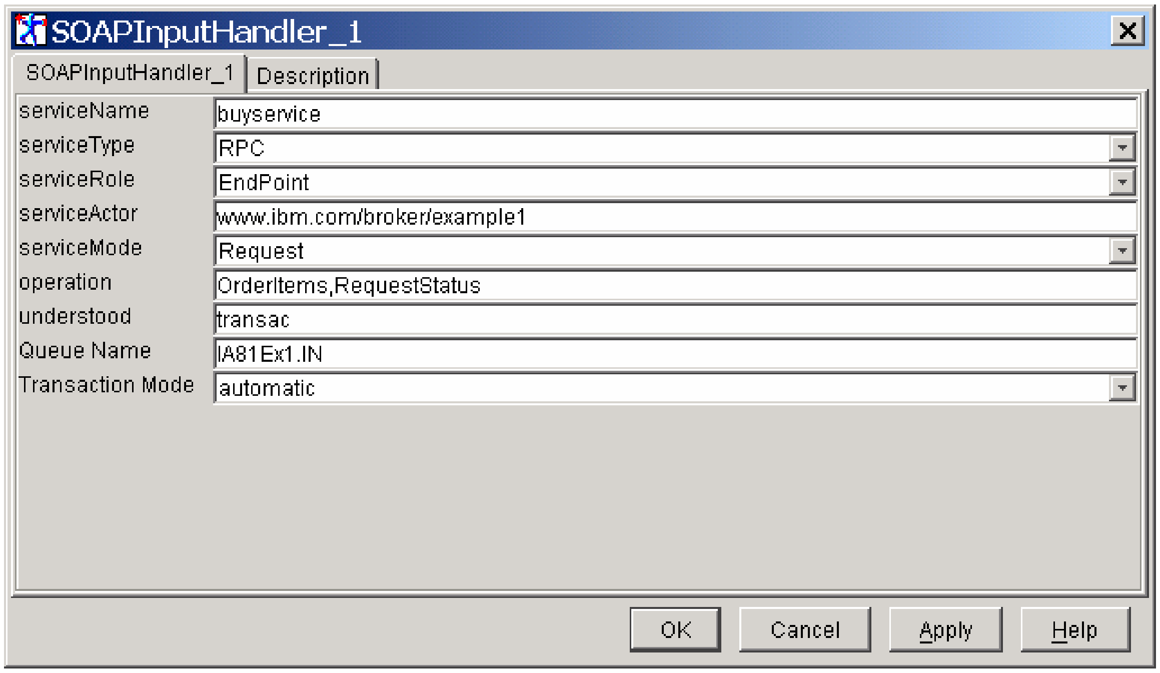
Validation is also carried out against the definition of the service as specified in the configuration properties for the SOAPInputHandler. The service definition holds a subset of the information available from the published WSDL for the service. The service definition data are stored in the Environment for the message flow and are set up using the SOAPInput Handler configuration properties.

There is no validation of the service data by the SOAPInputHandler. It is intended that the parameters to a SOAP operation or the XML document contained within a doc style request will be interpreted within the service, that is within the Message Flow.

4.3.2 Configuration Properties

Configuration of the SOAPInputHandler uses the standard WMQI properties mechanism. The properties define:

- critical information from the WSDL (ServiceName, ServiceType, operation)
- SOAP related characteristics of the message flow (ServiceMode, ServiceActor, ServiceRole, header entries understood)
- a subset of the MQInput properties to control message input



serviceName	buyservice
serviceType	RPC
serviceRole	EndPoint
serviceActor	www.ibm.com/broker/example1
serviceMode	Request
operation	OrderItems,RequestStatus
understood	transac
Queue Name	IA81Ex1.IN
Transaction Mode	automatic

ServiceName	Fully qualified service name
ServiceStyle	RPC/DOC
Service Role	SOAP Intermediary / SOAP end point service provider. Note that any single message flow is expected either to act either as an intermediary for all RPC operations or as an endpoint for all RPC operations for the service. Endpoint means the last point at which the service request is treated as a SOAP request. A message flow will still be a SOAP end point if it makes requests of non SOAP down stream systems.
ServiceActor	Name (URI) identifying the message flow as actor related to header relevance
ServiceMode	This gives the context in which the SOAPInputHandler is being called and should be: “Request” if incoming messages should be SOAP Requests “Response” if incoming messages should be SOAP Responses
operation	Comma separated list of supported operations
Understood	Comma separated list of header entries which are understood by this flow
Queue name	MQ Input queue
Transaction Mode	Transaction mode as defined for an MQInput node

4.3.3 SOAPInputHandler Terminals

The SOAPInputHandler has no input terminals; it acts as an MQInput node reading from a queue specified in its configuration parameters. The output terminals are:

SOAP Fault	This terminal indicates that a fault has been detected within the flow.
SOAP Request	This is the output node by which a decoded SOAP request is passed to the message flow.

The terminals should be wired as follows:

SOAP Fault	The Environment tree / ExceptionList tree will contain the SOAP fault data and / or exception sub tree. Control must be passed to a SOAP Fault Handler which generates the SOAP Fault message and returns this to the requestor. In the case of a node which is processing responses it will be necessary to use the correlation ID contained within the incoming message as a key to retrieve status information including the Reply destination of the original requestor. Additional processing steps may be wired in to meet local requirements.
SOAP Request	When the message structure is propagated to this terminal it implies that there is a SOAP request to process. The message flow is free to process such messages in accordance with the service implementation but must configure the compute nodes with Compute Model “All” in order to access the global environment.

If the SOAP message body contains an RPC style request rather than a document then the first step should be to branch on the operation which is contained in the Global Environment. The format and content of the SOAP body will vary according to the messages defined in the WSDL for each operation.

4.3.4 Validation of a SOAP Request

SOAP requests are validated against SOAP 1.1. When a fault is thrown within the SOAPINputHandler the message tree is propagated to the SOAPFault output terminal.

SOAP Envelope:	Message must have a SOAP Envelope as identified by string 'Envelope' as the first child of the SOAP Message. If this fails the SOAPFaultCode is set to ‘Client’, FaultString is set to 'SOAP message does not have an Envelope', and a fault is thrown.
SOAP Versioning:	The SOAPInputHandler identifies a v1.1 SOAP messages by the namespace specified as an attr4ibute of ‘Envelope’. For V1.1 it is mandatory to have the name space 'http://schemas.xmlsoap.org/soap/envelope'. If this V1.1 name space is not found or if a SOAP 1.2 name space, 'http://www.w3.org/2002/06/soap-envelope', is found the SOAPFaultCode is set to ‘VersionMismatch’, FaultString is set to 'Expected SOAP v1.1 specification message', and a fault is thrown.
SOAP Header	The message may have a SOAP Header as identified by string ‘Header’. If present the Header must be the first child of Envelope. If this fails the SOAPFaultCode is set to ‘Client’, FaultString is set to 'Badly formed message – Children of envelope incorrect', and a fault is thrown.
SOAP Body:	The message must have a SOAP Body as identified by the string 'Body' as the first child of Envelope or if a header is present, as a next sibling to the header. If this fails the SOAPFaultCode is set to ‘Client’, FaultString is set to 'Badly formed message – Children of envelope incorrect', and a fault is thrown.
SOAP Operation:	If an RPC style request message is expected, the message body must contain an Operation. The operation is validated against the list of supported operations held in the service definition set up by the configuration. The list of valid operations is held in Environment.SOAP sub tree as described in section 4.2. Nodes further down the message flow can transform or enrich the data defined by this operation and/or forward it to an ultimate SOAP Service. If first child of Body is not known to the message flow, a SOAP Client Fault is thrown with fault string ‘SOAP message has unknown operation’.

SOAP Operation: If an RPC style response message is expected the message Body must either contain a valid operation or a Fault element. If a valid operation is not found and the first child of Body is not a Fault element known to the message flow, a SOAP Client Fault is thrown with fault string ‘SOAP message has unknown operation’.

SOAP Fault Body: The Decode node expects a request message when parameter *Environment.SOAP.Service.MsgFlowMode* is set to ‘Request’. If a the node attempts to process a SOAP Fault message then a SOAP Fault will be thrown with fault sting ‘Request from SOAP Sender contained a Fault Element in Body’. A similar message will be thrown if any Fault elements are included within the Header.

4.3.5 SOAP Envelope Processing

Environment Tree: If the Envelope is found all elements, attributes and values are inserted into the environment tree *Environment.SOAP.Input.Envelope*.

4.3.6 SOAP Header Processing

A header is optional within a SOAP message. If present the header must follow the SOAP Envelope and is identified by the tag 'Header'. The Header may contain one or more Header Entries; each Header Entry is inserted into the Environment tree as a child of *Environment.SOAP.Input.Header*. The attributes ‘actor’ and ‘mustUnderstand’ are processed within the SOAPInputHandler. All attributes and values are copied to the Environment Tree.

The Header Entries are processed with respect to any mustUnderstand attributes.

If a Header Entry has a mustUnderstand field the SOAPInputHandler has to determine whether it has to be processed locally, may be processed locally or has to be forwarded to the next service. A WMQIAction attribute will be added to each Header Entry as a result of this processing as follows:

If mustUnderstand is present and it relates to this actor
    (i.e. Message flow is acting as an Endpoint rather than an Intermediary
    or the actor is not set
    or actor is this service as defined in the Service Definition Role)
Then we check if we understand it and throw a SOAP Fault if not or add WMQIAction with value “MustProcess”

If mustUnderstand is present and it does not relate to this actor then we add WMQIAction with value “Forward”

If mustUnderstand is not present we add the WMQIAction with value “MayProcess”

If a Header entry must be processed as defined by the role and mustUnderstand attributes, but the Header entry name is not defined in the node configuration parameter *Environment.SOAP.Service.HdrEntries.Understood* a SOAP Fault, is thrown with fault code MustUnderstand and fault string of ‘Did not understand *header entry name’*.

Depending on the Header entry attributes a last child attribute of *WMQIAction* is added to the *Environment.Input.Header* with values of either ‘MayProcess’, ‘MustProcess’ or ‘Forward’.

4.3.7 SOAP Body Processing

The WMQI SOAPInputHandler expects the Body to contain a single element.

In the case of service expecting an RPC request the first child of the Body will be the procedure call (that is the operation). The top level tag defines the operation which must be one supported by the service as defined in the configuration. The operation is copied to the Environment.SOAP.Input.Operation where it can be used to control the processing within the message flow. The children are the parameters of the operation and are copied as children of OutputRoot.XML.ServiceData.

In the case of a service expecting a doc style request it is assumed that the body contains a single element which is a valid XML document. This document (i.e. the first child of body) is copied directly to OutputRoot.XML

In the case of a SOAP Response there is again a difference between RPC and doc. In RPC the first child may again be a valid operation known to the service but it may also be a Fault element.

If a fault element is present the FaultCode, FaultString, FaultActor, and detail if present, are extracted and used to set up the SOAP.Fault entries in the environment. A fault is then thrown. The SOAP fault cannot be returned directly to the original requester as the original ReplyToQ and Qmgr are not known until restored within the message flow, see Section SOAP and Message Flow Implementation7 SOAP and Message Flow Implementation for further details.

If a valid operation, supported by the message flow, is found that operation is moved to Environment.SOAP.Input.Input.operation and the children copied into ServiceData; this is the same as for an RPC Request.

In the case of a doc response it is treated exactly as a doc request and passed directly through to OutputRoot.XML. It is expected that the message flow understands the structure of a doc response. This includes any fault elements within the document.

4.3.8 XML

The XML tree will contain only the data to be processed by the service. This will vary according to whether the request is RPC or doc.

In the case of the RPC type request the top level tag will always be Service Data. All parameters will then be contained as substructures within ServiceData.

The example below shows the incoming SOAP message:

```
<soapenv:Envelope
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <soapenv:Header>
        <Transaction soapenv:mustUnderstand = "1"> Type 1 </Transaction>
</soapenv:Header>
```

```
           <soapenv:Body>
                     <ns1:OrderItems xmlns:ns1="http://www.buyservice.com/buy">
                     <buyReq>
                              <toBuy>
                              <partcode>PT134xxx4444</partcode>
                              <amount>00000020</amount>
                              </toBuy>
                              <Customer>
                              <firstName>John</firstName>
                              <lastName>Smith</lastName>
                              <Account>xxx222333444</Account>
                              </Customer>
            </buyReq>
           </ns1:OrderItems>
          </soapenv:Body>
         </soapenv:Envelope>
```

When this has been processed by the SOAPInputHandler the message flow will be passed the following information:

In the Environment.SOAP.Input subtree

```
<operation>OrderItems</operation>
 < Envelope
        encodingStyle = 'http://schemas.xmlsoap.org/soap/encoding/'
        soapenv       = 'http://schemas.xmlsoap.org/soap/envelope/'
        xsd           = 'http://www.w3.org/2001/XMLSchema'
        xsi           = 'http://www.w3.org/2001/XMLSchema-instance'
        SOAP-ENC      = 'http://schemas.xmlsoap.org/soap/encoding/'
  </Envelope>                                    )
   <Header>
         <Transaction mustUnderstand = 1   WMQIAction     = 'Forward'>

   </Header>
```

In the XML subtree

```
<ServiceData ns1="http://www.buyservice.com/buy">
     <buyReq>
             <toBuy>
                <partcode>PT134xxx4444</partcode>
                <amount>00000020</amount>
                </toBuy>
              <Customer>
                <firstName>John</firstName>
                 <lastName>Smith</lastName>
                 <Account>xxx222333444</Account>
             </Customer>
     </buyReq>
</ServiceData>
```

In the case of a doc style service request the complete document will sit directly beneath the XML tag. It is expected that the WMQI message style fully understands how to process this document.

In both cases an MRM format may be used to reparse or manipulate the MRM data passed to the message flow.

4.4 SOAP Fault Handler

4.4.1 Overview

The SOAP fault handler will be called in two circumstances. The first is when a specific SOAP fault has been detected, for example the SOAPInputHandler detects a “VersionMismatch”. In this case the details of the fault are recorded in the Environment.SOAP.Fault fields. The SOAP Fault Handler detects this by testing for a non NULL FaultCode. It is therefore important to ensure that the FaultCode is only set when a SOAP error has actually occurred.

The second circumstance arises when an error occurs which is a WMQI error rather than a detected fault in the structure or content of a SOAP message. All such faults should be caught and passed to the SOAP Fault Handler which will generate a Server fault and process the exception list to give additional information.

The SOAP Fault handler will return the fault message to the requestor by placing it on the ReplyToQ. If a failure occurs during the processing of the Fault the message will be propagated to the failure terminal to be handled by non SOAP error handling mechanisms.

4.4.2 Fault Specification

The SOAP Fault Handler supports the SOAP1.1 requirements. A separate SOAP 1.2 Fault Handler will be introduced at a later stage if required. In the initial release of the SupportPac the arrival of SOAP 1.2 messages will generate a Version Mismatch soap fault which will be passed back to the requestor in a SOAP 1.1 fault format.

The following SOAP 1.1 faults are generated:

Name	Meaning
VersionMismatch	The processing party found an invalid namespace for the SOAP Envelope element
MustUnderstand	An immediate child element of the SOAP Header element that was either not understood or not obeyed by the processing party contained a SOAP mustUnderstand attribute with a value of "1"
Client	The Client class of errors indicate that the message was incorrectly formed or did not contain the appropriate information in order to succeed. It is generally an indication that the message should not be resent without change.
Server	The Server class of errors indicate that the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to the processing of the message. The message may succeed at a later point in time. SOAP fault codes within the message flow may use either Client or Server faults depending on the nature of the problem. If a general WMQI error is caught it will be treated as a server error.

The SOAP Fault Handler sends the SOAP fault message back to the requestor whose reply address is assumed to be in the ReplyToQMgr and ReplyToQ of the current message tree. Following standard MQ practice the SOAPFaultHandler will also take a CorrelationID from the current message. It is the responsibility of the message flow to ensure that CorrelationID and Reply destination are correctly set in the incoming message header.

4.4.3 Configuration Properties

The SOAP fault handler has two configuration properties defining the persistence and transactionality of the resulting SOAP Fault message.

The SOAP Fault Handler will expect the current message to have an MQMD header with a ReplyToQ, ReplyToQMgr and CorrelationID.

The SOAP Fault Handler will expect to have either details of a SOAP fault in the Environment.SOAP.Fault element (in particular a SOAP fault must have a non NULL value for Environment.SOAP.Fault.FaultCode) or one or more WMQI exceptions defined in the ExceptionList tree.

The SOAPFaultHandler will also add to the SOAPFault message any Header information placed in the Environment.SOAP.Output tree. It is the responsibility of the message flow to ensure that headers are correctly placed in this location if they are required.

4.4.4 SOAP Fault Handler Terminals

Input

The SOAP Fault Handler has a single input terminal on which it expects to receive a message tree including Root, the Environment tree, and optionally an ExceptionList.

No output

The SOAP Fault handler processes a fault passed to it in the Environment or Exception sub tree and generates a SOAP fault message which it returns to the ReplyToQ of the requestor using an embedded MQReply node. If the message is successfully submitted this path is finished and there will no further message tree propagated out of the SOAP fault handler.

Failure
In the case of a failure within the SOAP fault handler control will be passed to the failure node to be handled using whatever error handling processes are in use at the WMQI installation. This may indicate either that there were problems in processing an exception or that there were problems with the Reply address used to submit the message.

4.4.5 Message Format

SOAP faults generated by the SOAP Fault node contain a standard fully qualified fault element in the Body as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
<soapenv:Body>
<soapenv:Fault>
  <faultcode>soapenv:Server</faultcode>
  <faultstring>Server Error</faultstring>
 <faultactor>WMQIservice1</faultactor>
 <detail><Message>The  request  for  service  servicenamefailed.</Message>
</detail>
</soapenv:Fault>
</soapenv:Body>
</soapenv:Envelope>
```

The detail element is only present when an error occurs in the processing of the body. Details of exceptions extracted from the WMQI ExceptionList are placed in detail. Note also that the faultcode must be standard and must be qualified. IA81 uses soapenv as the tag for the standard namespace `"http://schemas.xmlsoap.org/soap/envelope/"`

In the case where SOAP faults have been directly detected the faultcode, faultstring and faultactor will be taken directly from the SOAP.Fault fields in the environment. The detail message will contain a standard message saying that the request for *ServiceName* failed, where service name is the name taken from the SOAP.Service.ServiceName element in the Environment tree.

In the case where an exception has been thrown within WMQI and passed to the SOAP fault node the Fault code will be “soapenv:Server” (assuming we have defined our namespace as soapenv as show in the example), the actor and detail will be as for SOAP detected faults and the fault string will be “WMQI failure – see detail”. The detail message will contain a string summarising the exception data from the WMI ExceptionList structure.

SOAP Fault Message where SOAP Processor cannot interpret the mustUnderstand Header entries:

```
<env:Envelope xmlns:soapenv='http://www.w3.org/2001/06/soap-envelope'
        xmlns:soapfault='http://www.w3.org/2001/06/soap-faults' >
   <soapenv:Body>
   <soapenv:Fault>
     <faultcode>soapenv:MustUnderstand</faultcode>
     <faultstring> Did not understand header entry name </faultstring>
   </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

When WMQI acts as an intermediary it is the responsibility of the SOAPResponseHandler to process the response from a downstream service. If a SOAP fault is received the

SOAPResponseHandler must correctly set up the ReplyToQ, correlation ID and Environment.SOAP.Fault elements before throwing an error to be caught by the SOAPFaultHandler.

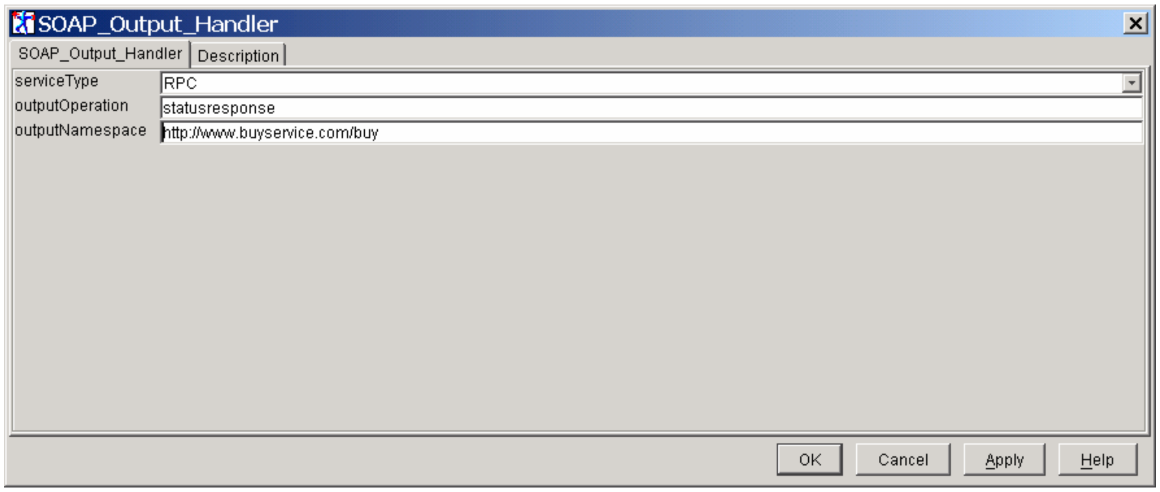
4.5 SOAPOutputHandler

4.5.1 Overview

The SOAPOutputHandler is required to generate a valid SOAP response message taking as its body the message data in the XML or MRM message tree and additional information in the Environment tree. The message generated will be propagated to the SOAPOutput terminal to be sent by a MQOuput or MQReply node.

4.5.2 Configuration

The SOAPOutputHandler requires the following information about the service type to be defined using the WMQI property configuration.



serviceType	RPC
outputOperation	statusresponse
outputNamespace	http://www.buyservice.com/buy

ServiceType	RPC/DOC determines the format of the Body of the output message
outputOperation	Specifies the operation name to be used for an RPC response
NameSpace	Namespace to be used to qualify an operation or top level document tag.

All other configuration data is taken from the Environment tree set up by the message flow before calling the SOAPOutputHandler.

The SOAPOutputHandler will expect the current message to have either an XML tree or an MRM tree which contains the data to be contained in the SOAP Body. This will either be a complete XML document (in the case of a DOC type message) or the parameters for an operation (in the case of an RPC message). It is the responsibility of any Message Flow acting as a Web Service provider to set up the response message in this way.

The SOAPOutputHandler will look for a Header in Environment.SOAP.Output.Header and if found will add the Header and all its entries to the outgoing message.

4.5.3 Terminals

Input

The SOAPOutputHandler has a single input terminal on which it expects to receive a message tree including Root, the Environment tree, and optionally an Exception tree.

SOAP Output

The SOAPOutputHandler processes the message data passed to it and if successful propagates a correctly formatted SOAP message to the SOAP Output Terminal.

SOAP Fault

In the case of a failure within the SOAPOutputHandler the SOAPOutputHandler will set up the SOAP Fault data and pass control to the SOAP Fault node.

The terminals should be wired as follows:

SOAP Output	This should be wired to an MQOutput Node (for a Request) or to an MQReply node (for a Response).
SOAP Fault	Control should be passed to a SOAPFaultHandler

4.5.4 Message Generation

SOAP messages are created in the XML tree. The SOAPOutputHandler generates a standard SOAP 1.1 envelope with appropriate namespaces. The prefix soapenv is used for the soap envelope name space
“http//schemas.xmlsoap.org/soap/envelope”

A header is added if one is found in the SOAP.Output tree. No attempt is made to validate the contents of the header and it is the responsibility of the message flow to create appropriate header entries.

A SOAP Body is created and contents added dependent on message type.

If the message type is RPC then an operation must be added. The SOAPOutputHandler will expect to find the operation in the output tree and will throw an error if none is found. The Operation will be qualified either by a specific name space found as an element of the operation in the Output tree An error will be thrown if no name space or no operation.

Once an operation has been added the children of either the MRM or XML message tree (whichever is present) will be added as children of the operation.

If the service is a DOC type then the child of MRM or XML is added to the SOAP Body qualified with the namespace from the configuration data.

5 Other Component Details

5.1 Gateway for SOAP/http to SOAP/JMS Protocol Conversion

5.1.1 Requirements

The component which acts as a gateway for protocol conversion is required to extend access to WMQI Web Services by allowing those clients which have no JMS or MQ support, or which cannot reach a WMQI MQ server because they are located outside firewalls, to make requests over http to a gateway. This is shown in Figure 12 below.

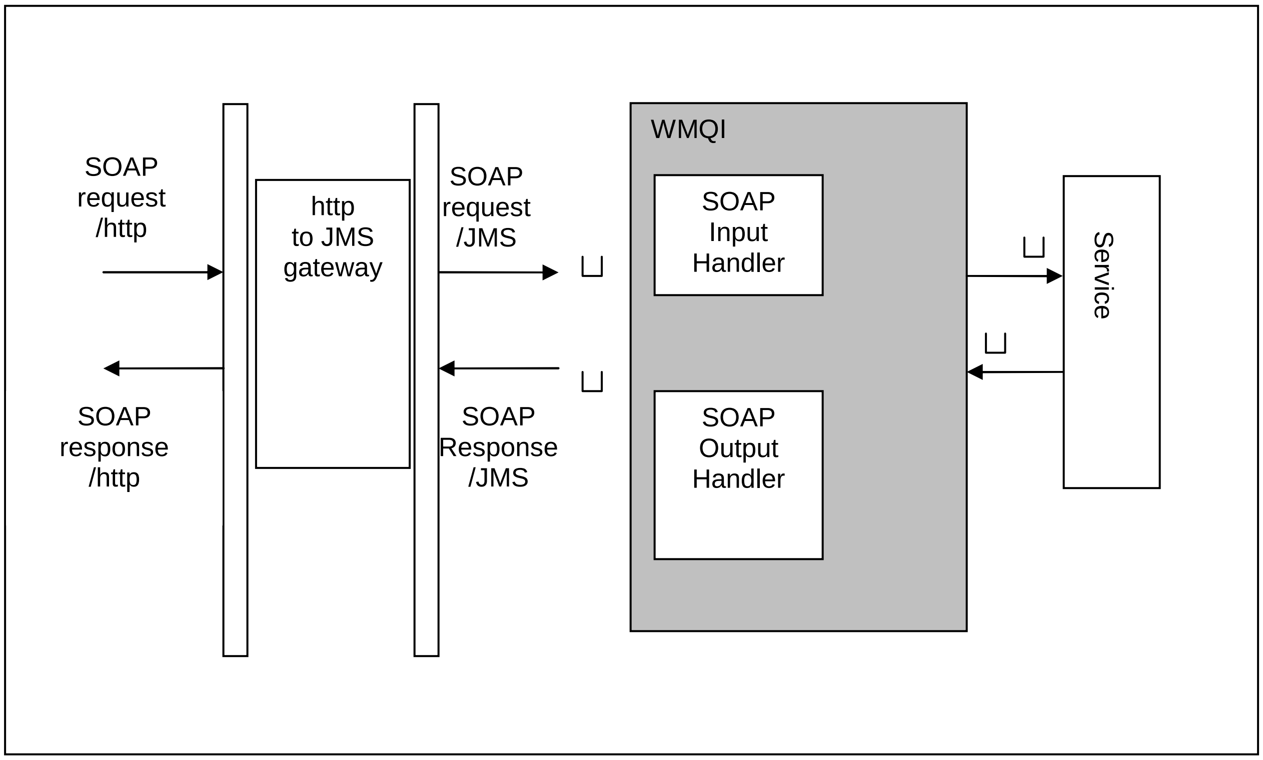


Figure 12 - http proxy listener in DMZ

The gateway is required to:

- receive SOAP requests over http
- generate corresponding SOAP requests over JMS to an associated WMQI service
- receive the SOAP response or fault from the WMQI service
- return the SOAP response or fault to the originating requestor over http

In more general terms the gateway will be required to authenticate incoming http connections and to provide a scalable solution.

5.1.2 Options

There are a number of possibilities for the gateway and selection will depend on the particular context. We have proposed two options for operation with this SupportPac. The first is applicable where a fully supported, full scalable product is required to support a production environment. For this we recommend the Web Services Gateway which is described in detail below. The second option is a plug in JMS transport for the Axis server, which is included in the SupportPac.

5.1.3 Web Services Gateway

The first option is the Web Services Gateway which is included in WebSphere Application Server Network Deployment Version 5.0. This is the option that we would recommend for a production environment. It provides the required functionality in a fully supported, currently available product. It is a WebSphere application and as such inherits the characteristics which ensure that it can meet non-functional requirements which are commonly encountered in a large scale production environment.

This option is supported by the tooling in the WSADIE development environment. This supports the development of services and associated WSDL and the generation of client stubs for requestors. In addition WSADIE supports the generation of XML schema and WSDL definitions. Schemas may be imported into the WSDL to define message parts and also into WMQI to improve the development of the message flow to process a SOAP body and invoke back end services.

Based on the WSDL given to the gateway for services to be accessed over JMS, the gateway will also generate the WSDL which allows the service to be publish by the gateway to make it accessible over http.

Disadvantages of this option are that it involves additional cost (for WebSphere Application Server Network Deployment) and the additional skills required to deploy and manage a WebSphere application. The use of a gateway based on J2EE Application server may be thought to be unacceptable to MQ/WMQI customers but there are no proposed solutions to this problem which do not require at least an http listener and servlet engine, a move to Web Services accessed across the internet will inevitably bring a move to new technologies. Those customers who find this unacceptable may be happier remaining with solutions based only on JMS.

5.1.4 Axis Server with JMS handler

The second option is supplied with source code as part of this SupportPac. It is not a supported product but an extension of the Axis server which provides a JMS transport handler. This would be an appropriate starting point for working with Web Services or for organisations with the skills to support Axis and extensions.

Using this extension the server can take SOAP Requests over HTTP, or any other Axis transport, convert the request to JMS, and send it to a WMQI input queue. The WMQI service returns a SOAP Response over JMS and the axis server receives the SOAP response based on JMS correlation ID of the request message and sends it back using the same Axis instance.

This approach supports both RPC and document SOAP transactions.

6 Sample Implementations

This SupportPac includes three worked examples of how WMQI can operate in a Web Services environment. The first shows WMQI providing a Web Services interface receiving SOAP requests and fronting a down stream application which provides the business functionality. The second shows how WMQI can receive messages in any format and can use these to create SOAP request messages which are then sent to a Web Service over JMS. Finally we look at a situation where WMQI simply acts as an intermediary receiving SOAP requests and forwarding them to a Web Service.

6.1 WMQI Web Services with JMS/MQ access

6.1.1 Service Functionality

This first scenario demonstrates how WMQI can provide a Web Services front end to any MQ enabled down stream capabilities. The transport layer is MQ based and can handle both base MQ and JMS messages. The WMQI SOAP Handlers make no use of MQRFH2 headers but will forward them to the message flow if received. In this scenario WMQI presents an “Order” service to Web Service clients which is available to the clients over an MQ transport. The service offers two operations, which demonstrate both Request / Reply capability and “Fire and Forget” capability. One operation also demonstrates how a database can be used to maintain state between SOAP requests and SOAP responses, an essential task when asynchronous transports are used in the service implementation.

The first operation, OrderItems, enables a requestor to place a request for a part. This operation treats the request as an asynchronous operation in that a response is returned to the requestor acknowledging receipt of the request when the request is sent to the downstream application not when the order is complete. It is assumed that completion of the transactions which the down stream application carries out in order to complete the order may take an unknown amount of time.

In this operation the response to the requestor is made before control passes out of the message flow and there is no need for the message flow to manage state between request and response. Although WMQI does not have to manage state in a normal environment the original requestor would have to manage state in that it needs to identify submitted orders so that status checks can be made at a later stage.

The second operation, “RequestStatus”, is also a Request / Reply operation with a response from a down stream application which will be received asynchronously by a second message flow. Status information will be stored in a database to enable the response to be returned to the requestor, this will be keyed on the message ID of the request which is expected to be present as the correlation ID of the response. This is based on the assumption that Web Services requestor will also be able to receive and process the information asynchronously.

6.1.2 Message Flow Description

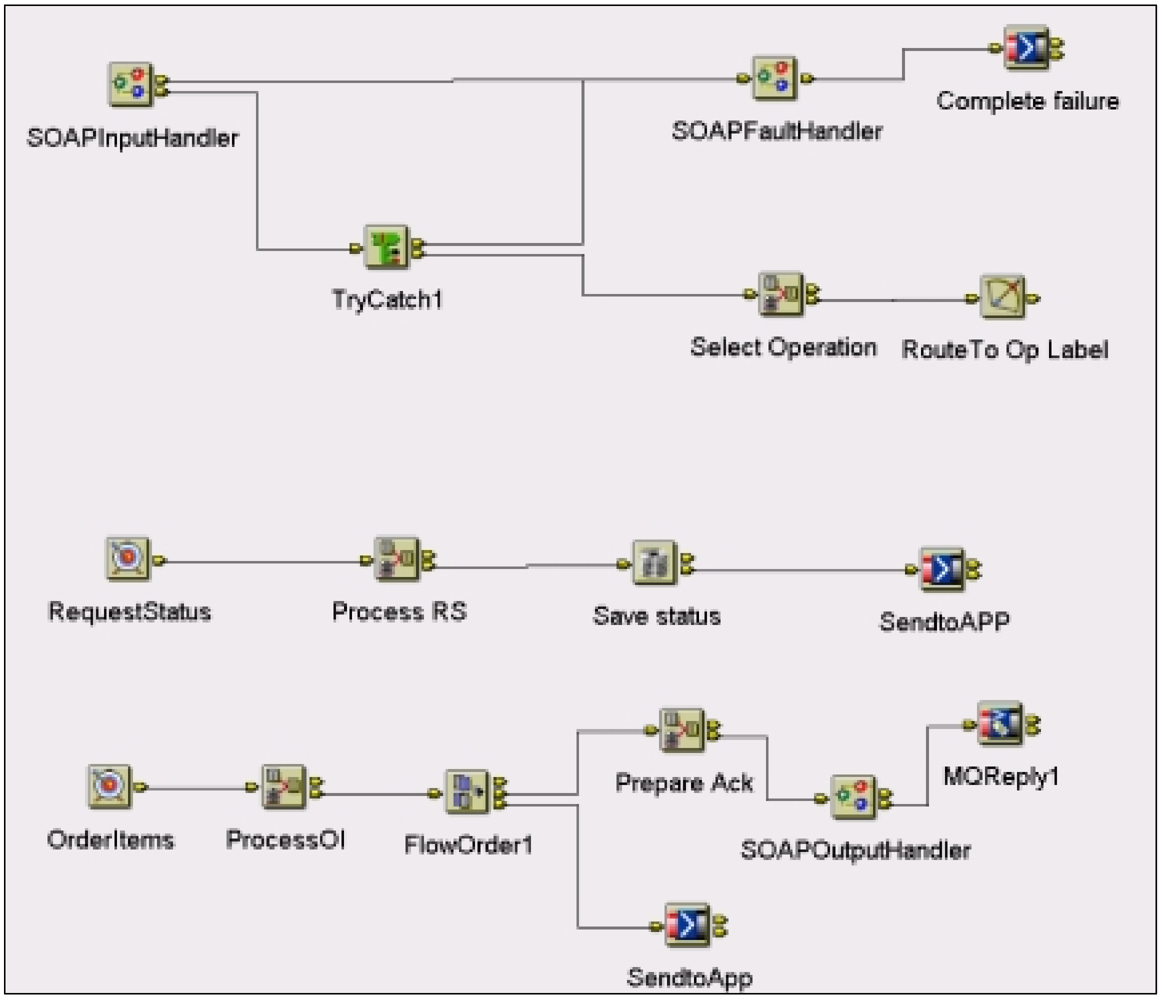


Figure 13 Example 1 Request Message Flow

The SOAPInputHandler receives and validates a request propagating the service data to be processed to the SOAP Request Terminal if successful and to the SOAP Fault terminal if unsuccessful.

After successfully decoding the request one of the two valid operations (OrderItems or Request Status) is set up in the Environment tree. This is used by the Select Operation node to branch to the correct label in the message flow.

The OrderItems part of the flow prepares and submits a message to the downstream application and prepares an acknowledgement which it passes to the SOAPOutputHandler. This return a SOAP response to the original requestor. Note that all nodes copy message headers and that the incoming MQMD still contains the Reply To address.

The RequestStatus part of the flow prepares a message for the legacy service, saves the status that will be needed to send a SOAP Response from the Response Message flow and then dispatches the request to the down stream application.

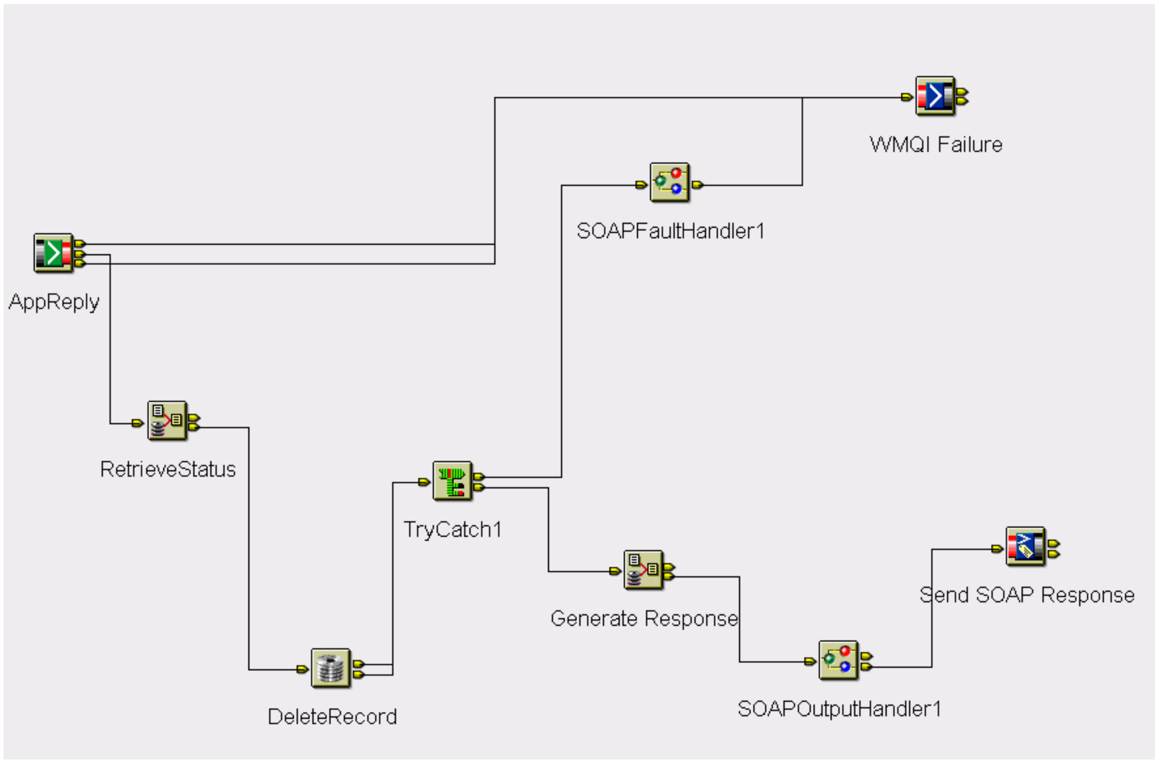


Figure 14 Example 1 Response Message Flow

The Response Message flow receives a response (standard MQ not SOAP) from the application and using the correlation ID of the response retrieves the status saved from the incoming request, then deleting the record. It uses this states information to set up correct MQMD headers and destination for the response. Note that correct SOAP responses can only be sent when a reply address has been retrieved, other failures must pass to the generic failure queue.

The Compute node generates the appropriate response data (setting SOAP Fault values in the Environment if errors are encountered) and passes the data to the SOAPOutputHandler which creates a SOAP response and returns it to the original requestor. All failures are caught first by the SOAPFaultHandler which attempts to return a valid SOAP Fault to the requestor and then by a general WMQI failure node, which in this case simply write a message to the IA81EX1_FAILURE queue.

6.1.3 SOAP Messages

```
<soapenv:Envelope
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <soapenv:Header>
  <Transaction> Type 1 </Transaction>
</soapenv:Header>
 <soapenv:Body>
  <ns1:OrderItems xmlns:ns1="http://www.buyservice.com/buy">
   <buyReq>
    <toBuy>
```

```
     <partcode>PT134xxx4444</partcode>
     <amount>00000020</amount>
    </toBuy>
    <Customer>
     <firstName>John</firstName>
     <lastName>Smith</lastName>
     <Account>xxx222333444</Account>
    </Customer>
   </buyReq>
  </ns1:OrderItems>
 </soapenv:Body>
</soapenv:Envelope>
```

Clients wishing to invoke the RequestStatus operation would submit a message similar to the following:

```
<soapenv:Envelope
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <soapenv:Header>
  <Transaction> Type 1 </Transaction>
</soapenv:Header>
 <soapenv:Body>
  <ns1:RequestStatus xmlns:ns1="http://www.buyservice.com/buy">
   <buyReq>
    <toBuy>
     <partcode>PT134xxx4444</partcode>
     <amount>00000020</amount>
    </toBuy>
    <Customer>
     <firstName>John</firstName>
     <lastName>Smith</lastName>
     <Account>xxx222333444</Account>
    </Customer>
   </buyReq>
  </ns1:RequestStatus>
 </soapenv:Body>
</soapenv:Envelope>
```

The following WSDL defines the service offered in Example 1. The definitions of the messages used in the operations are held in the imported schema file. If imported into the MRM this schema file would give the XML definition of the parameters for each operation.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://www.buyservice.com/buy"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <import location="C:/IA81/Example1.xsd" namespace="http://www.buyservice.com/buy"/>
   <message name="buyResponse">
      <part name="return" type="buyRespType"></part>
   </message>
   <message name="buyRequest">
      <part name="buyReq" type="buyReqType"></part>
```

```
  </message>
  <message name="StatusResponse">
    <part name="return" type="StatusRespType"></part>
  </message>
  <message name="StatusRequest">
    <part name="buyReq" type="buyStatusReqType"></part>
  </message>

  <portType name="BuyService">
    <operation name="OrderItems">
      <input message="buyRequest"/>
      <output message="buyResponse"/>
    </operation>
    <operation name="StatusRequest">
      <input message="buyRequest"/>
      <output message="buyResponse"/>
    </operation>
  </portType>

  <binding name="buyServiceSoapBinding1" type="BuyService">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="OrderItems">
      <input>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://www.buyservice.com/buy" use="literal"/>
        <soap:header     use="encoded"     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.buyservice.com/buy"></soap:header>
      </input>
      <output>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://www.buyservice.com/buy" use="literal"/>
      </output>
    </operation>
  </binding>

  <binding name="buyServiceSoapBinding2" type="BuyService">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="RequestStatus">
      <input>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://www.buyservice.com/buy" use="literal"/>
        <soap:header     use="encoded"     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.buyservice.com/buy"></soap:header>
      </input>
      <output>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://www.buyservice.com/buy" use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="buyService">
    <port binding="buyServiceSoapBinding" name="JMSPortJNDI">
      <soap:address
location="jms://IA81?requestQueue=IA81_Ex1_SOAPIn;responseQueue=IA81_Ex1_SOAPResp"></soap:addr
ess>
    </port>
  </service>
</definitions>
```

6.1.4 Instructions

These instructions assume that the message flows and message sets have been imported to your broker and that the database WMQISOAP has been set up with the table required for the IA81 exercises. Instructions on the initial set up are given in section 8 Description of Deliverables.

6.1.4.1 Create Queues

The first example requires the following queues to be created on the queue manager of your broker:

IA81_Ex1_SOAPIn	this is where the initial request should be placed, we advise that this be set up with a backout threshold and backout queue
IA81_Ex1_SOAPResp	this is where all responses to the service requestor are returned, this will include acknowledgements, responses and faults
IA81_Ex1_ToApp	this is the queue from which the legacy application takes its input. Example 1 takes a SOAP message and places a corresponding legacy request on this queue.
IA81_Ex1_FromApp	The legacy application returns responses to this queue. Example 1 collects these responses and places a corresponding SOAP response on IA81_Ex1_SOAPResp
IA81_Failure	This queue catches any failure which cannot be processed as a SOAP fault.

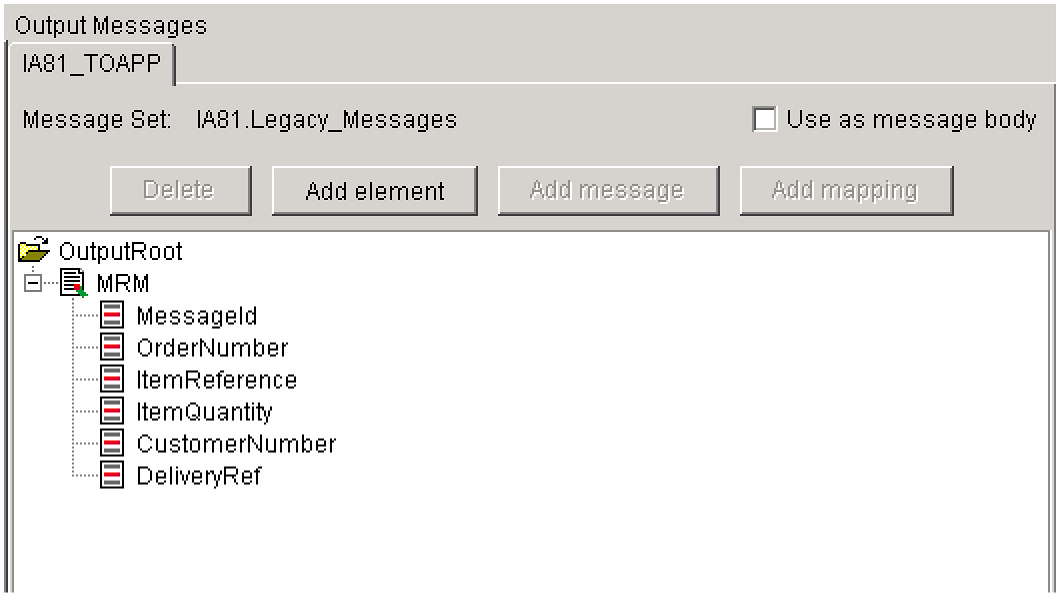
It also assumes that the Queue Manager used for clients and legacy applications is the broker's queue manager.

6.1.4.2 Deploy Message Flows and Message Set

Assign the message flows IA81_Ex1_SOAP to Legacy and IA81_Ex1_Response to an appropriate execution group.
Assign the message set IA81.Legacy_Messages to the same execution group and deploy.

6.1.4.3 Legacy Application

The legacy application reads messages in the format defined in the message set IA81 legacy. The first message simulates a request to a back end service.

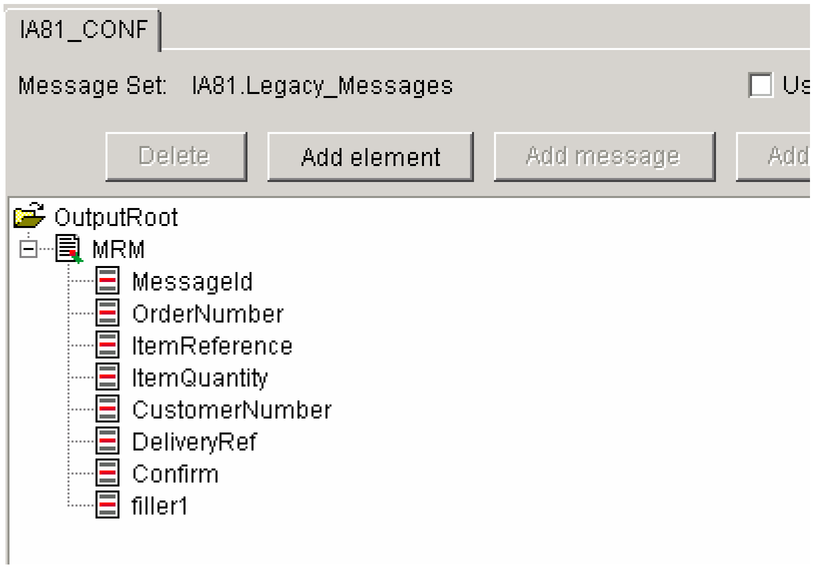


Message Set: IA81.Legacy_Messages

```
OutputRoot
  MRM
    MessageId
    OrderNumber
    ItemReference
    ItemQuantity
    CustomerNumber
    DeliveryRef
```

The SOAP request with operation OrderItems generates this message, with the first field MessageId set to IA81ORD. This is accepted by the application, which offers a Fire and Forget service and no response is returned.

The SOAP request with operation RequestStatus also generates this message, with the first field MessageId set to IA81RQS. This is accepted by the application which replies with the first field MessageId set to IA81CONF, followed by the same fields as in the request plus two extra fields indicating whether the order has been accepted. The decision on whether to give 'Y' or 'N' in the Confirm field is whether there is a 'Y' as any character of the input message ItemReference field. The application uses the ReplyToQ in the MQMD header as its response queue.



```
OutputRoot
  MRM
    MessageId
    OrderNumber
    ItemReference
    ItemQuantity
    CustomerNumber
    DeliveryRef
    Confirm
    filler1
```

6.1.4.4 Run the example

1. Start the legacy application …\IA81\legacy\legacyservice.exe which takes two arguments, queue manager and queue. In this case the queue manager is that of the broker and the queue is IA81_Ex1_ToApp.
2. Place a message containing the data in …\IA81\Example1\IA81_Ex1_Input1.txt onto queue IA81_Ex1_SOAPIn, ensure that the reply queue is set to IA81_Ex1_SOAPResp. This can be done using the API exerciser in advanced mode or any other tool of your choice. This message contains an SOAP request with operation OrderItems.
3. Check that an acknowledgement is received on IA81_Ex1_SOAPResp.
4. Place a message containing the data in …\IA81\Example1\IA81_Ex1_Input2.txt onto queue IA81_Ex1_SOAPIn, ensure that the reply queue is set to IA81_Ex1_SOAPResp. This message contains an SOAP request with operation RequestStatus.
5. Check that a confirmation message is received on queue IA81_Ex1_SOAPResp

Note that if a fault occurs this should also be available as a SOAP fault on the queue IA81_Ex1_SOAPResp. The contents of any messages on this queue should therefore be examined before assuming success.

6.2 Legacy Clients Request a SOAP service via WMQI

6.2.1 Service Functionality

This example shows access to a simple service which is called from a WMQI message flow. A legacy system makes a request to place an order and the WMQI message flow takes the request, build a SOAP Request message and invokes a Web Service.

6.2.2 Message Flow Description

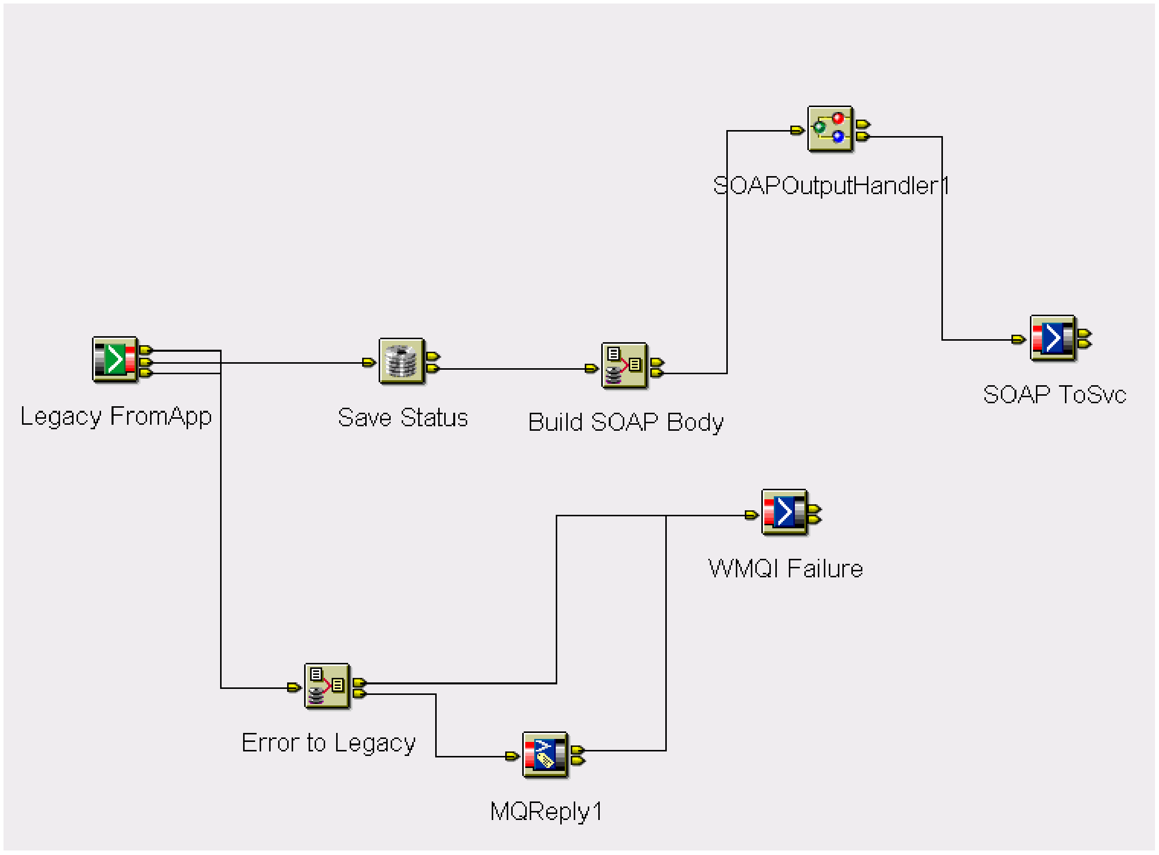


Figure 15 Exercise 2 Request Message Flow

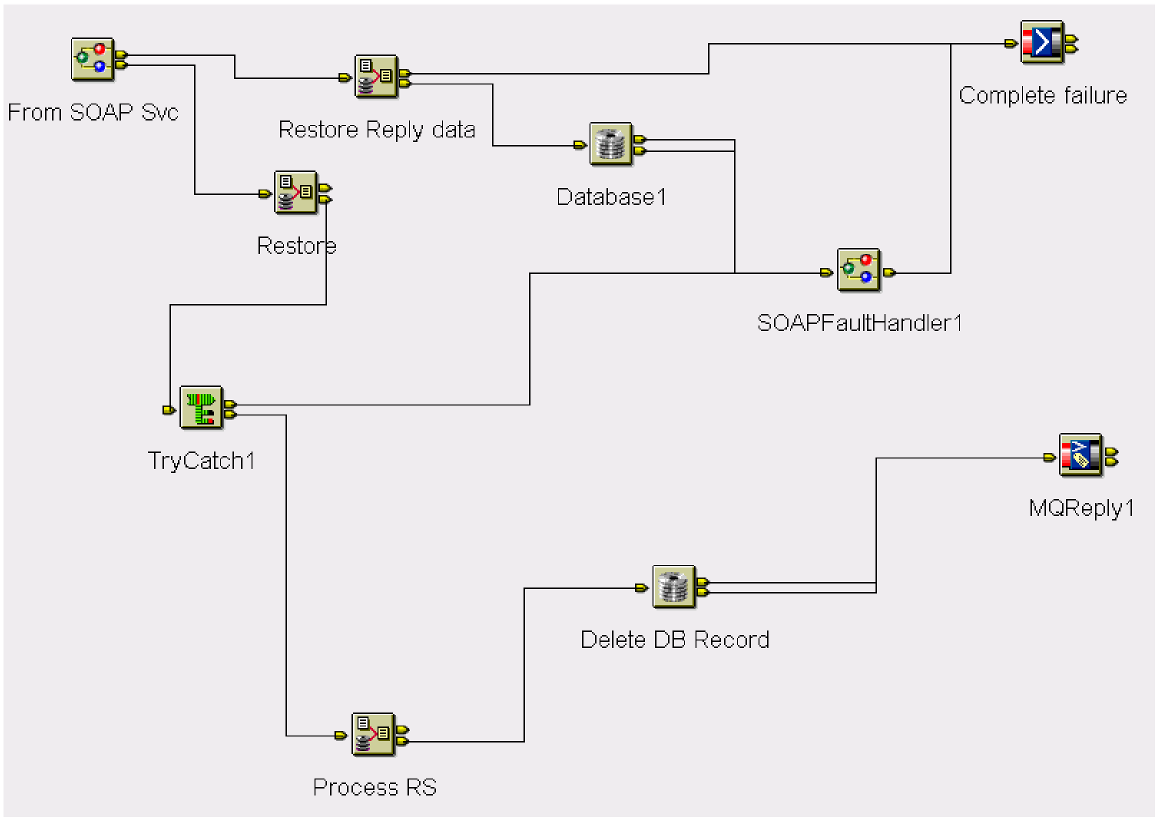


Figure 16 Exercise 2 Response Message Flow

The Response Message flow receives a SOAP response from the Web Service. The Compute node (Process RS) uses the correlation ID of the response to retrieve the state saved in the corresponding Request Message Flow and sets up a SOAP Response, this includes setting the MQMD values for destination from retrieved state. The next node deletes the database record. Note again that failures before a reply address has been retrieved go to the generic failure queue rather than being returned to the requestor.

The Compute node generates the appropriate response data (setting SOAP Fault values in the Environment if errors are encountered) and passes the data to the SOAPOutputHandler which creates a SOAP response and returns it to the original requestor. All failures are caught first by the SOAPFaultHandler which attempts to return a valid SOAP Fault to the requestor and then by a general WMQI failure node, which in this case simply write a message to the IA81EX1_FAILURE queue.

6.2.3 Instructions

These instructions assume that the message flows and message sets have been imported to your broker and that the database WMQISOAP has been set up with the table required for the IA81 exercises. Instructions on the initial set up are given in section 8 Description of Deliverables.

6.2.3.1 Create Queues

The second example requires the following queues to be created on the queue manager of your broker:

IA81_Ex2_FromApp	this is where the initial request should be placed, we advise that this be set up with a backout threshold and backout queue
IA81_Ex2_AppResp	this is where all responses to the service requestor are returned, this will include acknowledgements, responses and faults
IA81_Ex2_ToSvc	this is the queue from which the SOAP service takes its input. Example 2 takes a legacy message and places a corresponding SOAP request on this queue.
IA81_Ex1_FromSvc	The SOAP service returns SOAP responses to this queue. Example 2 collects these responses and places a corresponding legacy message on IA81_Ex2_AppResp
IA81_Failure	This queue catches any failure which cannot be processed as a SOAP fault.

It also assumes that the Queue Manager used is SOAP2. This will require changing in the following places if you use a different queue manager.

6.2.3.2 Deploy Message Flows and Message Set

Assign the message flows IA81_Ex2_SOAPSvc, IA81_Ex2_SOAP_Response and IA81.Ex2_Legacy_Request to an appropriate execution group.
Assign the message set IA81.Legacy_Messages to the same execution group and deploy.

IA81_Ex2_Legacy_Request processes legacy requests and issues a SOAP request
IA81_Ex2_SOAPSvc is a message flow which simulates a SOAP service, receiving the SOAP request and returning a SOAP response.
IA81_Ex2_SOAP_Response receives the SOAP response and generates a legacy response.

6.2.3.3 Run the example

1. Place a message containing the data in …\IA81\Example2\LegacyRequestMsg.txt onto queue IA81_Ex2_FromApp, ensure that the reply queue is set to IA81_Ex2_AppResp. This can be done using the API exerciser in advanced mode or any other tool of your choice. This message contains an SOAP request with operation OrderItems.
2. Check that a response is received on IA81_Ex2_AppResp.

Note that if a fault occurs this should also be available on the queue IA81_Ex2_AppResp. The contents of any messages on this queue should therefore be examined before assuming success.

6.3 WMQI acts as Intermediary between two SOAP interfaces

6.3.1 Service Functionality

In this example we have the same business functionality as in example 1. However here both requestor and service are Web Service enabled and all communications use SOAP messages. The role of WMQI is to translate between the format required by the published WSDL representing the service as hosted by WMQI and the WSDL for the back end application. This message flow is not included in the first release but will be included in a second release which supports access to SOAP services over http as well as JMS. The flow shown below is an initial view of an MQ version. There will also be a version accessing a service over http.

6.3.2 Message Flow Description

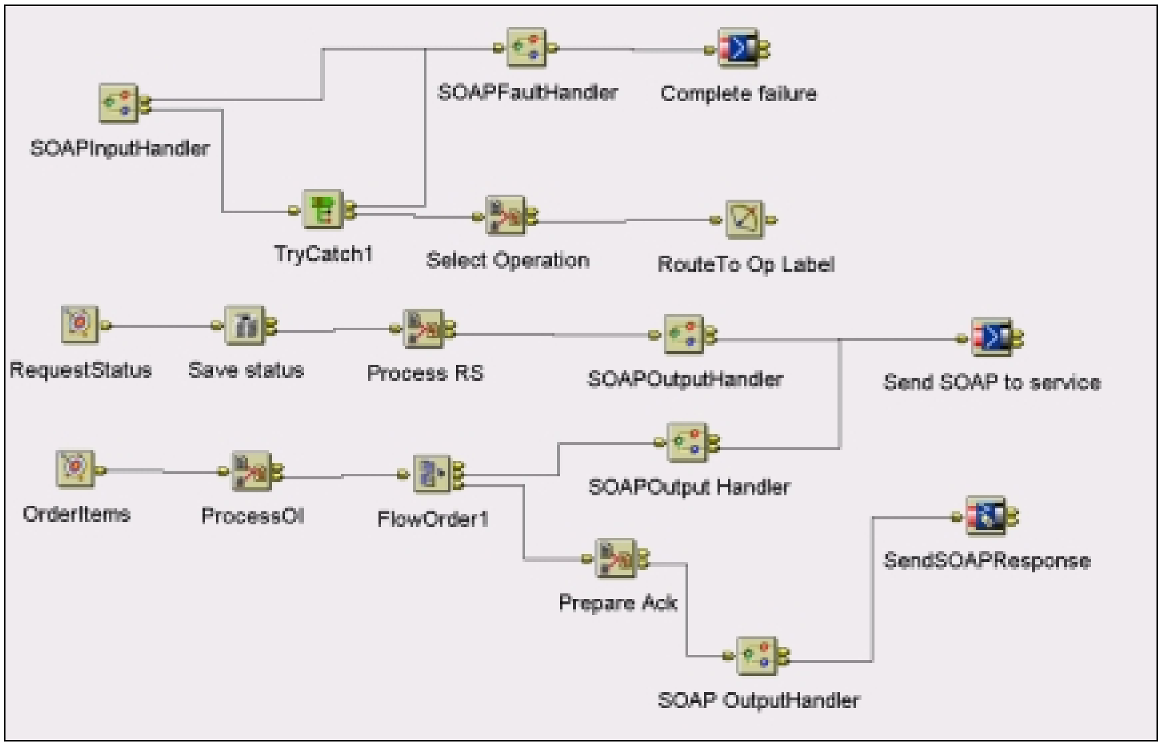


Figure 17 Example 3 Request Message Flow

The SOAPInputHandler receives and validates a request propagating the service data to be processed to the SOAP Request Terminal if successful and to the SOAP Fault terminal if unsuccessful.

After successfully decoding one of the two valid operations (OrderItems or Request Status) is set up in the Environment tree. This is used by the Select Operation node to branch to the correct label in the message flow.

The OrderItems part of the flow prepares a message, calls the SOAPOutputHandler to submit it to the downstream web service application, then prepares an acknowledgement and returns this to the requestor using the SOAPOutputHandler. Note that all nodes copy message headers and that the incoming MQMD still contains the Reply To address.

The RequestStatus part of the flow prepares a message for the legacy service, saves the status that will be needed to send a SOAP Response from the Response Message flow and then dispatches the request to the

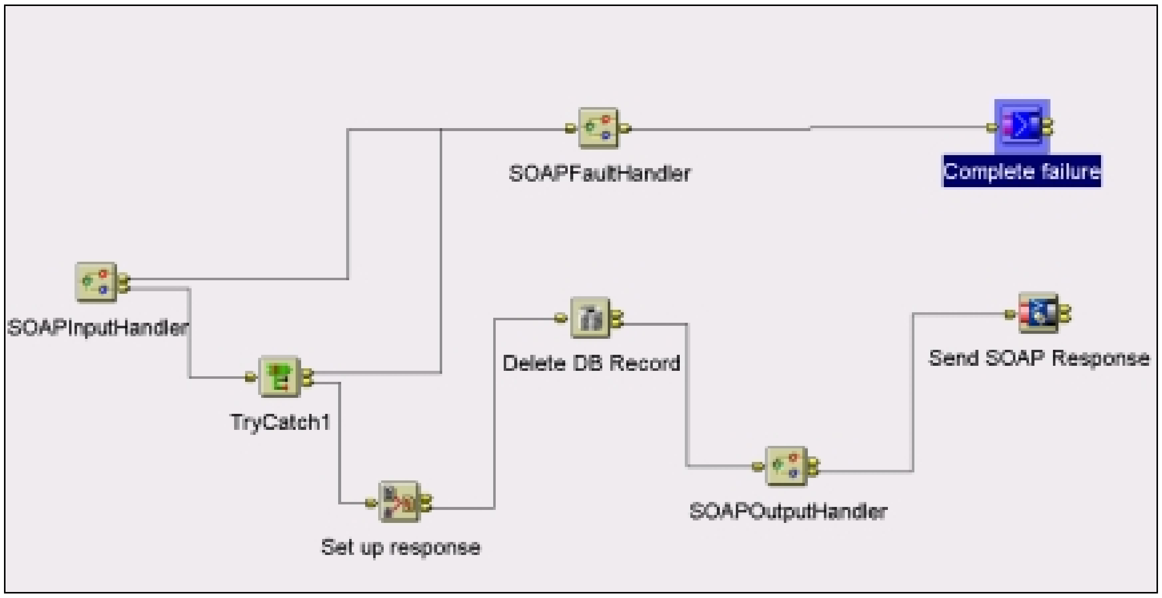


Figure 18 Example 3 Response Message Flow

The Response Message flow receives a SOAP Response from the Web Service called in the Request Message Flow. Using the correlation ID of the response, the Retrieve Status node recovers the state saved from the incoming request, then deleting the record. It uses this state information to set up correct MQMD headers and destination for the response.

The Compute node generates the appropriate response data (setting SOAP Fault values in the Environment if errors are encountered) and passes the data to the SOAPOutputHandler which creates a SOAP response and returns it to the original requestor. All failures are caught first by the SOAPFaultHandler which attempts to return a valid SOAP Fault to the requestor and then by a general WMQI failure node, which in this case simply write a message to the IA81EX1_FAILURE queue.

7 SOAP and Message Flow Implementation

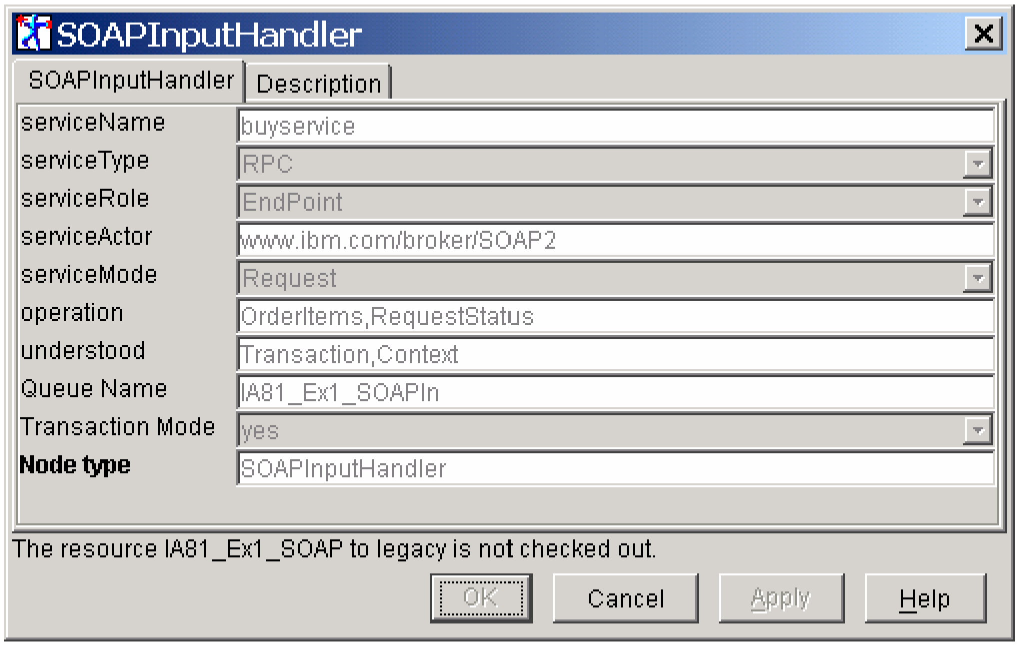
This section outlines the processing requirements for implementing message flows which incorporate the SOAP subflows.

The primary requirement is of course to provide the service required by the requestors. However there are some specific points related to the use of the node.

7.1 Receiving SOAP Requests

First consider the case of a message flow providing a SOAP interface to a legacy service. One message flow is required to decode the SOAP message, process the request and invoke the legacy service. A second message flow is required to receive a response from the legacy service, generate a SOAP response from it and return the SOAP response to the original requestor. Both message flows must be able to detect errors and return a SOAP fault.

RPC Services



serviceName	buyservice
serviceType	RPC
serviceRole	EndPoint
serviceActor	www.ibm.com/broker/SOAP2
serviceMode	Request
operation	OrderItems,RequestStatus
understood	Transaction,Context
Queue Name	IA81_Ex1_SOAPIn
Transaction Mode	yes
Node type	SOAPInputHandler

The resource IA81_Ex1_SOAP to legacy is not checked out.

A service offered by a message flow must be defined by the properties of the SOAPInputHandler. For a service supporting RPC operations the list of operations supported by the message flow must be given, The message flow is expected to process any requests for the given list of operations.

If a service is supporting RPC type requests the SOAPInputHandler presents to the message flow an XML message with top level element ServiceData. This element contains the parameters of the operation being requested. The operation itself is stored in the Environment (Environment.SOAP.Input.Operation). The message flow is required to branch on the operation and invoke the appropriate legacy service to meet the request. The SOAPInputHandler checks that any requested operation is one of those configured and the message flow is then expected support any valid operation.

All the namespace tags are stripped from data passed to the message flow. The namespace definitions are stored in the Environment but there is no requirement on every message flow to process these. They are available for information only.

The following RPC message is shown with its representation after the SOAPInputHandler.

```
<soapenv:Envelope
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <soapenv:Header>
          <Transaction> Type 1 </Transaction>
  </soapenv:Header>
  <soapenv:Body>
    <ns1:Operation xmlns:ns1="http://www.buyservice.com/buy">
      <parm1> 1234567</parm1>
      <parm2>
        <val1>abc</val1>
        <lastName>Smith</lastName>
        <val2>xxx</val2>
     </parm2>
    </ns1:Operation>
  </soapenv:Body>
</soapenv:Envelope>
```

The message flow will receive the parameters of the operation as follows with all other relevant information stored in the Environment (see section 4.2)

```
<ServiceData>
      <parm1> 1234567</parm1>
      <parm2>
        <val1>abc</val1>
        <lastName>Smith</lastName>
        <val2>xxx</val2>
     </parm2>
</ServiceData>
```

DOC Services

Where a message flow supports a ‘DOC’ service type the SOAPInputHandler simply passes it the entire XML message contained within the Body of the SOAP request. The message flow is expected to understand the XML message and process it accordingly.

Processing Header entries

This may be required for either RPC or DOC requests.

The properties defining the service specify which header entries are expected to be understood by message flow, which must review the Header data in the Environment. For any header entry In Environment.SOAP.Header there may be header entries, the SOAPInputHandler processes mustUnderstand and actor attributes together with the service role to determine the action required by the message flow. If the attribute WMQIAction is set to ‘MustProcess’ then the message flow must understand and handle the entry. If the attribute WMQIAction is set to ‘MayProcess’ then the message flow may process the entry but is not required to. If the attribute WMQIAction is set to ‘Forward’ then the message flow must pass the header entry onto the next SOAP service. This occurs only when the message flow is acting as an intermediary receiving SOAP requests and invoking a downstream service with a related SOAP request.

The following ESQL sample shows how the header may be processed by looping through each header entry and finding the WMQIAction. A header entry with WMQIAction set to Forward is simply copied to the Output tree to be picked up by the SOAPOutputHandler. For the other options the action will depend on the nature of the header and the service.

```
DECLARE HRef REFERENCE TO Environment.SOAP.Input;
DECLARE HRefOut REFERENCE TO Environment.SOAP.Output;
DECLARE ActRef REFERENCE TO InputBody;
DECLARE FWD BOOLEAN;
DECLARE HEntry CHAR;
SET FWD = FALSE;
-- Are there any headers to process?
IF Environment.SOAP.Input.Header IS NOT NULL THEN
  -- Go to Header and loop through entries
  MOVE HRef TO Environment.SOAP.Input.Header;
  MOVE HRef FIRSTCHILD TYPE 0x01000000;
  WHILE LASTMOVE (HRef) DO
    -- Loop through attributes checking for WMQIAction
    IF HRef.WMQIAction = 'Forward' THEN
      -- Create Output Header first time
      IF NOT FWD THEN
        CREATE LASTCHILD of Environment.SOAP.Output DOMAIN 'XML' TYPE 0x01000000 NAME 'Header';
        SET FWD = TRUE;
      END IF;
      -- Add Header entry to Output
      CREATE LASTCHILD of Environment.SOAP.Output.Header DOMAIN 'XML' TYPE 0x01000000 NAME
FIELDNAME(HRef);
      MOVE HRefOut TO Environment.SOAP.Output.Header;
      MOVE HRefOut LASTCHILD;
      -- Copy to Output for next service
      SET HRefOut = HRef;
      -- Remove the WMQIAction as it does not apply downstream
      SET HRefOut.WMQIAction = UNKNOWN;
    END IF;
    -- This will be one of a known list. Process it
    IF HRef.WMQIAction = 'MustProcess' THEN
      SET HEntry = FIELDNAME( HRef);
      -- HRef gives the reference to access all data in the Header Entry
    END IF;
    IF HRef.WMQIAction = 'MayProcess' THEN
      -- Check if this is something we know about
      -- Process it if we can
    END IF;
    MOVE HRef NEXTSIBLING;
  END WHILE;
END IF;
```

Reply Addresses

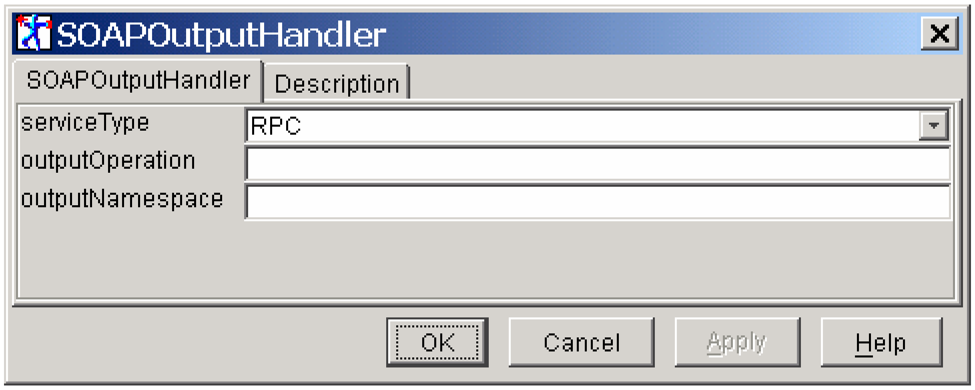
When a message flow invokes a downstream service over MQ/JMS the response is retrieved asynchronously. The reply address of the original requestor must therefore be saved. In the examples this is done using a database save against the MsgId as key and retrieved using correlation ID.

Once the original requestors reply address has been saved it can be overwritten by the reply address which allows the downstream service to reply to the message flow

7.2 Creating SOAP Responses

Now we consider the case where a message flow has invoked a service and received a response. The message flow receiving the response is responsible for creating the data required in the response and passing it to the SOAPOutputHandler to generate the SOAP response. Note that the SOAPOutputHandler does not actually send the message and the SOAP message terminal should be wired to an MQOutput or MQReply node.

This is effectively the reverse of the SOAP request handling covered previously but configuration parameters are simpler. RPC / DOC determines how response data is included in the response. Output Operation is required only for RPC and is included directly in the SOAP response enclosing the parameter(s) found in the message data. The output namespace is used to qualify the operation or the top level tag of a DOC response; it is required for an RPC style SOAP message but is optional for DOC style.



serviceType RPC
outputOperation
outputNamespace

RPC

The SOAPOutputHandler will take response data from either the MRM or XML subtrees of the current message.

For XML the parameters must be set in ServiceData. All children of ServiceData will be included in the SOAP response as children of the specified output operation.

In the MRM case the operation is again used to create the top level element of the response data. The MRM data will be created as a child of the operation with the MessageType used as a top level tag to include the message data. This approach has been taken as it gives a more natural model for the MRM messages but it does mean that only a single parameter is supported for the operation. If multiple parameters are required the data should be set up in the XML domain using service data as a top level tag which is then removed

DOC

With a DOC style response and XML data, the XML document is copied to the SOAP response as a child of the SOAP body element. The namespace from the SOAPOutputHandler properties is added as an attribute and used to qualify the first element of the XML document. The XML data should not be contained in a ServiceData element for DOC style responses. The XML data should be contained within its own top level element which will be included in the response.

With a DOC style response and MRM data the MRM Message type will be used as a top level enclosing element for the XML document created within the SOAP body. This element will be qualified by the specified output namespace.

Headers

The SOAPOutputHandler picks up any Header data in Environment.SOAP.Output.Header and creates a header in the SOAP response containing all the header entries. It is the responsibility of the message flow to create any necessary headers and to set their attributes appropriately. It is not essential to include header data and these facilities should only be used where the service requestor requires additional information on how to handle the response.

7.3 Creating SOAP Requests

The creation of a SOAP request is almost identical to creation of a SOAP response. The main difference is that inclusion of headers will be determined not by the message flow but by the WSDL and requirements of the SOAP service being invoked.

In the case where WMQI acts as a SOAP intermediary it will be the responsibility of the message flow to examine all headers and to copy to Environment.SOAP.Output.Header all entries which have the WMQI action ‘Forward’.

In all cases it is necessary to understand the behaviour of the service being requested and to create in Environment.SOAP.Output.Header any header entries required by the service for correct processing.

7.4 Receiving SOAP Responses

The SOAPInputHandler will process SOAP responses from a SOAP service as well as requests for SOAP services. The processing within the SOAPInputHandler differs only in the handling of fault responses.

The message flow will receive XML data extracted from the SOAP response in the same way as for SOAP requests.

7.5 SOAP Faults

The SOAP handlers will check for correct SOAP formats and will generate the appropriate SOAP fault data in the Environment. Message flows are required only to pass this data unchanged to the SOAPFaultHandler.

It will also be necessary to handle other faults occurring within the message flow. The SOAPFaultHandler can be wired directly to any failure path and will process the WMQI exception tree to give a SOAP fault. This is an adequate way of handling faults in the SOAP environment, Specific faults can be detected and added by throwing a user exception with additional information.

In some cases finer control may be desired. In this case the message flow may directly set the SOAP fault details in Environment.SOAP.Fault. The faultcode should always be ‘soapenv:Server’. The faultstring reflects the information on the error. The message flow should not set soap fault data directly in this way without an understanding of the SOAP protocol. It is recommended that a user exception be thrown allowing the SOAPFaultHandler to create the SOAP elements unless a finer level of control is required.

8 Description of Deliverables

This SupportPac is issued as a single zip file, IA81.zip. The directory structure is as follows:

Directory IA81 contains this document and the following directories:

DB Setup	Text file to create the database table used in exercises to save reply data
Example 1	Messages used in Example 1
Example2	Messages used in Example 2
Flows and Msg Sets	Export files for message flows and the message set used in the example The export files contain the SOAP subflows which support the invocation of SOAP services from WMQI and the processing of SOAP requests by WMQI.
Legacy	Contains the simple application which simulates a legacy application providing services to WMQI
SOAPhttp to SOAP JMS Axis gateway	Contains the documentation and code for installing and running a SOAP http to SOAP JMS protocol conversion.
WSDLplugin	Contains the plug in to specify the service configuration and remove name space qualifications from an incoming SOAP message.

8.1 Initial set up

Create the database IA81SOAP and register for ODBC access. Create a schema IA81 and a table within it that matches the description given in the …/IA81/DB Setup/Createtable.txt.

Import the message set in …/IA81/Flows and MsgSets/IA81exportmsgset.mrp into your MRM database using the MQSIimpexpmsgset command

Set up the WSDL plug in from …/IA81/WSDLplugin:

- copy all 5 gif files to <wmqi_root>\Tool\image\
- import WSDLSettings.xml into your workspace (File->Import to Workspace for Message Flows tab)
- copy WSDLSettings.jar to <wmqi_root>\jplugins
- add <wmqi_root>\classes\jplugin.jar to your CLASSPATH

Import the SOAP subflows and examples from …/IA81/ Flows and MsgSets/SOAP.xml

The examples can now be set up and run as described in section 6 Sample Implementations.

The SOAP/ http to SOAP/JMS Axis gateway is not required to run the examples but can be used to extend them to run over http. The instructions for installing and running the gateway are included in the separate document …/IA81/ SOAPhttp to SOAP JMS Axis gateway/ IA81Gateway.doc.

9 SOAP

This appendix outlines some of the major aspects of SOAP. It is not meant to be definitive, nor does it cover all aspects of SOAP. It assumes little knowledge of SOAP, and only a pragmatic level of knowledge of XML. It draws heavily on the SOAP V1.2 Primer, which can be found at http://www.w3.org/TR/soap12-part0.

It is meant to give sufficient knowledge of SOAP so that someone using this SupportPac can understand the various choices that we have made during the preparation of the SupportPac, as well as their main implications.

In this SupportPac we limited ourselves to SOAP 1.1, which is the most popular current implementation of SOAP. There are some differences noted in the text of this appendix. For a complete list of differences, see *"5. Changes between SOAP 1.1 and SOAP 1.2"* in the SOAP 1.2 Primer referred to above.

9.1 Overview

SOAP is a means for making it easier for applications to communicate over the web.

A SOAP message is fundamentally a one-way transmission between SOAP nodes, from a SOAP sender to a SOAP receiver.

The SOAP message passes over a SOAP message path.

A SOAP message path is the set of SOAP nodes through which a single SOAP message passes. This includes the initial SOAP sender, zero or more SOAP intermediaries, and an ultimate SOAP receiver.

Each SOAP node in a message path MUST process the message. The details of the role that each SOAP node assumes are not defined by the SOAP specification. They are defined as a part of the overall application semantics and associated message flow.

The ultimate recipient of a SOAP message sent from the sender is the receiver. Intermediate nodes may act in some way on the message, for example by logging, auditing or, possibly, amending the message.

SOAP messages are expected to be combined by applications to implement more complex interaction patterns ranging from request/response to multiple, back-and-forth "conversational" exchanges.

These exchanges will be between service requestors and service providers.

SOAP defines three aspects of the format of the messages. Note that use of each of these is optional:

- The SOAP envelope construct defines an overall framework for expressing **what** is in a message; **who** should deal with it, and **whether** it is optional or mandatory.
- The SOAP encoding defines a serialization mechanism that can be used to exchange instances of application-defined datatypes.
- The SOAP RPC representation defines a convention that can be used to represent remote procedure calls and responses.

(In this document, it is assumed that all SOAP messages conform to the SOAP envelope specification.)

Most aspects of the message and its processing are defined by the application. SOAP is silent on the semantics of any application-specific data it conveys, as it is on issues such as the routing of SOAP messages, reliable data transfer, firewall traversal, etc. However, SOAP provides the framework by which application-specific information may be conveyed in an extensible manner.

If SOAP defined elements *are* included within a SOAP message, SOAP provides a full description of the actions required to be taken by a SOAP processor on receiving the message; if a SOAP defined element is included, it MUST be processed according to SOAP rules.

9.2 SOAP Message Format

The following is used to illustrate various points about the message format:

```
<soapenv:Envelope
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <env:Header xmlns:env="http://www.w3.org/2001/06/soap-envelope">
    <t:Transaction
         xmlns:t=http://example.org/2001/06/tx
         env:mustUnderstand="1">
      5
    </t:Transaction>
  </env:Header>
  <soapenv:Body>
    <ns1:buy xmlns:ns1="http://www.buyservice.com/buy">
      <buyReq>
        <toBuy>
          <symbol>WSDL</symbol>
          <amount>5678</amount>
        </toBuy>
        <id>
          <firstName>John</firstName>
          <lastName>Doe</lastName>
        </id>
      </buyReq>
    </ns1:buy>
  </soapenv:Body>
</soapenv:Envelope>
```

A SOAP message is an XML document that consists of a mandatory SOAP envelope, an optional SOAP header, and a mandatory SOAP body.

This message implements the SOAP envelope with a header.

SOAP 1.1 and SOAP 1.2 are distinguished by the namespaces used for the SOAP envelope, as follows:
SOAP 1.1: `xmlns:soapenv="`**`http://schemas.xmlsoap.org/soap/envelope/`**`"`
SOAP 1.2: `xmlns:soapenv="`**`http://www.w3.org/2001/09/soap-envelope/`**`"`
Any other namespace is considered an error.

SOAP Encoding describes how to encode instances of data that conform to the data model described in the SOAP Data Model. This encoding MAY be used to transmit data in SOAP header blocks and/or SOAP bodies. Other data models, alternate encodings of the SOAP Data Model as well as unencoded data MAY also be used in SOAP messages.

9.3 SOAP Encoding

There are two main options for encoding the body of the message: literal encoding and SOAP encoding. This is specified with the encodingStyle attribute, which is used to specify the URI identifying the serialization rule or rules that can be used to deserialize the SOAP message. You can use a blank URI (="").

SOAP encoding means that the message conforms to a set of rules based on the XML schema datatypes. It is specified by specifying the encodingStyle attribute as follows:

SOAP 1.1: `<soapenv:`**`Envelope`**
`soapenv:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/`

SOAP 1.2: `<soapenv:`**`Envelope`**
`soapenv:encodingStyle=http://www.w3.org/2001/09/soap-encoding/`

You are not required to use a SOAP encoding style, and not using the SOAP encoding style is referred to as literal XML encoding.

The following are message fragments of the same data encoded in the two different ways (taken from Cerami, pp 59-60).

SOAP encoding:

```
<ns1:getProductResponse
  xmlns:ns1="urn:examples:productservice
  soapenv:encodingStyle="http://www.w3.org/2001/09/soap-encoding/"
  <return xmlns:ns2="urn:examples" xsi:type="ns2:product">
    <name xsi:type="xsd:string">Red Hat Linux</name>
    <price xsi:type="xsd:double">54.99</price>
    <description xsi:type="xsd:string">
      Red Hat linux Operating System
    </description>
    <SKU xsi:type="xsd:string">A358185</SKU>
  </return>
</ns1:getProductResponse>
```

Literal encoding:

```
<ns1:getProductResponse
  xmlns:ns1="urn:examples:XMLproductservice
  soapenv:encodingStyle="http://xml.apache.org/xml-soap/literalxml/"
  <return>
    <product sku="A358185">
      <name>Red Hat Linux</name>
      <price>54.99</price>
      <description>Red Hat linux Operating System</description>
    </product>
  </return>
```

If the same data appears multiple times in a message, use of SOAP encoding requires the use of multiple reference. Single reference values may also be encoded using this technique. These are illustrated here:

```
<soapenv:Header>
  <id href="#id0"/>
</soapenv:Header>
```

```
<soapenv:Body>
  <ns1:buy xmlns:ns1="http://www.buyservice.com/buy">
    <buyReq href="#id1"/>
  </ns1:buy>

  <multiRef id="id0" SOAP-ENC:root="0"
      soapenv:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
      xsi:type="ns2:toBuyType"
      xmlns:ns2="http://www.buyservice.com/buy">
    <firstName xsi:type="xsd:string">Shahryar</firstName>
    <lastName xsi:type="xsd:string">Sedghi</lastName>
  </multiRef>

  <multiRef id="id1" SOAP-ENC:root="0"
      soapenv:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
      xsi:type="ns3:buyReqType"
      xmlns:ns3="http://www.buyservice.com/buy">
    <toBuy href="#id2"/>
    <id href="#id0"/>
  </multiRef>

  <multiRef id="id2" SOAP-ENC:root="0"
      soapenv:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
      xsi:type="ns4:idType"
      xmlns:ns4="http://www.buyservice.com/buy">
   <symbol xsi:type="xsd:string">WSDL</symbol>
   <amount xsi:type="xsd:int">5678</amount>
  </multiRef>
 </soapenv:Body>
```

In the example above, instead of having two instances of the elements <firstName> and <lastName>, only one instance is included with references where needed. (Note that many XML parsers cannot currently handle this level of XML.)

9.4 SOAP Style

There are two ways to structure a SOAP document: document style and RPC style. (Initially, the SOAP specification defined only the RPC style.)

RPC style is more formal in that the invocation is represented by a single structure or array containing an element for each parameter. The structure or array is named identically to the procedure or method name. For instance, the example shown above (simplified below) might be the SOAP message sent corresponding to the call: `buy(toBuy, firstName, lastName)`

```
<soap:Body>
  <buy>
    <toBuy>
      <symbol>WSDL</symbol>
      <amount>5678</amount>
    </toBuy>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
  </buy>
</soap:Body>
```

Note that the only child of the <Body> element has the name of the method to be invoked; the parameters are, in turn, children of that element.

Optionally, the method can also be carried in the HTTP header in the SOAPAction field. Note that the presence of the SOAPAction header is mandatory; it is optional for it to carry a value. Some SOAP implementations require that this field contain a value.

There is no requirement to correlate the names of the request and response, although a common standard is to add Response to the request name, so the response to the above request might be:

```
<soap:Body>
  <buyResponse>
    .......
  </buyResponse>
</soap:Body>
```

With document style the Body of the message can be structured in any way that suits the requirements of the application.

Within a series of SOAP interactions, the different SOAP styles and encoding can be mixed to suit the requirements of the application. The example in the SOAP Primer uses literal encoding and Document style to negotiate the details of a flight reservation. To finalize the reservation, SOAP encoding with RPC style is used.

Typically, literal encoding would be used with Document style, and SOAP encoding with RPC style. The choice of which is used in any particular instance is an application design decision.

The SOAP requestor application needs to understand the requirements of the server application, in terms of style and encoding. This can be made available to the application developer in by encapsulating the requirements in a WSDL document.

9.5 SOAP Faults

In the event of an error, applications should return a Fault element in the Body element. The Fault element should be the only child of Body. The following is used to illustrate some aspects:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  <soapenv:Body>
    <soapenv:Fault>
      <faultcode>soapenv:Server</faultcode>
      <faultstring>Server Error</faultstring>
      <faultactor>WMQIservice1</faultactor>
      <detail>
        <Message>The request for service servicename failed.</Message>
      </detail>
    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

A SOAP fault message contains a mandatory *faultcode* element, and a mandatory *faultstring* element, both children of the Fault element.

The message can also contain two optional elements: *faultactor* and *detail*. The faultactor element is used to indicate the node where the Fault was generated and the detail element is used to carry application specific error information related to the SOAP Body.

SOAP 1.1 defines the following faults:

VersionMismatch	The SOAP Envelope included an invalid namespace for the SOAP Envelope element
MustUnderstand	The node signifying the fault could not process a SOAP Header element with a SOAP mustUnderstand attribute value of “1”.
Client	The message was incorrectly formed or did not contain the appropriate information in order to succeed. For example, the message contained an invalid method name or parameter information. It is generally an indication that the message is not to be resent without being changed.
Server	The message could not be processed for reasons attributable to the processing of the message rather than to the contents of the message itself. For example, processing could include communicating with an upstream SOAP node, which did not respond. The message may succeed if resent at a later point in time.

SOAP 1.2 defines an additional error, DataEncodingUnknown, where a header or body targeted at the faulting SOAP node has an encodingStyle attribute that the faulting node does not support.

SOAP 1.2 also renames the various elements (Code, Reason Code, Role, Detail respectively) and adds a new element, Node.

9.6 mustUnderstand Attribute

The mustUnderstand attribute can be applied to any element in the Header element:

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header>
    <abc:Extension1 xmlns:abc="http://example.org/2001/06/ext"
      env:mustUnderstand="1" />
    <def:Extension2 xmlns:def='http://example.com/stuff'
      env:mustUnderstand="1"/>
  </env:Header>
  <env:Body>
      . . .
  </env:Body>
</env:Envelope>
```

The mustUnderstand attribute can be set to "1" or "0". Omitting it is defined as being semantically equivalent to including it with a value of "0". It is recommended that requestors omit this attribute rather than set it to "0".

SOAP 1.2 extends the allowable values to include "true" and "false".

SOAP header blocks with a mustUnderstand attribute are presumed to somehow modify the semantics of other headers or body elements. Therefore for mandatory elements targeted to a node, that node MUST either process the header block, or not process the SOAP message at all and instead generate a fault. The mustUnderstand attribute thus assures that such modifications will not be silently (and, presumably, erroneously) ignored by a SOAP node to which the header block is targeted.

The mustUnderstand attribute information item is not intended as a mechanism for detecting errors in routing, misidentification of nodes, failure of a node to serve in its intended role(s), etc. Any of these conditions can result in a failure to even attempt processing of a given SOAP header block from a SOAP envelope. This specification therefore does not require any fault to be generated based on the presence or value of the mustUnderstand attribute information item on a SOAP header block not targeted at the current processing node. In particular, it is not an error for an ultimate SOAP receiver to receive a message containing a mandatory header block that is targeted at a role other than the ones assumed by the ultimate SOAP receiver. This is the case, for example, when a header block has survived erroneously due to a routing or targeting error at a preceding intermediary.

9.7 Actors and Roles

The terms Actor (SOAP 1.1) and Role (SOAP 1.2) are essentially equivalent.

A SOAP message travels from the originator to the ultimate destination, potentially by passing through a set of SOAP nodes along the message path. An intermediary node is one that is capable of both receiving and forwarding SOAP messages. These nodes are the “actors” that process the message.

Not all parts of a SOAP message may be intended for the ultimate destination of the SOAP message. Some parts may, instead, be intended for one or more of the intermediate nodes on the path. The role of a recipient of a header element is similar to that of accepting a contract in that it cannot be extended beyond the recipient. That is, a node receiving a header element MUST NOT forward that header element to the next node in the SOAP message path. The recipient MAY insert a similar header element but in that case, the contract is between that application and the recipient of that header element.

The SOAP actor global attribute can be used to indicate the recipient of a header element. The value of the SOAP actor attribute is a URI. The special URI
http://schemas.xmlsoap.org/soap/actor/next
indicates that the header element is intended for the next SOAP application that processes the message.

Omitting the SOAP actor attribute indicates that the recipient is the ultimate destination of the SOAP message.

SOAP 1.2 redefines some of these functions, but the principles remain the same.

10 WSDL

This appendix outlines some of the major aspects of WSDL - Web Services Description Language. It is not meant to be definitive, nor does it cover all aspects of WSDL. It assumes little knowledge of WSDL, and only a pragmatic level of knowledge of XML. It draws on the WSDL description published by W3C, which can be found at http://www.w3.org/TR/wsdl.

It is meant to give sufficient knowledge of WSDL so that someone using this SupportPac can understand the various choices that we have made during the preparation of the SupportPac, as well as their main implications.

10.1.1 Overview

The objective of WSDL is to encapsulate in one XML document all the information necessary to allow a requestor application to invoke a web service. The WSDL document would normally be used by an application development tool as part of the building of a requestor application.

A WSDL document is an XML document, containing a set of definitions, which define all the interfaces necessary for an application to invoke a web service.

These interfaces include the messages to be sent and received; data type information for those messages; the functions that can be invoked; how faults are to be handled, the transport protocol to be used; and address information for locating the service.

A WSDL document typically describes a set of services, together with their interfaces.

10.1.2 WSDL Document Contents

The following is a WSDL document used to illustrate the points made later in this section. It is not meant to be totally accurate in all particulars.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://www.buyservice.com/buy"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:impl="http://www.buyservice.com/buy-impl"
             xmlns:intf="http://www.buyservice.com/buy"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <import location="C:/Webservices/buyStock.xsd"
          namespace="http://www.buyservice.com/buy"/>

  <message name="buyResponse">
    <part name="return" type="intf:buyRespType"/>
  </message>
  <message name="buyRequest">
    <part name="buyReq" type="intf:buyReqType"/>
  </message>
  <message name="headerMessage">
    <part name="id" type="intf:idType"/>
  </message>

  <portType name="BuyService">
    <operation name="buy">
      <input message="intf:buyRequest"/>
      <output message="intf:buyResponse"/>
      <fault message=="intf:faultResponse"/>
    </operation>
  </portType>

  <binding name="buyServiceSoapBinding" type="intf:BuyService">
    <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="buy">
      <soap:operation soapAction=""/>
      <input>
        <soap:body message="intf:headerMessage"
          part="id"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://www.buyservice.com/buy"
          use="encoded"/>
        <soap:header use="encoded"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://www.buyservice.com/buy"/>
      </input>
      <output>
        <soap:body namespace=http://www.buyservice.com/buy
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          use="encoded"/>
      </output>
    </operation>
  </binding>

  <service name="buyServiceService">
    <port binding="intf:buyServiceSoapBinding" name="JMSPortJNDI">
      <soap:address
        location="jms://axisJms?requestQueue=httprDemoReq&amp;
        responseQueue=httprDemoResp"/>
    </port>
  </service>
</definitions>
```

A WSDL document is simply a set of definitions. There is a **definitions** element as the root, with definitions inside. There are six major elements used to define a service. The first three define abstract aspects of the service:

- **types**, which provides abstract data type definitions used to describe the messages exchanged
- **message**, which represents an abstract definition of the data being transmitted. A message consists of logical parts, each of which is associated with a definition within some type system
- **portType**, which defines a set of abstract operations and their associated messages.
    -
    - The next three define concrete aspects of the service:
    -
- **binding**, which specifies concrete protocol and data format operation refers to an input message and output messages specifications for the operations and messages defined by a particular portType
- **port**, which specifies an address for a binding, thus defining a single communication endpoint

- **service**, by being the parent of **port**, is used to aggregate a set of related ports.

Each of the elements refers, explicitly (by name) or implicitly (by being the parent), to one or more instances of the immediately preceding element, adding further information.

Use of the **import** element allows the separation of the different elements of a service definition into independent documents, which can then be imported as needed. This technique helps writing clearer service definitions, by separating the definitions according to their level of abstraction. It also maximizes the ability to reuse service definitions of all kinds. As a result, WSDL documents structured in this way are easier to use and maintain.

The **types** element encloses data type definitions that are relevant for the exchanged messages. WSDL prefers the use of XSD as the type system, although other definitions can be used.

The **message** element is used to define the various messages (input, output, header) that are recognised by the set of web services defined in this WSDL document. Message definitions are always considered to be an abstract definition of the message content. A message binding describes how the abstract content is mapped into a concrete format.

The **portType** is a named set of abstract operations and their associated abstract messages. Multiple portType elements are allowed within one WSDL document. If the operation defines a one-way service, then only an input message will be defined, if it is a request-response operation, then both input and output messages will be defined, as well as any optional fault elements, which specify the abstract message format for any error messages that may be output. (WSDL also defines ‘solicit-response’ and ‘notification’ but currently defines no bindings for these transmission styles.) The various services are distinguished by the existence and sequence of the input and output elements. Input only indicates one-way, input followed by output indicates request-response. The operation only specifies the request method, not the response.

The portType can be considered to be a class, and the various operations as methods of that class.

The **binding** element defines how the data will be transmitted over the wire. It specifies one protocol, and the associated portType elements. A given portType may be referenced by several bindings (e.g. to use SOAP HTTP and SOAP SMTP).

WSDL includes a binding for SOAP which supports the specification of SOAP specific information, including:

- The style (in the example, this is RPC style)
- The encoding (in the example this is SOAP Encoding)
- The SOAPAction HTTP header (optional) for the HTTP binding of SOAP
- Definitions of optional headers transmitted as part of the SOAP Envelope.
    -
    - Note that some of the information provided in the binding element has to be carried in the HTTP header rather than in the SOAP message itself. For more information see the HTTP Appendix ********CROSS REFERENCE********
    -

Other bindings can be specified and implemented, for example, AXIS includes and MQSeries binding.

The **port** element defines an individual endpoint by specifying a single address for a binding.

The **service** element is used to group a set of related ports.

10.1.3 Summary

A WSDL document defines a set of web services and how they are to be accessed. To do this it needs to define the messages that can be sent and received; this is done using **types** and **message** elements. Individual services, with their associated input and output messages are defined and grouped together with **portType** elements. The portType can be thought of as a class, and the individual services as methods of that class. The **binding** element defines *how* (transport protocol etc.) the message is sent, the **port** element is used to define *where* it is sent

11 Glossary

AXIS – Apache eXtensible Interaction System – is an implementation of SOAP for Java on the Apache platform. It can be downloaded from http://www.apache.org.

CWF – Custom Wire Format – is a wire (message) format supported by the *MRM* that describes messages used by C or COBOL applications.

document style see message style.

DTD Document Type Description. A means of defining the format and contents of an XML document. These have now been superseded by schemas.

Eclipse A kind of universal tool platform. Full details are available from http://www.eclipse.org/

encoding style In SOAP, the encoding style refers to the way the format of the contents, both header and body, of the SOAP message are defined. This can either be SOAP encoding, where the message conforms to a built in set of rules for encoding data, or literal encoding, where you can embed an XML document into the message. These can be distinguished by using the SOAP-ENV:encodingStyle namespace.

fire & forget A style of application design where one application invokes a service of another, but does not expect a reply. In an asynchronous environment, this typically requires some form of assured message delivery such as IBM's MQSeries. This model can be implemented in a web services environment by defining a request message without a corresponding response message.

JMS – Java Message Service – is an API defined by Sun as part of the Java specification. IBM has implemented the API, using MQSeries to provide the actual message transport.

literal encoding see encoding style.

message flow In this document, a message flow is effectively an application running under the control of IBM's WMQI. A message flow typically transforms input data (usually in an MQSeries message) from a source format (for example, an SAP iDoc) to a target format (perhaps a S.W.I.F.T. format) so that it can be understood by a target application. Additional functions such as database lookup and update, and message publishing, can also be included in a message flow.

message style This refers to the style of message exchange between the service requestor and service provider. The styles currently defined are document style and RPC style. There are no hard and fast rules as to where each style should be used, and either can be used with either encoding style. Essentially, RPC style matches an RPC call, where the exchanged messages conform to a well-defined signature for the remote call and its return. The only child element of the SOAP body is the method to be invoked, with the parameters of the method call being, in turn, the children of that element. Document style allows for more flexible message formats. For example, document style might be used while deciding the details of a flight reservation such as airport, date, etc., with RPC style being used for the final reservation.

MQ See MQSeries

MQ Internet Passthru A WebSphere MQ base product extension (SupportPac MS81) that can be used to implement messaging solutions between remote sites across the Internet. It makes the passage of WebSphere MQ channel protocols into and out of a firewall simpler and more manageable by tunnelling the protocols inside HTTP or by acting as a proxy.

MQI The base MQSeries API. There are Java classes available that allow Java applications to invoke this API, and JMS in MQSeries is implemented as a set of classes that, in turn, invoke the Java classes.

MQMD MQSeries Message Descriptor The MQSeries message header, containing data that describes the contents of the MQSeries message.

MQRFH2 – Rules and Formatting Header Version 2 – is an optional extra MQ header, supported by MQSeries and used by WMQI, and for message passing between MQSeries implementations of JMS.

MQSeries Messaging and Queuing. An IBM MoM product. The latest version is called WebSphere MQ.

MQSI MQSeries Integrator (see WMQI)

MRM Message Repository Manager. The part of WMQI where message definitions are stored.

namespace See XML namespace

Notification One of the four WSDL transmission types. The requestor expects a message from the service provider, with no reply. Note: WSDL currently defines no bindings for this transmission style.

One-way One of the four WSDL transmission types. The requestor sends a message to the service provider, with no reply.

provider see service provider

requestor see service requestor

request/reply A style of application design where one application invokes a service of another, expecting a reply. This can be implemented using either a synchronous or an asynchronous communication protocol.

Request-response One of the four WSDL transmission types. The requestor sends a message to the service provider, and receives a correlated reply.

RFH2 see MQRFH2.

RPC style see message style.

schema These provide a means for defining the structure, content and semantics of XML documents. See http://www.w3.org/XML/Schema.

service provider Any system that provides a service and makes it available to service requestors.

service registry A directory that contains details of services. In a SOAP environment, UDDI will be used to interrogate the directory in order to discover services, and WSDL will be used to describe the services, and how they are accessed.

service requestor The consumer or user of a service (provided by a service provider) in a *web services* environment.

SOA – Service Oriented Architecture – a method of designing and building loosely coupled software solutions that expose business functions as programmatically accessible software services for use by other applications through published and discoverable interfaces.

SOAP (originally: Simple Object Access Protocol, but because the protocol actually has nothing to do with objects, the W3C were unhappy with the name. However, as the term "SOAP" is well established, the W3C now states that SOAP does not actually stand for anything! – Cerami, p. 63) An XML based protocol for the exchange of business information between application programs.

SOAP Body The element in a SOAP message that contains the payload, including the service to be invoked, parameters or returned data. It may optionally contain a SOAP Fault element.

SOAP encoding see encoding style.

SOAP Envelope This is the wrapper for a SOAP message. It contains an optional SOAP Header and a SOAP Body element.

SOAP Fault An optional element of a SOAP Body that allows a service to return an error to the service requestor.

SOAP Header This is an optional SOAP message element allowing the specification of additional application requirements.

SOAP JMS Message In this document, this is used to indicate a SOAP message carried over a WebSphere MQ JMS messaging service.

SOAP web service see web service.

Solicit-response One of the four WSDL transmission types. The requestor expects a message from the service provider, and will send a reply. Note: WSDL currently defines no bindings for this transmission style.

SupportPacs These are packages of material, such as code or documentation, which complement various IBM products. SupportPacs for the WebSphere MQ family of products are available from:
http://www-3.ibm.com/software/ts/mqseries/txppacs/txpsumm.html

UDDI – Universal Description, Discovery and Integration – is a specification for distributed Web-based information registries of Web services, UDDI is also a publicly accessible set of implementations of the specification that allow businesses to register information about the Web services they offer so that other businesses can find them.

URI Uniform Resource Identifier

URN Uniform Resource Name

WAS WebSphere Application Server

WBC WebSphere Business Connect

WBI WebSphere Business Integrator

web server A web server is a service running in a computer that accepts and processes HTTP requests. Examples include the Apache web server, Netscape, and the IBM HTTP server. Typically, it services HTML requests, and passes on servlet or other requests to other functions within the processor. For example, servlet requests might go to the Apache servlet engine and requests for EJBs to the IBM WebSphere Application Server.

web service The definition agreed upon by the W3C is:

> A web service is a software application identified by a URI whose interfaces and binding are capable of being defined, described, and discovered by XML artefacts, and which supports direct interactions with other software applications using XML-based messages via Internet-based protocols.

Note that the only ‘standard’ mentioned is XML; there is no mention of SOAP, nor of HTTP.

By extension, therefore, a SOAP web service is a web service that conforms to the SOAP protocol. A SOAP web service may additionally be defined with WSDL semantics, and discoverable using UDDI protocols.

WebSphere MQ JMS A set of Java classes that implement Sun’s JMS interfaces; with both the point-to-point and publish/subscribe models being supported, which build on the WebSphere MQ classes for Java. They enable JMS programs to get and put messages using WebSphere MQ systems, allowing them to communicate with WebSphere MQ JMS applications as well as WebSphere MQ applications that do not use the JMS API, such as those written in Java, C, C++, or COBOL.

WMQI WebSphere MQ Integrator

WSAD/IE WebSphere Studio Application Development Integration Edition

WSDE Web Services Development Environment

WSDK WebSphere SDK (Software Development Kit) for Web Services

WSDL – Web Services Description Language – is an XML format that describes web services available over the Internet. For a web service, the WSDL defines, most importantly: the functions or operations available; the message formats expected, and returned, by the service; the transport used to invoke the service (e.g. HTTP POST or SOAP); and where the service is located.

WSGW – Web Services Gateway – is an IBM product that can be used to transform a web service request from one format into another.

WSIF – Web Services Invocation Framework – is a framework or API created by IBM that allows an application programmer to generate SOAP requests, without having to write any SOAP specific code.

WSTK Web Services Tool Kit

XMI – XML Metadata Interchange Format – specifies an open information interchange model that gives developers working with object technology the ability to exchange programming data over the Internet in a standardized way.

XML namespace These provide a simple method for qualifying element and attribute names used in XML documents by associating them with namespaces identified by URI references.

XML-RPC – XML Remote Procedure Call – is a specification that allows programs to make function calls over the web. See http://xml-rpc.com/

XSL Extensible Stylesheet Language

XSLT XSL Transformations

.NET A set of Microsoft software technologies for connecting information, people, systems, and devices. See http://www.microsoft.com/net/