The Java API for XML Based RPC (JAX-RPC) 2.0

Editors Draft
February 25, 2004

Editors:
Marc Hadley
Roberto Chinnici

Comments to: jaxrpc-spec-comments@sun.com

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 USA

Java(TM) API for XML based Remote Procedure Call (JAX-RPC) 2.0 Specification ("Specification")
Version: 2.0
Status: editors copy
Release: February 25, 2004

Contents

Chapter 1

Introduction

A remote procedure call (RPC) mechanism allows a client to invoke the methods of a remote service using a familiar local procedure call paradigm. On the client, the RPC infrastructure manages the task of converting the local procedure call arguments into some standard request representation, communicating the request to the remote service and converting any response back into procedure call return values. On the server, the RPC infrastructure manages the task of converting incoming requests into local procedure calls, converting the result of local procedure calls into responses and communicating responses to the client.

XML[1] is a platform-independent means of representing structured information. XML based RPC mechanisms use XML for the representation of RPC requests and responses and inherit XML's platform independence. SOAP[2, 3, 4] describes one such XML based RPC mechanism and "defines, using XML technologies, an extensible messaging framework containing a message construct that can be exchanged over a variety of underlying protocols".

WSDL[5] is "an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information". WSDL can be considered the defacto interface definition language(IDL) for XML based RPC.

JAX-RPC 1.0[6] defines APIs and conventions for supporting XML based RPC in the Java™ platform. JAX-RPC 1.1[7] adds support for the WS-I Basic Profile 1.0[8] to improve interoperability between JAX-RPC implementations and with services implemented using other technologies.

JAX-RPC 2.0 (this specification) supersedes JAX-RPC 1.1, extending it as described in the following sections.

1.1 Goals

Since the release of JAX-RPC 1.0[6], new specifications and new versions of the standards it depends on have been released. JAX-RPC 2.0 relates to these specifications and standards as follows:

JAXB Due primarily to scheduling concerns, JAX-RPC 1.0 defined its own data binding facilities. With the release of JAXB 1.0[9] there is no reason to maintain two separate sets of XML mapping rules in the Java™ platform. JAX-RPC 2.0 will delegate data binding-related tasks to the JAXB 2.0[10] specification that is being developed in parallel with JAX-RPC 2.0.

JAXB 2.0[10] will add support for Java to XML mapping, additional support for less used XML schema constructs and provide bidirectional customization of Java ⇔ XML data binding. JAX-

RPC 2.0 will allow full use of JAXB provided facilities including binding customization and optional schema validation.

SOAP 1.2 Whilst SOAP 1.1 is still widely deployed, it's expected that services will migrate to SOAP 1.2[3, 4] now that it is a W3C Recommendation. JAX-RPC 2.0 will add support for SOAP 1.2 whilst requiring continued support for SOAP 1.1.

WSDL 1.2 The W3C is expected to progress WSDL 1.2[11] to Recommendation during the lifetime of this JSR. JAX-RPC 2.0 will add support for WSDL 1.2 whilst requiring continued support for WSDL 1.1.

WS-I Basic Profile 1.1 JAX-RPC 1.1 added support for WS-I Basic Profile 1.0. WS-I Basic Profile 1.1 is expected to supersede 1.0 during the lifetime of this JSR and JAX-RPC 2.0 will add support for the additional clarifications it provides.

A Metadata Facility for the Java Programming Language (JSR 175) JAX-RPC 2.0 will use Java annotations[12] to simplify the most common development scenarios for both clients and servers.

Web Services Metadata for the Java™ Platform (JSR 181) JAX-RPC 2.0 will align with and complement the annotations defined by JSR 181[13].

Implementing Enterprise Web Services (JSR 109) The JSR 109[14] defined `jaxrpc-mapping-info` deployment descriptor provides deployment time Java ⇔ WSDL mapping functionality. In conjunction with JSR 181[13], JAX-RPC 2.0 will complement this mapping functionality with development time Java annotations that control Java ⇔ WSDL mapping.

Web Services Security (JSR 183) JAX-RPC 2.0 will align with and complement the security APIs defined by JSR 183[15].

JAX-RPC 2.0 will improve support for document/message centric usage:

Asynchrony JAX-RPC 2.0 will add support for client side asynchronous operations.

Non-HTTP Transports JAX-RPC 2.0 will improve the separation between the XML based RPC framework and the underlying transport mechanism to simplify use of JAX-RPC with non-HTTP transports.

Message Access JAX-RPC 2.0 will simplify client and service access to the messages underlying an exchange.

Session Management JAX-RPC 1.1 session management capabilities are tied to HTTP. JAX-RPC 2.0 will add support for message based session management.

JAX-RPC 2.0 will also address issues that have arisen with experience of implementing and using JAX-RPC 1.0:

Inclusion in J2SE JAX-RPC 2.0 will prepare JAX-RPC for inclusion in a future version of J2SE. Application portability is a key requirement and JAX-RPC 2.0 will define mechanisms to produce fully portable clients.

Handlers JAX-RPC 2.0 will simplify the development of handlers and will provide a mechanism to allow handlers to collaborate with service clients and service endpoint implementations.

Versioning and Evolution of Web Services JAX-RPC 2.0 will describe techniques and mechanisms to ease the burden on developers when creating new versions of existing services.

Backwards Compatibility of Binary Artifacts JAX-RPC 2.0 will not preclude preservation of binary compatibility between JAX-RPC 1.x and 2.0 implementation runtimes.

1.2 Non-Goals

The following are non-goals:

Pluggable data binding JAX-RPC 2.0 will defer data binding to JAXB[10], it is not a goal to provide a plug-in API to allow other types of data binding technologies to be used in place of JAXB. However, JAX-RPC 2.0 will maintain the capability to selectively disable data binding to provide an XML based fragment suitable for use as input to alternative data binding technologies.

SOAP Encoding Support Use of the SOAP encoding is essentially deprecated in the web services community, e.g. the WS-I Basic Profile[8] excludes SOAP encoding. Instead, literal usage is preferred, either in the RPC or document style.

SOAP 1.1 encoding is supported in JAX-RPC 1.0 and 1.1 but its support in JAX-RPC 2.0 runs counter to the goal of delegation of data binding to JAXB. Therefore JAX-RPC 2.0 will make support for SOAP 1.1 encoding optional and defer description of it to JAX-RPC 1.1.

Support for the SOAP 1.2 Encoding[4] is optional in SOAP 1.2 and JAX-RPC 2.0 will not add support for SOAP 1.2 encoding.

Backwards Compatibility of Generated Artifacts JAX-RPC 1.0 and JAXB 1.0 bind XML to Java in different ways. Generating source code that works with unmodified JAX-RPC 1.x client source code is not a goal.

Support for Java versions prior to J2SE 1.5 JAX-RPC 2.0 relies on many of the Java language features added in J2SE 1.5. It is not a goal to support JAX-RPC 2.0 on Java versions prior to J2SE 1.5.

Service Registration and Discovery It is not a goal of JAX-RPC 2.0 to describe registration and discovery of services via UDDI or ebXML RR. This capability is provided independently by JAXR[16].

1.3 Requirements

1.3.1 Describe Relationship To JAXB

JAX-RPC describes the WSDL ⇔ Java mapping, but data binding is delegated to JAXB[10]. The specification must clearly designate where JAXB rules apply to the WSDL ⇔ Java mapping without reproducing those rules and must describe how JAXB capabilities (e.g. the JAXB binding language) are incorporated into JAX-RPC. JAX-RPC is required to be able to influence the JAXB binding, e.g. to avoid name collisions and to be able to control schema validation on serialization and deserialization.

1.3.2 Standardized WSDL Mapping

WSDL is the de-facto interface definition language for XML-based RPC. The specification must specify a standard WSDL ⇔ Java mapping. The following versions of WSDL must be supported:

- WSDL 1.1[5] as clarified by the WS-I Basic Profile[8, 17],
- WSDL 1.2[11, 18, 19].

The standardized WSDL mapping will describe the default WSDL ⇔ Java mapping. The default mapping may be overridden using customizations as described below.

1.3.3 Customizable WSDL Mapping

The specification must provide a standard way to customize the WSDL ⇔ Java mapping. The following customization methods will be specified:

Java Annotations The specification will define a set of standard annotations that may be used in Java source files to specify the mapping from Java artifacts to their associated WSDL components. The annotations will support mapping to both WSDL 1.1 and WSDL 1.2. The annotations defined by JAX-RPC will mesh cleanly with those defined by JAXB[10] and JSR 181[13].

WSDL Annotations The specification will define a set of standard annotations that may be used either within WSDL documents or as in an external form to specify the mapping from WSDL components to their associated Java artifacts. The annotations will support mapping from both WSDL 1.1 and WSDL 1.2. The annotations defined by JAX-RPC will mesh cleanly with those defined by the JAXB binding language.

The specification must describe the precedence rules governing combinations of the customization methods.

1.3.4 Standardized Protocol Bindings

The specification must describe standard bindings to the following protocols:

- SOAP 1.1[2] as clarified by the WS-I Basic Profile[8, 17],
- SOAP 1.2[3, 4].

The specification must not prevent non-standard bindings to other protocols.

1.3.5 Standardized Transport Bindings

The specification must describe standard bindings to the following protocols:

- HTTP/1.1[20].

The specification must not prevent non-standard bindings to other transports.

1.3.6 Standardized Handler Framework

The specification must include a standardized handler framework that describes:

Data binding for handlers The framework will offer data binding facilities to handlers and will support handlers that are decoupled from the SAAJ API.

Handler Context The framework will describe a mechanism for communicating properties between handlers and the associated service clients and service endpoint implementations.

Bidirectional handler chains Support for bidirectional handler chains that are used for both outgoing and incoming messages will be added to the existing unidirectional handler chains.

Unified Response and Fault Handling The `handleResponse` and `handleFault` methods will be unified and the the declarative model for handlers will be improved.

1.3.7 Versioning and Evolution

The specification must describe techniques and mechanisms to support versioning of service endpoint interfaces. The facilities must allow new versions of an interface to be deployed whilst maintaining compatibility for existing clients.

1.3.8 Standardized Synchronous and Asynchronous Invocation

There must be a detailed description of the generated method signatures to support both asynchronous and synchronous method invocation in stubs generated by JAX-RPC. Both forms of invocation will support a user configurable timeout period.

1.3.9 Session Management

The specification must describe a standard session management mechanism including:

Session APIs Definition of a session interface and methods to obtain the session interface and initiate sessions for handlers and service endpoint implementations.

HTTP based sessions The session management mechanism must support HTTP cookies and URL rewriting.

SOAP based sessions There must be a standardized way to identify the location of a session identifier in SOAP message content using Java or WSDL annotations.

1.4 Use Cases

1.4.1 Handler Framework

Reliable Messaging Support

A developer wishes to add support for a reliable messaging SOAP feature to an existing service endpoint. The support takes the form of a JAX-RPC handler.

Message Logging

A developer wishes to log incoming and outgoing messages for later analysis, e.g. checking messages using the WS-I testing tools.

WS-I Conformance Checking

A developer wishes to check incoming and outgoing messages for conformance to one or more WS-I profiles at runtime.

1.5 Conventions

The keywords ‘MUST’, ‘MUST NOT’, ‘REQUIRED’, ‘SHALL’, ‘SHALL NOT’, ‘SHOULD’, ‘SHOULD NOT’, ‘RECOMMENDED’, ‘MAY’, and ‘OPTIONAL’ in this document are to be interpreted as described in RFC 2119[21].

For convenience, conformance requirements are numbered and shown as follows:

Conformance Requirement (Example): Implementations MUST do something.

Java code and XML fragments are formatted as shown in figure 1.1:

Figure 1.1: Example Java Code

```
package com.example.hello;

public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

This specification uses a number of namespace prefixes throughout; they are listed in Table 1.1. Note that the choice of any namespace prefix is arbitrary and not semantically significant (see XML Infoset[22]).

Prefix	Namespace	Notes
env	http://www.w3.org/2003/05/soap-envelope	A normative XML Schema[23, 24] document for the http://www.w3.org/2003/05/soap-envelope namespace can be found at http://www.w3.org/2003/05/soap-envelope.
xsd	http://www.w3.org/2001/XMLSchema	The namespace of the XML Schema[23, 24] specification
jaxb	http://java.sun.com/xml/ns/jaxb	The namespace of the JAXB [9] specification

Table 1.1: Prefixes and Namespaces used in this specification.

Namespace names of the general form ‘http://example.org/...’ and ‘http://example.com/...’ represent application or context-dependent URIs (see RFC 2396[20]).

All parts of this specification are normative, with the exception of examples and sections explicitly marked as ‘Non-Normative’.

1.6 Expert Group Members

TBD

1.7 Acknowledgements

TBD

Chapter 2

Service Client APIs

This chapter describes the standard APIs provided for client side use of JAX-RPC. These APIs allow a client to configure generated stubs, create dynamic proxies for remote service endpoints and dynamically construct operation invocations.

2.1 javax.xml.rpc.ServiceFactory

`ServiceFactory` is an abstract factory class that provides various methods for the creation of `Service` instances (see section 2.2 for details of the `Service` interface).

Conformance Requirement (Concrete `ServiceFactory` required): A client side JAX-RPC runtime system, henceforth in this chapter simply called 'an implementation', must provide a concrete class that extends `ServiceFactory`.

2.1.1 Configuration

Editors Note 2.1 *The JAX-RPC 1.1 ServiceFactory provides no API to configure factory properties. These would be useful, e.g. in DII without WSDL to set the style of operations. Should we add a generic property capability ?*

The `ServiceFactory` implementation class is set using the system property named `javax.xml.rpc-.ServiceFactory` (the constant: `ServiceFactory.SERVICEFACTORY_PROPERTY`).

Conformance Requirement (`Service` class loading): An implementation MUST provide facilities to enable the `ServiceFactory.loadService(Class)` method to succeed provided all the generated artifacts are packaged with the application.

An implementation can either use a consistent naming convention for generated service implementation classes or allow an application developer to specify sufficient configuration information to locate `Service` implementation classes. Examples of such configuration information include:

- System properties
- Properties or XML-based configuration files that are looked up as resources via the `getResource` or `getResources` methods of `java.lang.ClassLoader`
- User and system preference and configuration data retrieved via the `java.util.prefs` facilities.

2.1.2 Factory Usage

A J2SE service client should use `ServiceFactory` to create `Service` instances. Alternatively, a J2SE based service client may use the JNDI naming context to lookup a service instance.

A J2EE-based service client should not use `ServiceFactory` to create `Service` instances. Instead, J2EE-based service clients should use JNDI to lookup an instance of a `Service` class as specified in JSR-109[14]. Moreover, packaging implementation-specific artifacts (including classes and configuration information) with a J2EE application is strongly discouraged as this makes the application non-portable.

2.1.3 Example

The following shows a typical use of `ServiceFactory` to create a `Service` instance.

```
ServiceFactory sf = ServiceFactory.newInstance();
Service s = sf.createService(wsdlDoc, serviceName);
```

2.2 javax.xml.rpc.Service

`Service` is an abstraction that represents a WSDL service. A WSDL service is a collection of related ports, each of which represents a remote service interface bound to a particular protocol and available at particular endpoint address.

`Service` instances are created using methods on a concrete `ServiceFactory` instance or are obtained via JNDI. `Service` instances provide facilities to:

- Obtain a stub via one of the `getPort` methods. See section 2.5 for information on stubs.
- Create a dynamic proxy via one of the `getPort` methods. See section 2.5.2 for information on dynamic proxies.
- Create a `Dispatch` instance via one of the `createDispatch` methods. See section 2.6 for information on the `Dispatch` interface.
- Create a `Call` instance via one of the `createCall` methods. See section 2.7 for information on the `Call` interface.
- Create a `ProtocolBinding` instance via one of the `createProtocolBinding` methods. See section 2.4 for information on the `ProtocolBinding` interface.

2.3 javax.xml.rpc.JAXRPCContext

Additional metadata is often required to control and annotate information exchanges. This metadata forms the context of an exchange and the `JAXRPCContext` interface provides programmatic access to this metadata. Request metadata is created by a client and passed to a protocol binding when an operation is invoked, response metadata is created by a protocol binding and passed to the client when an operation returns a response.

Protocol bindings are responsible for annotating outbound protocol data units when the metadata is required to be transferred with a request. Protocol bindings are responsible for extracting metadata from inbound protocol data units when that data is required by clients. E.g. a handler in a SOAP binding might introduce a header into a SOAP request message to carry metadata from the `JAXRPCContext` and might create a metadata entry in the `JAXRPCContext` from the contents of a header in a response SOAP message.

Metadata takes the form of a set of named properties. JAX-PRC defines a set of standard properties and implementations can add additional proprietary properties although use of these is discouraged as it may limit application portability.

2.3.1 Standard Properties

Table 2.1 lists a set of standard properties that may be set on a `JAXRPCContext` instance and shows which properties are optional for implementations to support.

Name	Type	Mandatory	Description
`javax.xml.rpc.security`			
`.username`	String	Y	Username for HTTP basic authentication. Section **??** describes the JAX-RPC support for HTTP authentication.
`.password`	String	Y	Password for HTTP basic authentication.
`javax.xml.rpc.session`			
`.maintain`	Boolean	Y	Used by a client to indicate whether it is prepared to participate in a service endpoint initiated session. The default value is `false`. Section **??** describes the JAX-RPC support for HTTP sessions.
`javax.xml.rpc.soap.operation` Properties			
`.style`	String	N	The style of the operation: either `rpc` or `document`.
`javax.xml.rpc.soap.http.soapaction`			
`.use`	Boolean	N	Controls whether the `SOAPAction` HTTP header is used in SOAP/HTTP requests. Default value is `false`.
`.uri`	String	N	The value of the `SOAPAction` HTTP header if the `javax.xml.rpc.soap.http.soapaction.use` property is set to `true`. Default value is an empty string.
`javax.xml.rpc.encodingstyle.namespace`			
`.uri`	String	N	The encoding style specified as a URI. Default value is the URI corresponding to SOAP encoding: `http://schemas.xmlsoap.org/soap/encoding/`.

Table 2.1: Standard `JAXRPCContext` properties.

Conformance Requirement (Required `JAXRPCContext` properties): An implementation MUST support the properties shown as mandatory in table 2.1.

Conformance Requirement (Optional `JAXRPCContext` properties): An implementation MAY support the properties shown as optional in table 2.1.

Conformance Requirement (Unsupported `JAXRPCContext` properties): An implementation MUST throw `JAXRPCExcpetion` if a client attempts to set the value of an unsupported property or if an error in the value of the property is detected.

2.3.2 Additional Properties

Conformance Requirement (Additional `JAXRPCContext` properties): An implementation MAY support additional implementation specific properties not listed in table 2.1. Such properties MUST NOT use the `javax.xml.rpc` prefix in their names.

2.4 javax.xml.rpc.ProtocolBinding

Editors Note 2.2 *This is a placeholder for description of a new protocol binding interface. This is used to configure the binding of a dispatch instance and as a base class for the SOAP protocol binding in the handler proposal.*

Current thoughts:

- *Stubs provide read-only access to their protocol binding.*
- *Dispatch instances allow read/write access to their protocol binding.*
- *Need to think about how ProtocolBinding relates to JAXRPCContext, it might be possible to collapse the two into one interface but then things get tricky in the async case.*
- *For dispatch we might need a way to create a well-known protocol binding in the absence of WSDL so dispatch can be used entirely standalone. We could add a Service.createProtocolBinding(URI) and define one or more URIs for SOAP/HTTP etc.*

We would define at least one protocol binding subclass for the SOAP/HTTP binding - this would provide the SOAP handler framework.

2.5 javax.xml.rpc.Stub

WSDL to Java mapping implementations generate strongly typed Java interfaces for services described in WSDL, see chapter **??**. Implementations also allow generation of client side stub classes and server side tie classes that implement the mapping between Java and the protocol binding described in the WSDL. Generated stub classes implement the Java interface generated from the WSDL and also implement the `Stub` interface.

Conformance Requirement (Implementing `Stub`): Client-side generated stubs MUST implement the `javax-.xml.rpc.Stub` interface.

Conformance Requirement (Stub class binding): A generated stub class SHOULD be bound to a particular protocol and transport (if applicable).

Generated stub classes are either named after the protocol binding in the WSDL (*`BindingName`*`_Stub` where *BindingName* is the value of the `name` attribute of the corresponding WSDL `binding` element) or using some other implementation specific naming scheme.

2.5.1 Configuration

Stub classes are not required to be dynamically configurable for different protocols and bindings but may support configuration of certain aspects of their operation. Stub classes support two forms of configuration:

Static The WSDL binding from which the stub class was generated contains static information including the protocol binding and service endpoint address.

Dynamic The `Stub` interface provides methods to dynamically query and change the values of stub properties.

Stub properties are name, value pairs. JAX-RPC defines a standard set of properties and implementations may define additional properties. Standard properties are listed in table 2.1.

Conformance Requirement (`Stub` configuration): An implementation MUST throw a `JAXRPCException` if a client attempts to set an unknown or unsupported optional property or if an implementation detects an error in the value of a property.

Editors Note 2.3 *The remains of this subsection proposes a relationship between JAXRPCContext and the existing Stub config methods; either method of setting properties is equivalent. A better solution might be to deprecate the existing stub config methods (_setProperty etc) in favor of the context setter and getter ?*

The `JAXRPCContext` (see section 2.3) interface provides a common container interface for metadata that may be used with `Stub` and `Dispatch` (see section 2.6) instances. For backwards compatibility with JAX-RPC 1.1, the `Stub` property methods are retained. Setting the value of a property in a `JAXRPCContext` instance and then setting the context of a `Stub` instance is equivalent to setting the value of the property directly on the `Stub` instance. E.g. in the following code fragment, line 6 would print `true`.

```
1  javax.xml.rpc.Stub stub = ...;
2  stub._setProperty("javax.xml.rpc.session.maintain", new Boolean(false));
3  javax.xml.rpc.JAXRPCContext context = stub.createContext();
4  context.setProperty("javax.xml.rpc.session.maintain", new Boolean(true));
5  stub.setContext(context);
6  System.out.println(stub._getProperty("javax.xml.rpc.session.maintain");
```

2.5.2 Dynamic Proxy

In addition to statically generated stub classes, JAX-RPC also provides dynamic proxy generation support. Dynamic proxies provide access to service endpoint interfaces at runtime without requiring static generation of a stub class. See `java.lang.reflect.Proxy` for more information on dynamic proxies.

Conformance Requirement (Dynamic proxy support): An implementation MUST support generation of dynamic proxies.

Conformance Requirement (Implementing `Stub` required): Dynamic proxy instances MUST implement the `javax.xml.rpc.Stub` interface.

A dynamic proxy is created using the `getPort` method of a `Service` instance. The `serviceEndpoint-Interface` parameter specifies the interface that will be implemented by the generated proxy. The service endpoint interface provided by the client needs to conform to the WSDL to Java mapping rules specified in chapter **??** (WSDL 1.1) or chapter **??** (WSDL 2.0). Generation of a dynamic proxy can fail if the interface doesn't conform to the mapping or if any WSDL related metadata is missing from the `Service` instance.

Conformance Requirement (Failed `Service.getPort`): An implementation MUST throw a `javax.xml-.rpc.ServiceException` if generation of a dynamic proxy fails.

An implementation is not required to fully validate the service endpoint interface provided by the client against the corresponding WSDL definitions and may choose to implement any validation it does require in an implementation specific manner (e.g. lazy and eager validation are both acceptable).

Example

The following example shows the use of a dynamic proxy to invoke a method (`getLastTradePrice`) on a service endpoint interface (`com.example.StockQuoteProvider`). Note that no statically generated stub class is involved.

```
1  javax.xml.rpc.Service service = ...;
2  java.rmi.Remote proxy = service.getPort(portName,
3      com.example.StockQuoteProvider.class)
4  javax.xml.rpc.Stub stub = (javax.xml.rpc.Stub)proxy;
5  javax.xml.rpc.JAXRPCContext context = stub.createContext();
6  context.setProperty("javax.xml.rpc.session.maintain", new Boolean(true));
7  stub.setContext(context);
8  com.example.StockQuoteProvider sqp = (com.example.StockQuoteProvider)proxy;
9  sqp.getLastTradePrice("ACME");
```

Lines 1–3 show how the dynamic proxy is generated. Lines 4–7 cast the proxy to a `Stub` and perform some dynamic configuration of the stub. Lines 8–9 cast the the proxy to the service endpoint interface and invoke the method.

2.6 javax.xml.rpc.Dispatch

Editors Note 2.4 *`Dispatch` is a placeholder, other alternatives considered include `Post`, `Transmit` and `Send`. Alternative suggestions welcome.*

XML based RPC represents RPC invocations and any associated responses as XML messages. The higher level JAX-RPC APIs are designed to hide the details of converting between Java objects and their XML representations but in some cases operating at the XML representation level is desirable. The `Dispatch` interface provides support for this mode of interaction.

Conformance Requirement (`Dispatch` support): Implementations MUST support the `javax.xml.rpc-.Dispatch` interface.

`Dispatch` is a low level API that requires clients to construct XML based RPC message payloads as XML fragments and requires an intimate knowledge of the desired payload structure. For convenience the `Dispatch` API also provides a means for direct use of JAXB objects. This allows clients to use JAXB objects generated from an XML Schema to create and manipulate XML representations and to use these objects with JAX-RPC without requiring an intermediate XML serialization.

2.6.1 Configuration

`Dispatch` instances are obtained using one of the `createDispatch` methods of a `Service` instance. `Dispatch` instances are not thread safe, use of the same instance in multiple threads requires considerable care. The `clone` method allows creation of exact copies of an existing `Dispatch` instance for use in other threads.

A `Dispatch` instance can be created in one of two states depending on the combination of how the `Service` instance was obtained and which `createDispatch` variant was used:

Unbound The client is responsible for binding the `Dispatch` instance to a protocol and endpoint address prior to use.

Bound The `Dispatch` instance is ready for use.

Table 2.2 shows the state for each combination. Combinations shown as U/B may result in either unbound or bound `Dispatch` instances, the result is implementation dependent. A client can determine whether a `Dispatch` instance is bound or not by use of the `isBound` method. This method returns `true` for bound instances ('B' in table 2.2), `false` otherwise.

	`createDispatch` Arguments	
	None	`QName port`
`createService` Arguments		
`QName svc`	U	U
`URL wsdlDoc, QName svc`	U	B
`loadService` Arguments		
`Class sei`	U	U/B
`URL wsdlDoc, Class sei, Properties p`	U	U/B
`URL wsdlDoc, QName svc, Properties p`	U	U/B

Table 2.2: `Dispatch` states resulting from combinations of `Service` creation and `createDispatch` variants.

Unbound `Dispatch` instances require configuration using the appropriate setter methods prior to use of the `invoke` and `invokeOneWay` methods. `Dispatch` instances are mutable, a client may change the configuration of an existing instance and re-use it.

Conformance Requirement (Unbound `Dispatch`): An implementation MUST throw a `JAXRPCException` if the `invoke` or `invokeOneWay` methods are called on an unbound instance.

Setter methods are provided to configure the protocol to which the instance is bound and the endpoint address.

2.6.2 Operation Invocation

A `Dispatch` instance supports three invocation modes:

Synchronous request response (`invoke` methods) The operation invocation blocks until the remote operation completes and the results are returned.

Asynchronous request response (`invokeAsync` methods) The operation invocation returns immediately, any results are provided either through a callback or via a polling object.

One-way (`invokeOneWay` methods) The operation invocation is logically non-blocking, subject to the capabilities of the underlying protocol, no results are returned.

Conformance Requirement (Failed `Dispatch.invoke`): When an operation is invoked using an `invoke` method, an implementation MUST throw a `JAXRPCException` if there is any error in the configuration of the `Dispatch` instance or a `java.rmi.RemoteException` if an error occurs during the remote operation invocation.

Conformance Requirement (Failed `Dispatch.invokeAsync`*):* When an operation is invoked using an `invokeAsync` method, an implementation MUST throw a `JAXRPCException` if there is any error in the configuration of the `Dispatch` instance. Errors that occur during the invocation are reported when the client attempts to retrieve the results of the operation.

Conformance Requirement (Failed `Dispatch.invokeOneWay`*):* When an operation is invoked using an `invokeOneWay` method, an implementation MUST throw a `JAXRPCException` if there is any error in the configuration of the `Dispatch` instance or if an error is detected[1] during the remote operation invocation.

Conformance Requirement (One-way operations): When invoking one-way operations where the binding is SOAP/HTTP an implementation MUST block until the HTTP response is received or an error occurs. Completion of the HTTP request simply means that the transmission of the request is complete, not that the request was accepted or processed.

Editors Note 2.5 *When using the JAXB oriented invoke methods its possible that responses containing application defined faults can be bound to Java objects. We need a way to get these objects (or their XML representation when mapping is not possible) from either a RemoteException (sync invoke) or JAXRPCException (async invoke). Using the cause mechanism seems like the right way to go in combination with the pending work on representing protocol specific fault information. If we go with the current proposal then we could add a new field to ProtocolDetailsException for the object unmarshalled from the response by JAXB.*

2.6.3 Operation Response

The following interfaces are used to obtain the results of an operation invocation:

`javax.xml.rpc.Response` A generic interface that is used to group the results of an invocation with the response context.

`javax.xml.rpc.AsyncResponse` A generic interface that extends `Future<T>` to provide asynchronous result polling capabilities.

`javax.xml.rpc.AsyncHandler` A generic interface that clients implement to receive results in an asynchronous callback.

2.6.4 Asynchronous Response

`Dispatch` supports two forms of asynchronous invocation:

Polling The `invokeAsync` method returns a typed `AsyncResponse` (either `AsyncResponse<Source>` or `AsyncResponse<Object>`) that may be polled using the methods inherited from `Future<T>` to determine when the operation has completed and to retrieve the results.

Callback The client supplies a typed `AsyncHandler` (either `AsyncHandler<Source>` or `AsyncHandler<Object>`) and the runtime calls the `handleResponse` method when the results of the operation are available. The `invokeAsync` method returns a wildcard `AsyncResponse` (`AsyncResponse<?>`) that may be polled using the methods inherited from `Future<T>` to determine when the operation has completed. The object returned from `AsyncResponse<?>.get` has no standard type. Client code should not attempt to cast the object to any particular type as this will result in non-portable behaviour.

[1]The invocation is logically non-blocking so detection of errors during operation invocation is dependent on the underlying protocol in use. For SOAP/HTTP it is possible that certain HTTP level errors may be detected.

In both cases, errors that occur during the invocation are reported via an exception when the client attempts to retrieve the results of the operation.

Conformance Requirement (Reporting asynchronous errors): An implementation MUST throw a `JAXRPCException` from `AsyncResponse.get` and `Response.get` if the operation invocation failed.

2.6.5 Examples

The following examples demonstrate use of `Dispatch` methods in the synchronous, asynchronous polling and asynchronous callback modes. For ease of reading, error handling has been omitted.

Synchronous

```
javax.xml.rpc.Service service = ...;
javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
Source reqMsg = ...;
JAXRPCContext reqCtx = disp.createJAXRPCContext();
reqCtx.setProperty(...);
Response<Source> res = disp.invoke(reqMsg, reqCtx);
Source resSrc = res.get();
JAXRPCContext resCtx = res.getContext();
\\ do something with the results
```

Synchronous With JAXB Objects

```
javax.xml.rpc.Service service = ...;
javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
Unmarshaller u = jc.createUnmarshaller();
PurchaseOrder po = (PurchaseOrder)u.unmarshal(
    new FileInputStream( "po.xml" ) );
JAXRPCContext reqCtx = disp.createJAXRPCContext();
reqCtx.setProperty(...);
Response<Object> res = disp.invoke(po, jc, reqCtx);
OrderConfirmation conf = (OrderConfirmation)res.get();
JAXRPCContext resCtx = res.getContext();
\\ do something with the results
```

In the above example `PurchaseOrder` and `OrderConfirmation` are interfaces pre-generated by JAXB from the schema ‘primer.po’.

Asynchronous Polling

```
javax.xml.rpc.Service service = ...;
javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
Source reqMsg = ...;
JAXRPCContext reqCtx = disp.createJAXRPCContext();
reqCtx.setProperty(...);
AsyncResponse<Source> res = disp.invokeAsync(reqMsg, reqCtx);
while (!res.isDone()) {
    // do something while we wait
```

```
}
Response<Source> response = res.getResponse();
Source resSrc = response.get();
JAXRPCContext resCtx = response.getContext();
\\ do something with the results
```

Asynchronous Callback

```
class MyHandler implements AsyncHandler<Source> {
    ...
    public void handleResponse(Response<Source> res) {
        Source resSrc = res.get();
        JAXRPCContext resCtx = res.getContext();
        \\ do something with the results
    }
}

javax.xml.rpc.Service service = ...;
javax.xml.rpc.Dispatch disp = service.createDispatch(portName);
Source reqMsg = ...;
JAXRPCContext reqCtx = disp.createJAXRPCContext();
reqCtx.setProperty(...);
MyHandler handler = new MyHandler();
(void)disp.invokeAsync(reqMsg, reqCtx, handler);
// result will be handled in a different thread
```

2.7 javax.xml.rpc.Call

Editors Note 2.6 *The EG has agreed to reposition the Call API as a SOAP Encoding centric API and is considering whether to deprecate it in favor of the* `Dispatch` *API which is more document/rpc literal friendly.*

The `Call` interface provides support for dynamically constructing operation invocations either with or without a WSDL description of the service endpoint.

Conformance Requirement (`Call` *support):* Implementations MUST support the `javax.xml.rpc.Call` interface.

2.7.1 Configuration

`Call` instances are obtained using one of the `createCall` methods of a `Service` instance. A `Call` instance can be created in one of two states depending on the combination of how the `Service` instance was obtained and which `createCall` variant was used:

Unconfigured The client is responsible for configuring `Call` instance prior to use

Configured The `Call` instance is ready for use

Table 2.3 shows the state for each combination. Combinations shown as U/C may result in either unconfigured or configured `Call` instances, the result is implementation dependent.

		`Service.createCall` Arguments		
	None	`QName port`	`QName port, QName op`	`QName port, String op`
`createService` Arguments				
`QName svc`	U	U	U	U
`URL wsdlDoc, QName svc`	U	U	C	C
`loadService` Arguments				
`Class si`	U	U	U/C	U/C
`URL wsdlDoc, Class si, Properties p`	U	U	U/C	U/C
`URL wsdlDoc, QName svc, Properties p`	U	U	U/C	U/C

Table 2.3: Call states resulting from combinations of `Service` creation and `createCall` variants.

Unconfigured `Call` instances require configuration using the appropriate setter methods prior to use of the `invoke` and `invokeOneWay` methods. `Call` instances are mutable, a client may change the configuration of an existing instance and re-use it.

Conformance Requirement (Unconfigured `Call`): An implementation MUST throw a `JAXRPCException` if the `invoke` or `invokeOneWay` methods are called on an unconfigured instance.

Setter methods are provided to configure:

- The name of the operation to invoke
- The names, types and modes of the operation parameters
- The operation return type
- The name of the port type
- The endpoint address
- Additional properties (see section 2.3.1 on page 9 for a list of standard properties).

Conformance Requirement (`Call` configuration): An implementation MUST throw a `JAXRPCException` if a client attempts to set an unknown or unsupported optional property or if an implementation detects an error in the value of a property.

A client can determine the level of configuration a `Call` instance requires by use of the `isParameterAndReturnSpecRequired` method. This method returns `false` for operations that only require the operation name to be configured, `true` for operations that require operation name, parameter and return types to be configured.

2.7.2 Operation Invocation

When an operation is invoked, the `Call` instance checks that the passed parameters match the number, order and types of parameters configured in the instance.

Conformance Requirement (Misconfigured invocation): When an operation is invoked, an implementation MUST throw a `JAXRPCException` if the `Call` instance is incorrectly configured or if the passed parameters do not match the configuration.

A `Call` instance supports two invocation modes:

Synchronous request response (`invoke` method) The operation invocation blocks until the remote operation completes and the results are returned.

One-way (`invokeOneWay` method) The operation invocation is logically non-blocking, subject to the capabilities of the underlying protocol, no results are returned.

Conformance Requirement (Failed `invoke`): When an operation is invoked using the `invoke` method, an implementation MUST throw a `java.rmi.RemoteException` if an error occurs during the remote invocation.

Conformance Requirement (Failed `invokeOneWay`): When an operation is invoked using the `invokeOneWay` method, an implementation MUST throw a `JAXRPCException` if an error occurs during the remote invocation.

Conformance Requirement (One-way operations): When invoking one-way operations where the binding is SOAP/HTTP, an implementation MUST block until the HTTP response is received or an error occurs. Completion of the HTTP request simply means that the transmission of the request is complete, not that the request was accepted or processed.

Once an operation has been invoked the values of the operation's output parameters may be obtained using the following methods:

`getOutputParams` The returned `Map` contains the output parameter values keyed by name. The type of the key is `String` and the type of the value depends on the mapping between XML and Java types.

`getOutputValues` The returned `List` contains the values of the output parameters in the order specified for the operation. The type of the value depends on the mapping between XML and Java types.

Conformance Requirement (Missing invocation): An implementation MUST throw `JAXRPCException` if `getOutputParams` or `getOutputValues` is called prior to `invoke` or following `invokeOneWay`.

2.7.3 Example

As described in section 2.7.1, a `Call` instance can be either configured or unconfigured. Use of a configured instance is simpler since the `Call` instance takes the responsibility of determining the corresponding types for the parameters and return value. Figure 2.1 shows an example of using a configured `Call` instance

```
javax.xml.rpc.Service service = ...
javax.xml.rpc.Call call = service.createCall( portName, operationName);
Object[] inParams = new Object[] {"hello world!"};
Integer ret = (Integer) call.invoke(inParams);
Map outParams = call.getOutputParams();
String outValue = (String)outParams.get("param2");
```

Figure 2.1: Use of configured `Call` instance

Alternatively, the `Call` instance can be unconfigured or only partially configured. In this case the client is responsible for configuring the call prior to use. Figure 2.2 shows an example of using an unconfigured `Call` instance to invoke the same operation as shown in figure 2.1.

Lines 3–5 in figure 2.2 are concerned with configuring the `Call` instance prior to its use in line 7.

```
javax.xml.rpc.Service service = ...
javax.xml.rpc.Call call = service.createCall( portName, operationName);
call.addParameter("param1", <xsd:string>, ParameterMode.IN);
call.addParameter("param2", <xsd:string>, ParameterMode.OUT);
call.setReturnType(<xsd:int>);
Object[] inParams = new Object[] {"hello world!"};
Integer ret = (Integer) call.invoke(inParams);
Map outParams = call.getOutputParams();
String outValue = (String)outParams.get("param2");
```

Figure 2.2: Use of unconfigured `Call` instance

2.8 Exceptions

The following standard exceptions are defined by JAX-RPC.

`javax.xml.rpc.ServiceException` A checked exception that is thrown by methods in the `ServiceFactory` and `Service` interfaces.

`javax.xml.rpc.JAXRPCException` A runtime exception that is thrown by methods in service client APIs when errors occurs during local processing. `java.rmi.RemoteException` is thrown when errors occurs during processing of the remote method invocation.

2.9 Additional Classes

The following additional classes are defined by JAX-RPC:

`javax.xml.rpc.NamespaceConstants` Contains constants for common XML namespace prefixes and URIs.

`javax.xml.rpc.encoding.XMLType` Contains constants for the QNames of the supported set of XML schema types and SOAP encoding types.

Appendix A

Conformance Requirements

Bibliography

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Recommendation, W3C, October 2000. See http://www.w3.org/TR/2000/REC-xml-20001006.

[2] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. Note, W3C, May 2000. See http://www.w3.org/TR/SOAP/.

[3] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. Recommendation, W3C, June 2003. See http://www.w3.org/TR/2003/REC-soap12-part1-20030624.

[4] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 2: Adjuncts. Recommendation, W3C, June 2003. See http://www.w3.org/TR/2003/REC-soap12-part2-20030624.

[5] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Note, W3C, March 2001. See http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[6] Rahul Sharma. The Java API for XML Based RPC (JAX-RPC) 1.0. JSR, JCP, June 2002. See http://jcp.org/en/jsr/detail?id=101.

[7] Roberto Chinnici. The Java API for XML Based RPC (JAX-RPC) 1.1. Maintenance JSR, JCP, August 2003. See http://jcp.org/en/jsr/detail?id=101.

[8] Keith Ballinger, David Ehnebuske, Martin Gudgin, Mark Nottingham, and Prasad Yendluri. Basic Profile Version 1.0. Board Approval Draft, WS-I, July 2003. See http://www.ws-i.org/Profiles/Basic/2003-06/BasicProfile-1.0-BdAD.html.

[9] Joseph Fialli and Sekhar Vajjhala. The Java Architecture for XML Binding (JAXB). JSR, JCP, January 2003. See http://jcp.org/en/jsr/detail?id=31.

[10] Joseph Fialli and Sekhar Vajjhala. The Java Architecture for XML Binding (JAXB) 2.0. JSR, JCP, August 2003. See http://jcp.org/en/jsr/detail?id=222.

[11] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language. Working Draft, W3C, June 2003. See http://www.w3.org/TR/2003/WD-wsdl12-20030611.

[12] Joshua Bloch. A Metadata Facility for the Java Programming Language. JSR, JCP, August 2003. See http://jcp.org/en/jsr/detail?id=175.

[13] Jim Trezzo. Web Services Metadata for the Java Platform. JSR, JCP, August 2003. See http://jcp.org/en/jsr/detail?id=181.

[14] Jim Knutson and Heather Kreger. Web Services for J2EE. JSR, JCP, September 2002. See http://jcp.org/en/jsr/detail?id=109.

[15] Nataraj Nagaratnam. Web Services Message Security APIs. JSR, JCP, April 2002. See http://jcp.org/en/jsr/detail?id=181.

[16] Farrukh Najmi. Java API for XML Registries 1.0 (JAXR). JSR, JCP, June 2002. See http://www.jcp.org/en/jsr/detail?id=93.

[17] Keith Ballinger, David Ehnebuske, Chris Ferris, Martin Gudgin, Marc Hadley, Anish Karmarkar, Canyang Kevin Liu, Mark Nottingham, Jorgen Thelin, and Prasad Yendluri. Basic Profile Version 1.1. Working Group Draft, WS-I, July 2003. Not yet published.

[18] Martin Gudgin, Amy Lewis, and Jeffrey Schlimmer. Web Services Description Language (WSDL) Version 1.2 Part 2: Message Patterns. Working Draft, W3C, June 2003. See http://www.w3.org/TR/2003/WD-wsdl12-patterns-20030611.

[19] Jean-Jacques Moreau and Jeffrey Schlimmer. Web Services Description Language (WSDL) Version 1.2 Part 3: Bindings. Working Draft, W3C, June 2003. See http://www.w3.org/TR/2003/WD-wsdl12-bindings-20030611.

[20] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. RFC, IETF, March 1997. See http://www.ietf.org/rfc/rfc2396.txt.

[21] S. Bradner. RFC 2119: Keywords for use in RFCs to Indicate Requirement Levels. RFC, IETF, March 1997. See http://www.ietf.org/rfc/rfc2119.txt.

[22] John Cowan and Richard Tobin. XML Information Set. Recommendation, W3C, October 2001. See http://www.w3.org/TR/2001/REC-xml-infoset-20011024/.

[23] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. Recommendation, W3C, May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/.

[24] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. Recommendation, W3C, May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.