Class : II MCA **Sub.Code : MC 5012**

Faculty: Dr.G.Pradeep **Title :SERVICE ORIENTED ARCHITECTURE**

UNIT IV SOA in J2EE and .NET

SOA PLATFORM BASICS

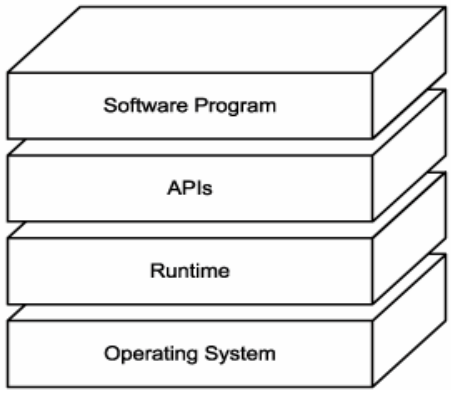
Let's first establish some of the common aspects of the physical development and runtime environments required to build and implement SOA-compliant services.

(a)Basic platform building blocks

The realization of a software program puts forth some basic requirements, mainly:

- We need a development environment with which to program and assemble the software program. This environment must provide us with a development tool that supports a programming language.
- We need a runtime for which we will be designing our software.
- We need APIs that expose features and functions offered by the runtime.
- Finally, we need an operating system on which to deploy the runtime, APIs, and the software program.

Each of these requirements can be represented as a layer that establishes a base architecture



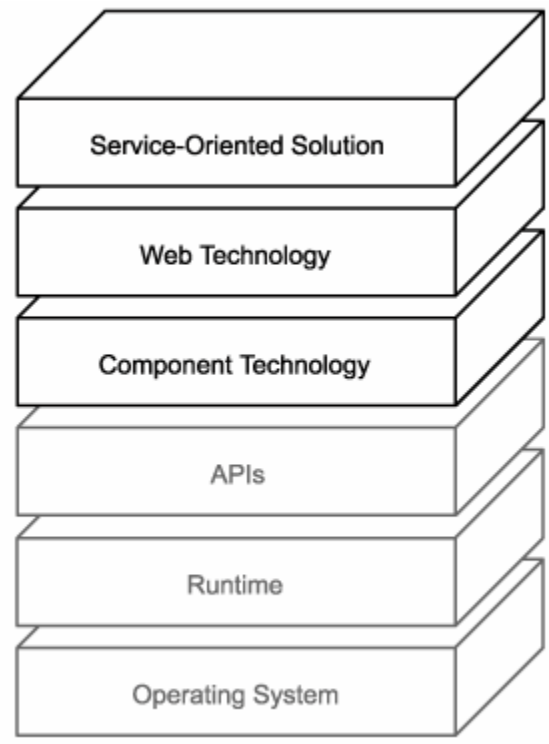
model **Fundamental software technology architecture layers.**

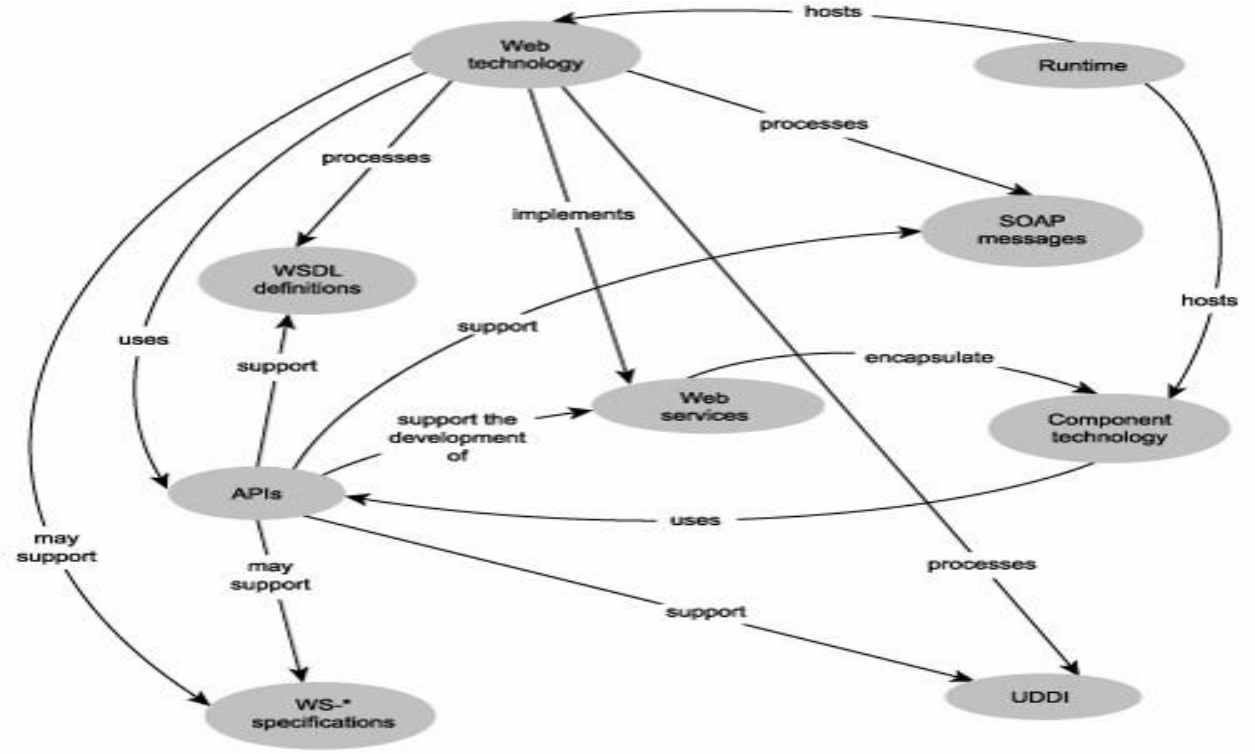
(b)Common SOA platform layers

Contemporary SOA is a distributed architectural model, built using Web services. Therefore, an SOA-capable development and runtime platform will be geared toward a distributed programming architecture that provides support for the Web services technology set. As a result, we have two new requirements:

- We need the ability to partition software programs into self-contained and composable units of processing logic (components) capable of communicating with each other within and across instances of the runtime.
- We need the ability to encapsulate and expose application logic through industry standard Web services technologies.

The common layers required by a development and runtime platform for building SOA.





(c)Relationship between SOA layers and technologies

The Primary relationships are as below:

- The Web Technology layer needs to provide support for the first-generation Web services technology set to enable us to build a primitive SOA.
- The Web Technology layer needs to provide support for WS-* specifications for us to fulfill some of the contemporary SOA characteristics.

A logical view of the basic relationships between the core parts of a service-oriented architecture.

- The Web Technology layer needs to provide a means of assembling and implementing its technology support into Web services.
- The Component Technology layer needs to support encapsulation by Web services.
- The Runtime layer needs to be capable of hosting components and Web services.
- The Runtime layer needs to provide a series of APIs in support of components and Web services.
- The APIs layer needs to provide functions that support the development and processing of components and Web services technologies.

(d)Fundamental service technology architecture

So far we've established the overall pieces that comprise a fundamental, abstract service-oriented architecture. What is of further interest to us are the specifics behind the relationship between the Web Technology and Component Technology layers.

By studying this relationship, we can learn how service providers and service requestors within an SOA can be designed, leading us to define a service-level architecture.

Service processing tasks

Service providers are commonly expected to perform the following tasks:

- Supply a public interface (WSDL definition) that allows it to be accessed and invoked by a service requestor.
- Receive a SOAP message sent to it by a service requestor.
- Process the header blocks within the SOAP message.
- Validate and parse the payload of the SOAP message.
- Transform the message payload contents into a different format.
- Encapsulate business processing logic that will do something with the received SOAP message contents.
- Assemble a SOAP message containing the response to the original request SOAP message from the service requestor.
- Transform the contents of the message back into the format expected by the service requestor.
- Transmit the response SOAP message back to the service requestor.

Service requestors are commonly expected to:

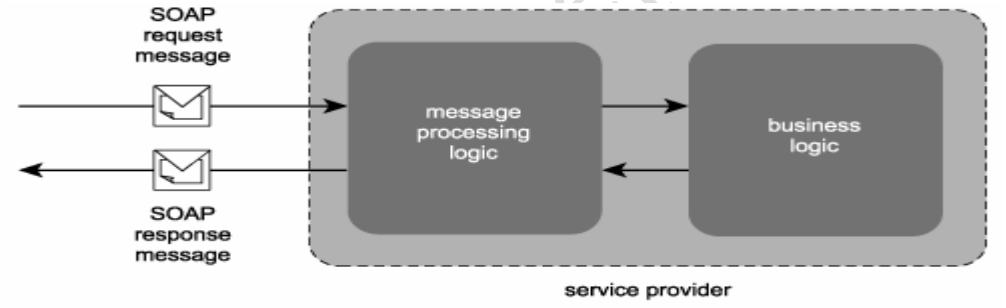
- Contain business processing logic that calls a service provider for a particular reason.
- Interpret (and possibly discover) a service provider's WSDL definition.
- Assemble a SOAP request message (including any required headers) in compliance with the service provider WSDL definition.
- Transform the contents of the SOAP message so that they comply with the format expected by the service provider.
- Transmit the SOAP request message to the service provider.
- Receive a SOAP response message from the service provider.
- Validate and parse the payload of the SOAP response message received by the service provider.
- Transform the SOAP payload into a different format.
- Process SOAP header blocks within the message.

Service processing logic

Looking at these tasks, it appears that the majority of them require the use of Web technologies. The only task that does not fall into this category is the processing of business logic, where the contents of the SOAP request are used to perform some function that may result in a response. Let's therefore group our service provider and requestor tasks into two distinct categories.

- Message Processing Logic The part of a Web service and its surrounding environment that executes a variety of SOAP message processing tasks. Message processing logic is performed by a combination of runtime services, service agents, as well as service logic related to the processing of the WSDL definition.
- Business Logic The back-end part of a Web service that performs tasks in response to the receipt of SOAP message contents. Business logic is application-specific and can range dramatically in scope, depending on the functionality exposed by the WSDL definition. For example, business logic can consist of a single component providing service-specific functions, or it can be represented by a legacy application that offers only some of its functions via the Web service.

A service provider consisting of message processing and business logic.

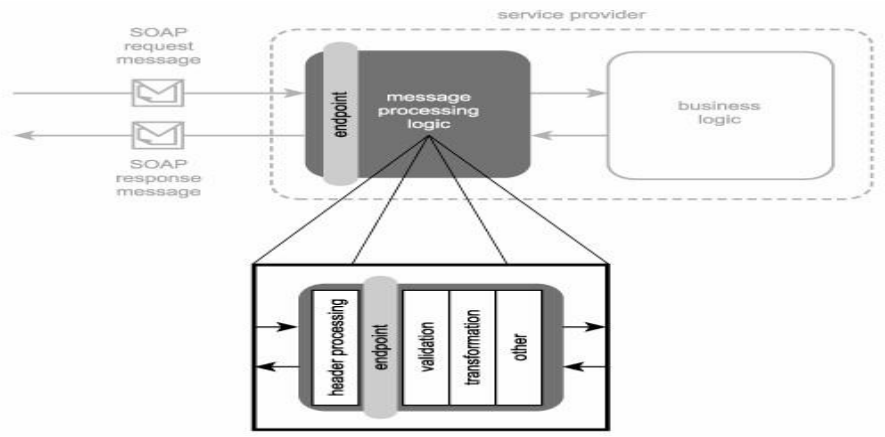


Service requestor logic categorization.	
Message Processing Logic	**Business Logic**
WSDL interpretation (and discovery).	Application-specific business processing logic.
SOAP message transmission and receipt.	
SOAP message header processing.	
SOAP message payload validation and parsing.	
SOAP message payload transformation.	

Message processing logic

Let's now take a closer look at the typical characteristics of the message processing logic of a service provider and service requestor. This part consists of functions or tasks performed by a combination of runtime services and application-specific extensions. It is therefore not easy to nail down which elements of the message processing logic belong exclusively to the service.

An example of the types of processing functions that can comprise the message processing logic of a service.

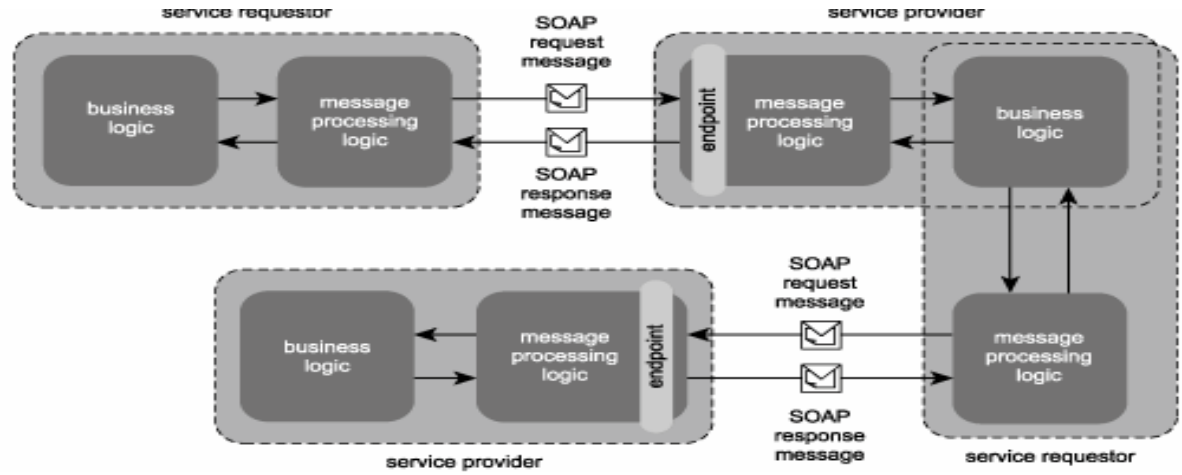


Business logic

As we previously established, business logic can exist as a standalone component, housing the intelligence required to either invoke a service provider as part of a business activity or to respond to a request in order to participate in such an activity.

As an independent unit of logic, it is free to act in different roles

The same unit of business logic participating within a service provider and a service requestor.



If units of business logic exist as physically separate components, the same business logic can be encapsulated by different service providers

Service agents

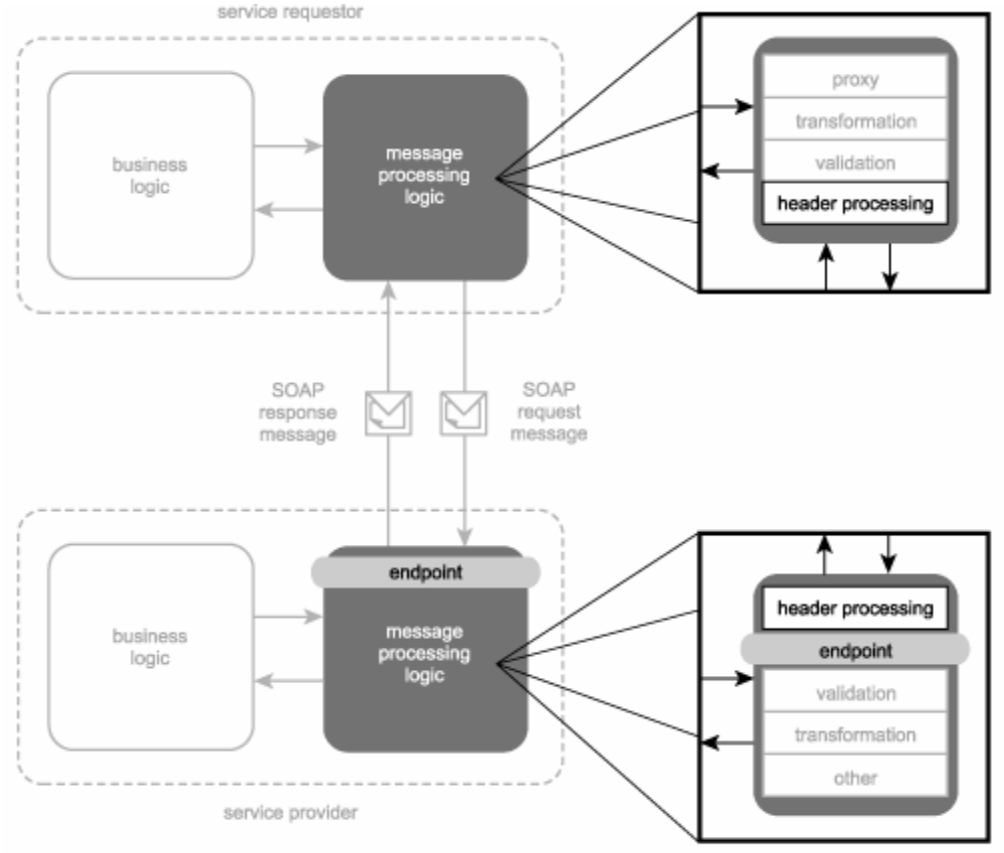
A type of software program commonly found within the message processing logic of SOA platforms is the service agent. Its primary role is to perform some form of automated processing prior to the transmission and receipt of SOAP messages. As such, service agents are a form of intermediary service.

For example, service agents that reside alongside the service requestor will be engaged after a SOAP message request is issued by the service requestor and before it actually is transmitted to the service provider. Similarly, requestor agents generally kick in upon the initial receipt of a SOAP response, prior to the SOAP message being received by the remaining service requestor logic.

Examples of the types of tasks performed by service agents include:

- SOAP header processing
- filtering (based on SOAP header or payload content)
- authentication and content-based validation
- logging and auditing
- routing

Service agents processing incoming and outgoing SOAP message headers.



An agent program usually exists as a lightweight application with a small memory footprint. It typically is provided by the runtime but also can be custom developed.

Vendor platforms

Let's now explore SOA support provided by both J2EE and .NET platforms. The next two sections consist of the following sub-sections through which each platform is discussed:

- Architecture components
- Runtime environments
- Programming languages
- APIs
- Service providers
- Service requestors
- Service agents
- Platform extensions

Because we are exploring platforms from the perspective that they are comprised of both standards and the vendor manufactured technology that implements and builds upon these standards, we mention example vendor products that can be used to realize parts of a platform.

SOA SUPPORT IN J2EE

The Java 2 Platform Enterprise Edition (J2EE) is one of the two primary platforms currently being used to develop enterprise solutions using Web services. This section briefly introduces parts of the J2EE platform relevant to SOA. The purpose of this section is simply to continue our exploration of SOA realization. In doing so, we highlight some of the main areas of interest within the J2EE platform.

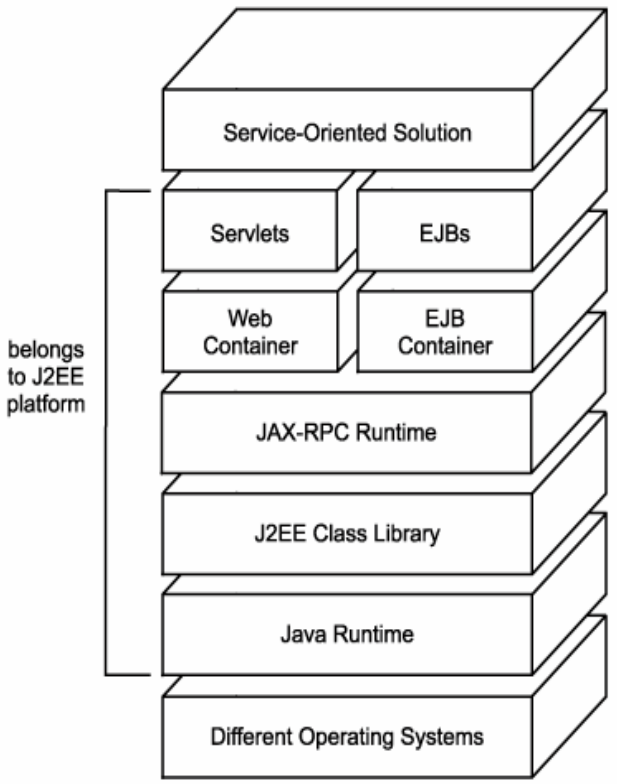
Platform overview

The Java 2 Platform is a development and runtime environment based on the Java programming language. It is a standardized platform that is supported by many vendors that provide development tools, server runtimes, and middleware products for the creation and deployment of Java solutions.

The Java 2 Platform is divided into three major development and runtime platforms, each addressing a different type of solution. The **Java 2 Platform Standard Edition (J2SE)** is designed to support the creation of desktop applications, while the **Micro Edition (J2ME)** is geared toward applications that run on mobile devices. The **Java 2 Platform Enterprise Edition (J2EE)** is built to support large-scale, distributed solutions. J2EE has been in existence for over five years and has been used extensively to build traditional n-tier applications with and without Web technologies.

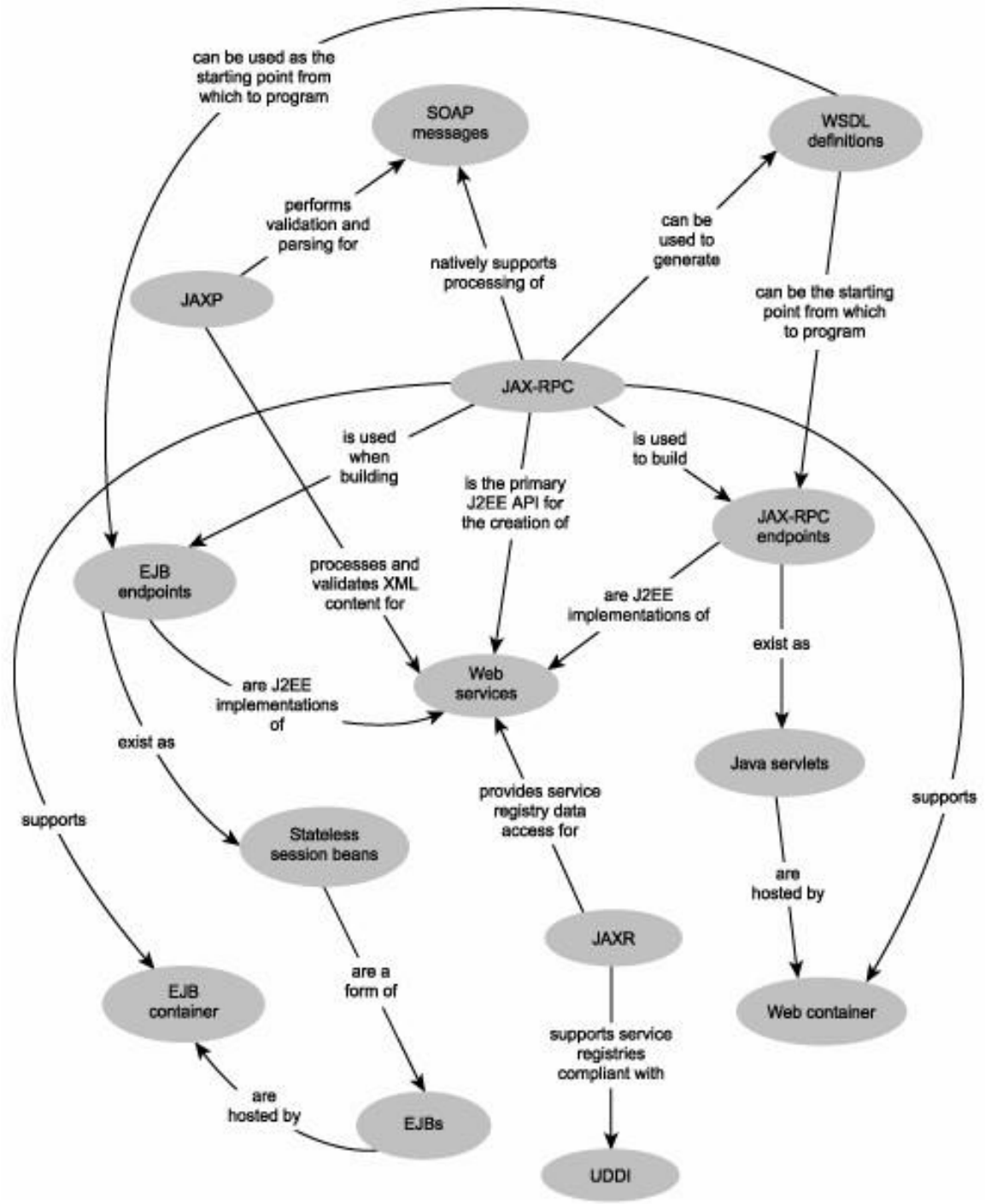
The J2EE development platform consists of numerous composable pieces that can be assembled into full-fledged Web solutions. Let's take a look at some of the technologies more relevant to Web services.

Relevant layers of the J2EE platform as they relate to SOA.



The Servlets + EJBs and Web + EJB Container layers (as well as the JAX-RPC Runtime) relate to the Web and Component Technology layers established earlier in the SOA platform basics section. They do not map cleanly to these layers because to what extent component and Web technology is incorporated is largely dependent on how a vendor chooses to implement this part of a J2EE architecture.

How parts of the J2EE platform inter-relate.



Three of the more significant specifications that pertain to SOA are listed here:

- Java 2 Platform Enterprise Edition Specification This important specification establishes the distributed J2EE component architecture and provides foundation standards that J2EE product vendors are required to fulfill in order to claim J2EE compliance.
- Java API for XML-based RPC (JAX-RPC) This document defines the JAX-RPC environment and associated core APIs. It also establishes the Service Endpoint Model used to realize the JAX-RPC Service Endpoint, one of the primary types of J2EE Web services (explained later).
- Web Services for J2EE The specification that defines the vanilla J2EE service architecture and clearly lays out what parts of the service environment can be built by the developer, implemented in a vendor-specific manner, and which parts must be delivered according to J2EE standards.

Architecture components

J2EE solutions inherently are distributed and therefore componentized. The following types of components can be used to build J2EE Web applications:

- Java Server Pages (JSPs) Dynamically generated Web pages hosted by the Web server. JSPs exist as text files comprised of code interspersed with HTML.
- Struts An extension to J2EE that allows for the development of Web applications with sophisticated user-interfaces and navigation.
- Java Servlets These components also reside on the Web server and are used to process HTTP request and response exchanges. Unlike JSPs, servlets are compiled programs.
- Enterprise JavaBeans (EJBs) The business components that perform the bulk of the processing within enterprise solution environments. They are deployed on dedicated application servers and can therefore leverage middleware features, such as transaction support.

While the first two components are of more relevance to establishing the presentation layer of a service-oriented solution, the latter two commonly are used to realize Web services.

Runtime environments

The J2EE environment relies on a foundation Java runtime to process the core Java parts of any J2EE solution. In support of Web services, J2EE provides additional runtime layers that, in turn, supply additional Web services specific APIs (explained later). Most notable is the JAX-RPC runtime, which establishes fundamental services, including support for SOAP communication and WSDL processing.

Additionally, implementations of J2EE supply two types of component containers that provide hosting environments:

- EJB container This container is designed specifically to host EJB components, and it provides a series of enterprise-level services that can be used collectively by EJBs

participating in the distributed execution of a business task. Examples of these services include transaction management, concurrency management, operation-level security, and object pooling.

- Web container A Web container can be considered an extension to a Web server and is used to host Java Web applications consisting of JSP or Java servlet components. Web

containers provide runtime services geared toward the processing of JSP requests and servlet instances.

Programming languages

As its name implies, the Java 2 Platform Enterprise Edition is centered around the Java programming language. Different vendors offer proprietary development products that provide an environment in which the standard Java language can be used to build Web services.

APIs

J2EE contains several APIs for programming functions in support of Web services. The classes that support these APIs are organized into a series of packages. Here are some of the APIs relevant to building SOA.

- Java API for XML Processing (JAXP) This API is used to process XML document content using a number of available parsers. Both Document Object Model (DOM) and Simple API for XML (SAX) compliant models are supported, as well as the ability to transform and validate XML documents using XSLT stylesheets and XSD schemas. Example packages include:

  ~ javax.xml.parsersA package containing classes for different vendor-specific DOM and SAX parsers.

  ~ org.w3c.dom and org.xml.saxThese packages expose the industry standard DOM and SAX document models.

  ~ javax.xml.transformA package providing classes that expose XSLT transformation functions.

- Java API for XML-based RPC (JAX-RPC) The most established and popular SOAP processing API, supporting both RPC-literal and document-literal request-response exchanges and one-way transmissions. Example packages that support this API include:

~ javax.xml.rpc and javax.xml.rpc.serverThese packages contain a series of core functions for the JAX-RPC API.

~ javax.xml.rpc.handler and javax.xml.rpc.handler.soapAPI functions for runtime message handlers are provided by these collections of classes. (Handlers are discussed shortly in the Service agents section.)

~ javax.xml.soap and javax.xml.rpc.soapAPI functions for processing SOAP message content and bindings.

- Java API for XML Registries (JAXR) An API that offers a standard interface for accessing business and service registries. Originally developed for ebXML directories, JAXR now includes support for UDDI.

  ~ javax.xml.registryA series of registry access functions that support the JAXR API.

  ~ javax.xml.registry.infomodelClasses that represent objects within a registry.

- Java API for XML Messaging (JAXM) An asynchronous, document-style SOAP messaging API that can be used for one-way and broadcast message transmissions (but can still facilitate synchronous exchanges as well).
- SOAP with Attachments API for Java (SAAJ) Provides an API specifically for managing SOAP messages requiring attachments. The SAAJ API is an implementation of the SOAP with Attachments (SwA) specification.
- Java Architecture for XML Binding API (JAXB) This API provides a means of generating Java classes from XSD schemas and further abstracting XML-level development.
- Java Message Service API (JMS) A Java-centric messaging protocol used for traditional messaging middleware solutions and providing reliable delivery features not found in typical HTTP communication.

Of these APIs, the two most commonly used for SOA are JAX-RPC to govern SOAP messaging and JAXP for XML document processing. The two other packages relevant to building the business logic for J2EE Web services are javax.ejb and javax.servlet, which provide fundamental APIs for the development of EJBs and servlets.

Service providers

J2EE Web services are typically implemented as servlets or EJB components. Each option is suitable to meet different requirements but also results in different deployment configurations, as explained here:

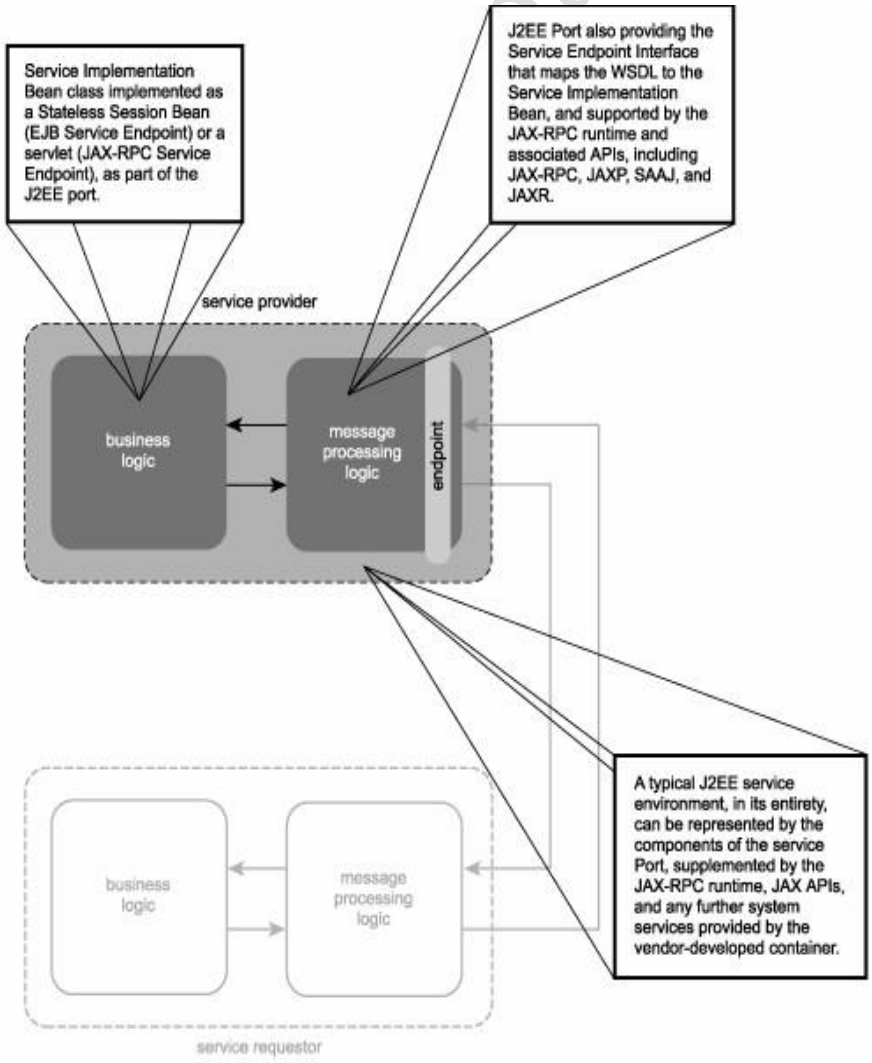
- JAX-RPC Service Endpoint When building Web services for use within a Web container, a JAX-RPC Service Endpoint is developed that frequently is implemented as a servlet by the underlying Web container logic.

- EJB Service Endpoint The alternative is to expose an EJB as a Web service through an EJB Service Endpoint. This approach is appropriate when wanting to encapsulate existing legacy logic or when runtime features only available within an EJB container are required.

Also a key part of either service architecture is an underlying model that defines its implementation, called the Port Component Model. As described in the Web Services for J2EE specification, it establishes a series of components that comprise the implementation of a J2EE service provider, including:

- Service Endpoint Interface (SEI) A Java-based interpretation of the WSDL definition that is required to follow the JAX-RPC WSDL-to-Java mapping rules to ensure consistent representation.
- Service Implementation Bean A class that is built by a developer to house the custom business logic of a Web service.

A typical J2EE service provider.



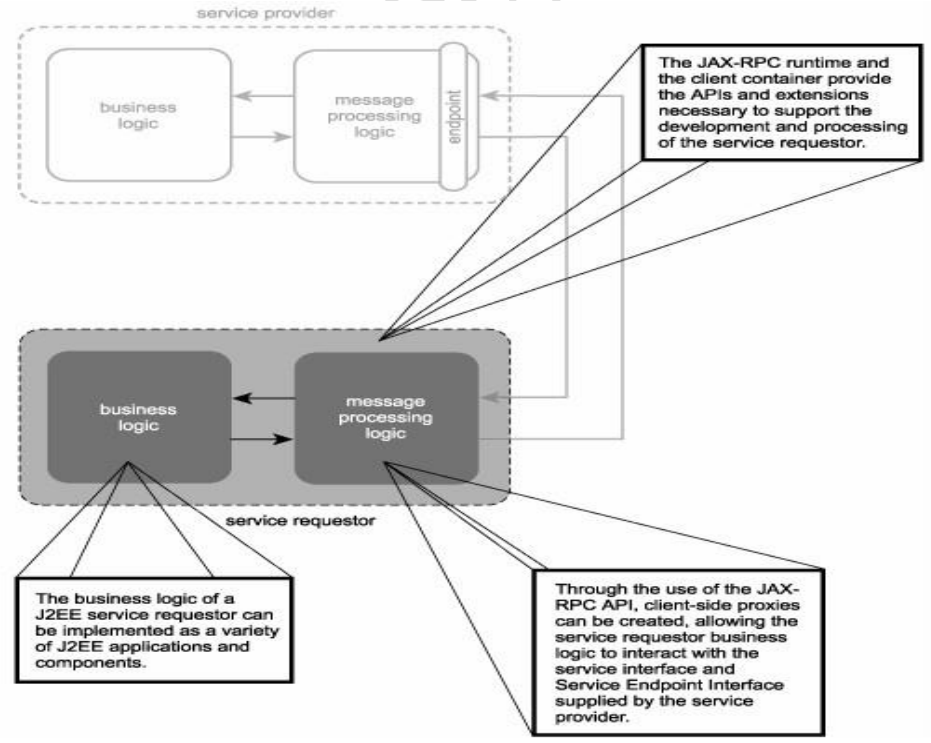
Service requestors

The JAX-RPC API also can be used to develop service requestors. It provides the ability to create three types of client proxies, as explained here:

- Generated stub The generated stub (or just "stub") is the most common form of service client. It is auto-generated by the JAX-RPC compiler (at design time) by consuming the service provider WSDL, and producing a Java-equivalent proxy component.

- Dynamic proxy and dynamic invocation interface Two variations of the generated stub are also supported. The dynamic proxy is similar in concept, except that the actual stub is not created until its methods are invoked at runtime. Secondly, the dynamic invocation interface bypasses the need for a physical stub altogether and allows for fully dynamic interaction between a Java component and a WSDL definition at runtime.

A typical J2EE service requestor.



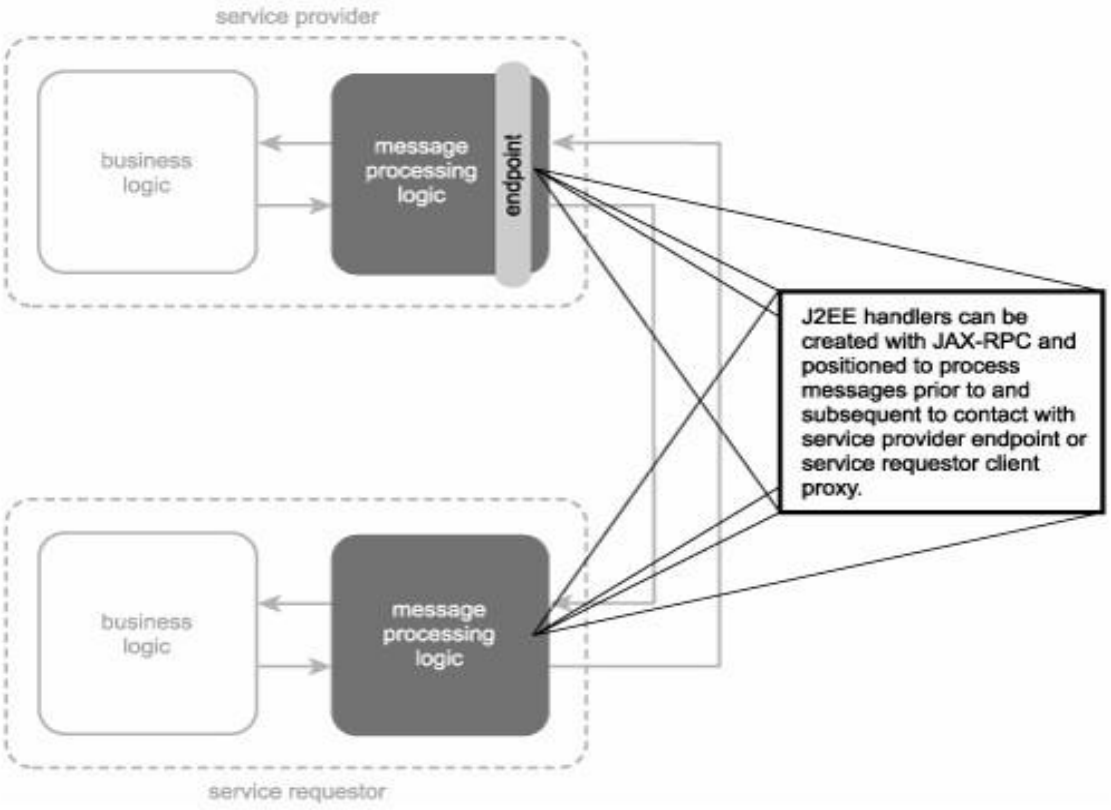
Service agents

Vendor implementations of J2EE platforms often employ numerous service agents to perform a variety of runtime filtering, processing, and routing tasks. A common example is the use of

service agents to process SOAP headers.

To support SOAP header processing, the JAX-RPC API allows for the creation of specialized service agents called handlers runtime filters that exist as extensions to the J2EE container environments. Handlers can process SOAP header blocks for messages sent by J2EE service requestors or for messages received by EJB Endpoints and JAX-RPC Service Endpoints.

J2EE handlers as service agents.



Multiple handlers can be used to process different header blocks in the same SOAP message.

Platform extensions

Different vendors that implement and build around the J2EE platform offer various platform extensions in the form of SDKs that extend their development tool offering. Following are two examples of currently available platform extensions.

- IBM Emerging Technologies Toolkit A collection of extensions that provide prototype implementations of a number of fundamental WS-* extensions, including WS-Addressing, WS-ReliableMessaging, WS-MetadataExchange, and WS-Resource Framework.
- Java Web Services Developer Pack A toolkit that includes both WS-* support as well as
- the introduction of new Java APIs. Examples of the types of extensions provided include WS-Security (along with XML-Signature), and WS-I Attachments.

Primitive SOA support

The J2EE platform provides a development and runtime environment through which all primitive SOA characteristics can be realized, as follows.

Service encapsulation

Both EJBs and servlets can be encapsulated using Web services. This turns them into EJB and JAX-RPC Service Endpoints, respectively. The underlying business logic of an endpoint can further compose and interact with non-endpoint EJB and servlet components. As a result, well-defined services can be created in support of SOA.

Loose coupling

The use of interfaces within the J2EE platform allows for the abstraction of metadata from a component's actual logic. When complemented with an open or proprietary messaging technology, loose coupling can be realized.

Messaging

Prior to the acceptance of Web services, the J2EE platform supported messaging via the JMS standard, allowing for the exchange of messages between both servlets and EJB components. With the arrival of Web services support, the JAX-RPC API provides the means of enabling SOAP messaging over HTTP.

Also worth noting is the availability of the SOAP over JMS extension, which supports the delivery of SOAP messages via the JMS protocol as an alternative to HTTP. The primary benefit here is that this approach to data exchange leverages the reliability features provided by the JMS framework.

Support for service-orientation principles

We've established that the J2EE platform supports and implements the first-generation Web services technology set. It is now time to revisit the four principles of service-orientation not automatically provided by Web services and briefly discuss how each can be realized through J2EE.

Autonomy

For a service to be fully autonomous, it must be able to independently govern the processing of its underlying application logic. A high level of autonomy is more easily achieved when building Web services that do not need to encapsulate legacy logic. JAX-RPC Service Endpoints exist as standalone servlets deployed within the Web container and are generally built in support of newer SOA environments.

Reusability

The advent of Enterprise Java Beans during the rise of distributed solutions over the past decade established a componentized application design model that, along with the Java programming language, natively supports object-orientation. As a result, reusability is achievable on a component level.

Statelessness

JAX-RPC Service Endpoints can be designed to exist as stateless servlets, but the JAX-RPC API does provide the means for the servlet to manage state information through the use of the HTTPSession object. It is therefore up to the service designer to ensure that statelessness is maximized and session information is only persisted in this manner when absolutely necessary.

Discoverability

As with reuse, service discoverability requires deliberate design. To make a service discoverable, the emphasis is on the endpoint design, in that it must be as descriptive as possible.

Service discovery as part of a J2EE SOA is directly supported through JAXR, an API that provides a programmatic interface to XML-based registries, including UDDI repositories. The JAXR library consists of two separate APIs for publishing and issuing searches against registries.

Contemporary SOA support

As WS-* extensions have not yet been standardized by the vendor-neutral J2EE platform, they require the help of vendor-specific tools and features.

Based on open standards

The API specifications that comprise the J2EE platform are themselves open standards, which further promotes vendor diversity, as described in the next section.

Supports vendor diversity

Adherence to the vanilla J2EE API standards has allowed for a diverse vendor marketplace to emerge. Java application logic can be developed with one tool and then ported over to another.

Similarly, Java components can be designed for deployment mobility across different J2EE server products.

Intrinsically interoperable

Interoperability is, to a large extent, a quality deliberately designed into a Web service. Aside from service interface design characteristics, conformance to industry-standard Web services specifications is critical to achieving interoperable SOAs, especially when interoperability is required across enterprise domains.

Promotes federation

Strategically positioned services coupled with adapters that expose legacy application logic can establish a degree of federation. Building an integration architecture with custom business services and legacy wrapper services can be achieved using basic J2EE APIs and features. Supplementing such an architecture with an orchestration server further increases the potential of unifying and standardizing integrated logic.

Architecturally composable

Given the modular nature of supporting API packages and classes and the choice of service-specific containers, the J2EE platform is intrinsically composable. This allows solution designers to use only the parts of the platform required for a particular application. For example, a Web services solution that only consists of JAX-RPC Service Endpoints will likely not have a need for the JMS class packagesora J2EE SOA that does not require a service registry will not implement any part of the JAXR API.

Extensibility

As with any service-oriented solution, those based on the J2EE platform can be designed with services that support the notion of future extensibility. This comes down to fundamental design characteristics that impose conventions and structure on the service interface level.

Supports service-oriented business modeling

Beyond consistent and standardized design approaches to building service layers along the lines of the application, entity, and task-centric services we've established in previous chapters, there is no inherent support for service-oriented business modeling within J2EE. This is primarily because the concept of orchestration is not a native part of the J2EE platform.

Logic-level abstraction

JAX-RPC Service Endpoints and EJB Service Endpoints can be designed into service layers that abstract application-specific or reusable logic. Further, entire J2EE solutions can be exposed through these types of services, when appropriate.

Organizational agility and enterprise-wide loose coupling

The creation of these service layers is possible with the help of a vendor-specific orchestration server. Although the orchestration offering is proprietary, the fact that other Web services are J2EE standardized further promotes an aspect of agility realized through the vendor diverse nature of the J2EE marketplace.

JAVA API FOR XML BASED WEB SERVICES (JAX-WS)

JAX-WS is the core Java web service technology (standard for Java EE):

JAX-WS is the standard programming model / API for WS on Java (JAX-WS became a standard part of Java as of version 1.6).

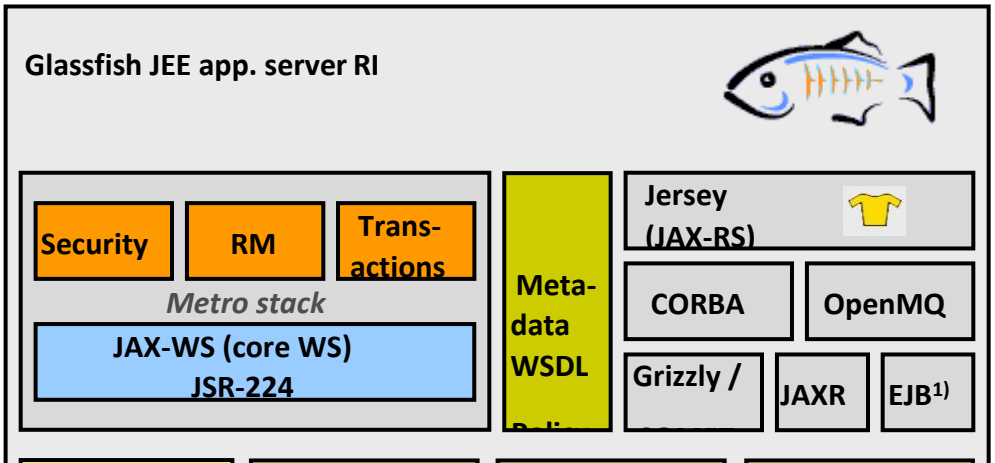
JAX-WS is platform independent (many Java platforms like Glassfish, Axis2 or CXF support JAX-WS). Services developed on one platform can be easily ported to another platform.

JAX-WS makes use of annotations like @WebService. This provides better scalability (no central deployment descriptor for different WS classes is required).

JAX-WS uses the POJO concept (use of plain Java classes to define web service interfaces).

JAX-WS replaces / supersedes JAX-RPC (= old Java web services, basically RMI over web service). **JAX-**WS is more document orientiented instead of RPC-oriented.

Glassfish is the JAX-WS reference implementation (RI, see https://jax-ws.java.net/).
Glassfish = Reference implementation for JAX-WS
Glassfish: Java EE reference implementation (RI), reference Java application server. Metro: WS stack as part of Glassfish (consists of JAX-WS and WSIT components). JAXB: Data binding (bind objects to XML documents or fragments).
SAAJ: SOAP with Attachments API for Java. JAX-WS: Java core web service stack.
WSIT: Web Services Interoperability Technologies (enables interop with .Net).
JAXR: Java API for XML Registries (WS registry).
StaX: Streaming API for XML.



Glassfish projects, Java packages (selection)

Glassfish is developed in various independent projects. Together, they build the Glassfish service platform.

Parent project	Project	Description	Java package
Glassfish	GF-CORBA	CORBA	com.sun.corba.se.*
Glassfish	OpenMQ	JMS	javax.jms.*
Glassfish	WADL	IDL for REST services	org.vnet.ws.wadl.* org.vnet.ws.wadl2jav a.*
Glassfish	JAX-RS, Jersey (=RI)	REST services	javax.ws.rs.*
jax-ws-xml, jwsdp	JAXB	Java API for XML Binding	javax.xml.bind.*
jax-ws-xml, jwsdp	JAX-RPC	Java API for XML RPC Old Java WS (superseded by JAX-WS)	java.xml.rpc.*
Metro (Glassfish)	JAX-WS	Java API for XML Web Services	javax.jws. * javax.xml
Metro (Glassfish)	SAAJ	SOAP attachments	javax.xml.soap.*

Metro (Glassfish)	WSIT (previousl y Tango)	Web Services Interoperability Technologies (WS-Trust etc.)	Various

POJO / Java bean as web service class (1/2)

Web services through simple annotations:
The programming model of JAX-WS relies on simple annotations.
This allows to turn existing business classes (POJOs) into web services with very little effort.

Web service endpoints are either explicit or implicit, depending on the definition or absence of a Java interface that defines the web service interface.

Implicit SEI (Service Endpoint Interface):
Service does not have an explicit service interface , i.e. the class itself is the service interface.

```
@WebService
public class HelloWorld
{
@WebMethod          public String sayHello() {...}

public HelloWorld() {...}

@PostConstruct      public void init() {...}

@PreDestroy
public void teardown() {...}
}
```

POJO / Java bean as web service class (2/2)

Explicit SEI:

The service bean has an associated service endpoint interface (reference to interface).

```
public interface IHello
{

Explicit endpoint interface.

@WebMethod          public String sayHello() {...}
}

 @WebService(endpointInterface=IHello)    public class HelloWorld
{
```

```
public String sayHello() {...}
}
```

Reference to explicit service endpoint interface.

JAX-WS JavaBeans versus provider endpoint (1/3)

Web services on the service provider side can be defined at 2 levels:

The annotation @WebService is a high-level annotation to turn a POJO into a web service.

The @WebServiceProvider is lower-level and gives more control over the web service.

JAX-WS JavaBeans versus provider endpoint (2/3)

A.JavaBeans endpoints (@WebService annotation):

JavaBeans as endpoints allow to expose Java classes (actually methods) as web services. This hides most of the WS complexity to the programmer.

JavaBeans endpoint example:

```
@WebService
public class HelloWorld
{
   @WebMethod
   public String sayHello() {...}
}
```

JAX-WS JavaBeans versus provider endpoint (3/3)

B.Provider endpoints (@WebServiceProvider annotation):

Provider endpoints are more generic, message-based service implementations. The service operates directly on (SOAP) message level. The service implementation defines 1 generic method "invoke" that accepts a SOAP message and returns a SOAP result. The provider interface allows to operate a web service in payload mode (service method directly receives XML messages).

Provider endpoint example:

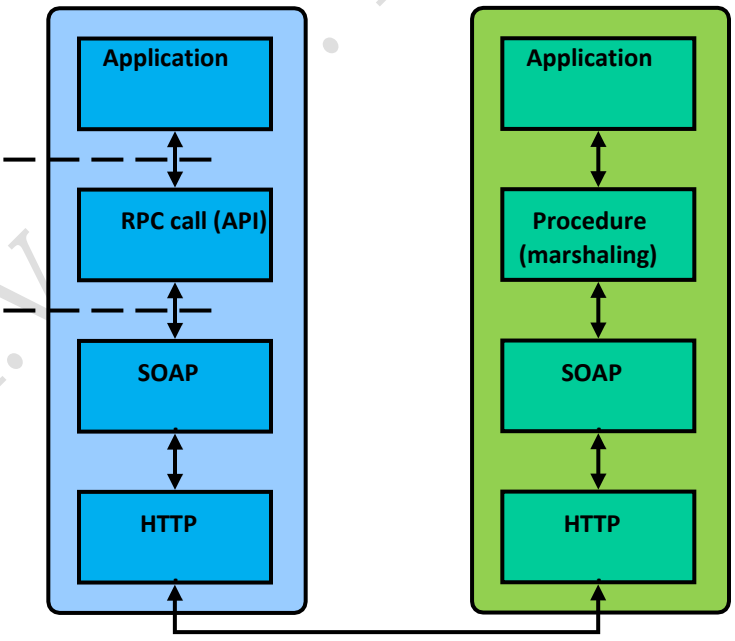
The web service class has only a single method invoke. This method receives the SOAP messages. Decoding of the message takes place in this method.

```
@WebServiceProvider(
    portName="HelloPort",
    serviceName="HelloService",

    targetNamespace="http://helloservice.org/wsdl",
    wsdlLocation="WEB-INF/wsdl/HelloService.wsdl"

)

@BindingType(value="http://schemas.xmlsoap.org/wsdl/soap/http")
```

```
@ServiceMode(value=javax.xml.ws.Service.Mode.MESSAGE)

public class HelloProvider implements Provider<SOAPMessage> {

  private static final String helloResponse = ...
  public SOAPMessage invoke(SOAPMessage req)  {

    ...

  }

}
```

JAX-WS dispatch client versus dynamic client proxy API (1/3)
Similar to the server APIs, JAX-WS clients may use 2 different APIs for sending web service requests.

- A dispatch client gives direct access to XML / SOAP messages.
- A dynamic proxy client provides an RPC-like interface to the client application.

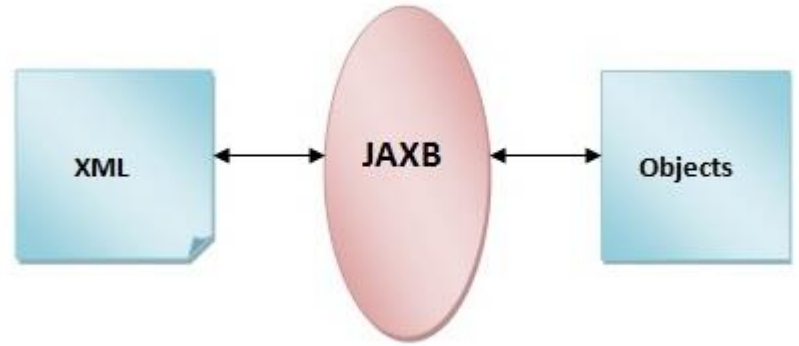


JAVA ARCHITECTURE FOR XML BINDING(JAXB)

JAXB, stands for **Java Architecture for XML Binding**, using JAXB annotation to convert Java object to / from XML file. In this tutorial, we show you how to use JAXB to do following stuffs :

1. Marshalling – Convert a Java object into a XML file.
2. Unmarshalling – Convert XML content into a Java Object.

JAXB stands for Java Architecture for XML Binding. It provides mechanism to marshal (write) java objects into XML and unmarshal (read) XML into object. Simply, you can say it is used to convert java object into xml and vice-versa.



Features of JAXB 2.0

JAXB 2.0 includes several features that were not present in JAXB 1.x. They are as follows:

1) Annotation support: JAXB 2.0 provides support to annotation so less coding is required to develop JAXB application. The javax.xml.bind.annotation package provides classes and interfaces for JAXB 2.0.

2) Support for all W3C XML Schema features: it supports all the W3C schema unlike JAXB 1.0.

3) Additional Validation Capabilities: it provides additional validation support by JAXP 1.3 validation API.

4) Small Runtime Library: it required small runtime library that JAXB 1.0.

5) Reduction of generated schema-derived classes: it reduces a lot of generated schema-derived classes.

JAXB Annotation

For object that need to convert to / from XML file, it have to annotate with JAXB annotation. The annotation are pretty self-explanatory, you can refer to this JAXB guide for detail explanation.

```
package com.mkyong.core;

import javax.xml.bind.annotation.XmlAttribute;

import javax.xml.bind.annotation.XmlElement;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement

public class Customer {

    String name;

    int age;

    int id;

    public String getName() {

            return name;

    }

    @XmlElement

    public void setName(String name) {

            this.name = name;

    }

    public int getAge() {

            return age;

    }

    @XmlElement

    public void setAge(int age) {

            this.age = age;

    }

    public int getId() {

            return id;
```

```
        }
        @XmlAttribute
        public void setId(int id) {
                this.id = id;
        }
    }
```

Simple JAXB Marshalling Example: Converting Object into XML

Let's see the steps to convert java object into XML document.

- Create POJO or bind the schema and generate the classes
- Create the JAXBContext object
- Create the Marshaller objects
- Create the content tree by using set methods
- Call the marshal method

Simple JAXB UnMarshalling Example: Converting Object into XML

By the help of **UnMarshaller** interface, we can unmarshal(read) the object into xml document.

In this example, we are going to convert simple xml document into java object.

Let's see the steps to convert XML document into java object.

- Create POJO or bind the schema and generate the classes
- Create the JAXBContext object
- Create the Unmarshaller objects
- Call the unmarshal method
- Use getter methods of POJO to access the data

JAXB Annotation

For object that need to convert to / from XML file, it have to annotate with JAXB annotation. The annotation are pretty self-explanatory, you can refer to this JAXB guide for detail explanation.

```
package com.mkyong.core;
```

```
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement
public class Customer {
    String name;
    int age;
    int id;
    public String getName() {
        return name;
    }
    @XmlElement
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    @XmlElement
    public void setAge(int age) {
        this.age = age;
    }
    public int getId() {
        return id;
    }
    @XmlAttribute
    public void setId(int id) {
        this.id = id;
    }
}
```


JAVA API FOR XML REGISTRIES(JAXR)

The Java API for XML Registries (JAXR) provides a convenient way to access standard business registries over the Internet. Business registries are often described as electronic yellow pages because they contain listings of businesses and the products or services the businesses offer.

Registries facilitate the creation of business relationships by providing an independent online information exchange service that allows service providers (e.g. sellers) to advertise their products and services, and service consumers (e.g. buyers) to discover these products and services.

JAXR gives developers writing applications in the Java programming language a uniform way

to use business registries that are based on open standards (such as ebXML) or industry consortium-led specifications (such as UDDI).

Businesses can register themselves with a registry or discover other businesses with which they might want to do business. In addition, they can submit material to be shared and search for material that others have submitted. Standards groups have developed DTDs for particular kinds of XML documents, and two businesses might, for example, agree to use the DTD for their industry's standard purchase order form.

Because the DTD is stored in a standard business registry, both parties can use JAXR to access it. Registries are becoming an increasingly important component of web services because they allow businesses to collaborate with each other dynamically in a loosely coupled way. Accordingly, the need for JAXR, which enables enterprises to access standard business registries from the Java programming language, is also growing.

Registry standards

UDDI (Universal Description, Discovery, and Integration)

UDDI can be considered as a telephone book with white, yellow and green pages.

Figure 1: UDDI structure

The UDDI specification defines how Web Services can be published und found in UDDI Business Registries. These specifications are based upon XML data structures and SOAP communication APIs.

ebXML (electronic business XML)

ebXML can be considered as a cheap alternative to EDI (Electronic Data Interchange). It is based upon XML and usable over the internet. It defines electronic business including the communication specification and the profiles of business partners. It is separated in registry und repository. The registry references to data, the repository saves data.

API Overview

The Java API for XML Registries (JAXR) provides a uniform and standard Java API for accessing different kinds of XML Registries. An XML registry is an enabling infrastructure for building, deploying, and discovering Web services. Currently there are a variety of specifications for XML registries including, pre- eminently, the ebXML Registry and Repository standard, which is being developed by OASIS and U.N./CEFACT and the UDDI specification, which is being developed by a vendor consortium.

JAXR enables Java software programmers to use a single, easy-to-use abstraction API to access a variety of XML registries. Simplicity and ease of use are facilitated within JAXR by a unified JAXR information model, which describes content and metadata within XML registries.

JAXR provides rich metadata capabilities for classification and association, as well as rich query capabilities. As an abstraction-based API, JAXR gives developers the ability to write registry client programs that are portable across different target registries. This is consistent with the Java philosophy of Write Once, Run Anywhere. Similarly, JAXR also enables value-added capabilities beyond those of the underlying registries.

The actual version of the JAXR specification (version 1.0) includes detailed bindings between the JAXR information model and both the ebXML Registry and the UDDI Registry v2.0 specifications.

JAXR works in synergy with related Java APIs for XML, such as Java API for XML Processing (JAXP), Java Architecture for XML Binding (JAXB), Java API for XML-based RPC (JAX-RPC), and Java API for XML Messaging (JAXM), to enable web services within the Java 2 Platform, Enterprise Edition (J2EE).

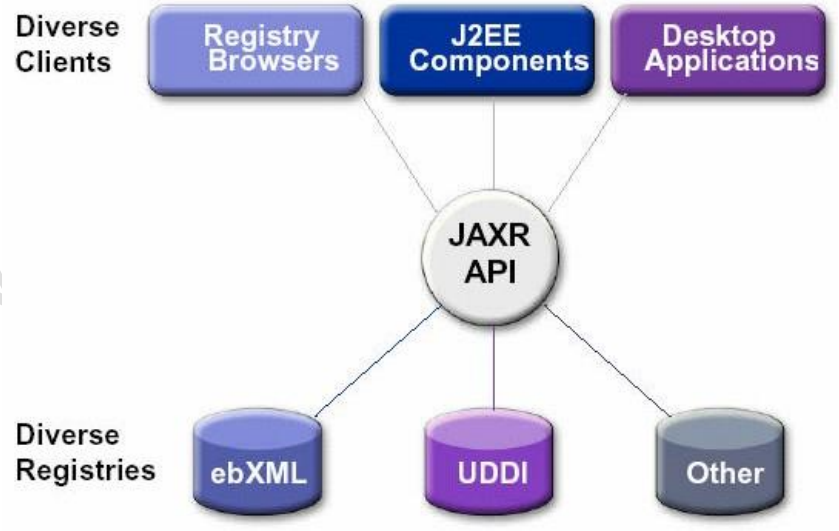
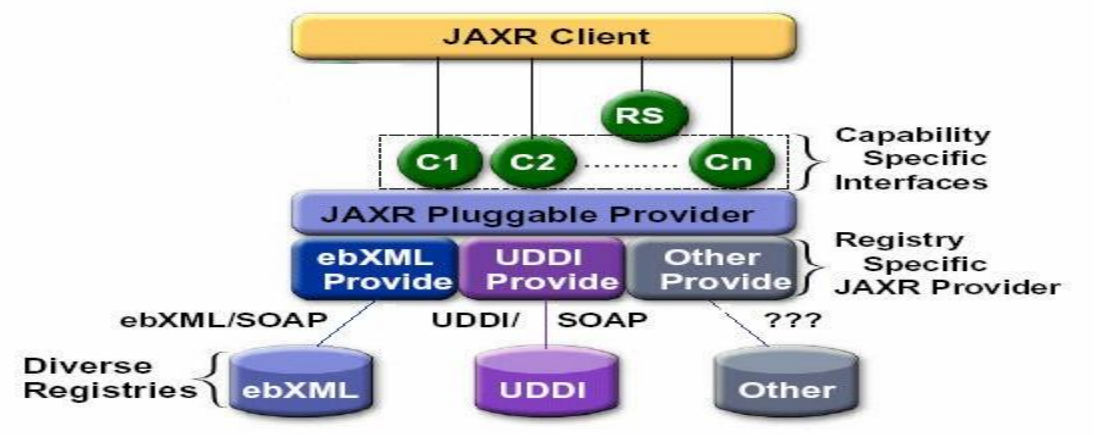


Figure 2: Interoperability between diverse JAXR clients and diverse registries

Architecture

Overview

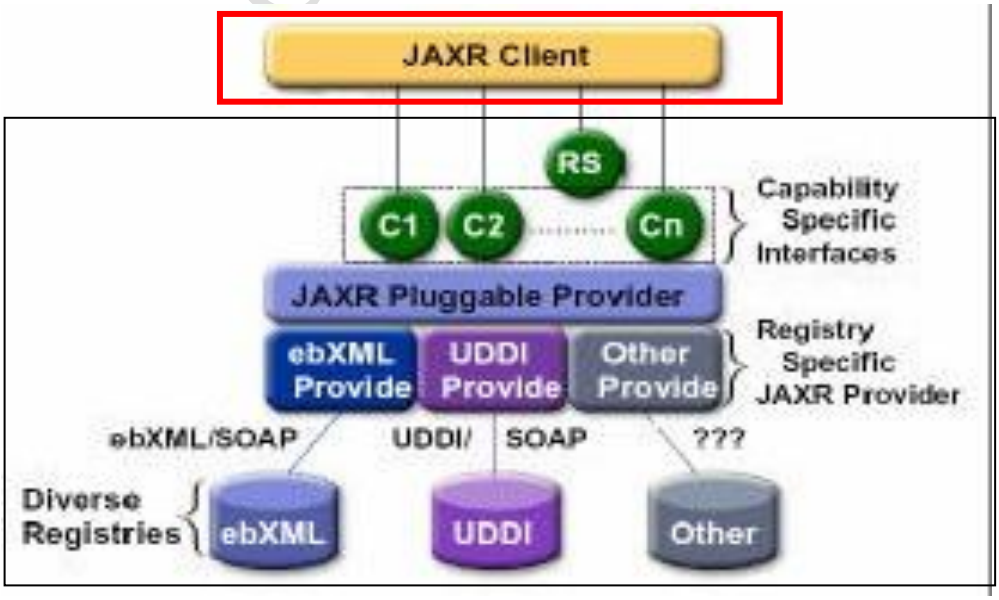
Figure 3 JAXR Architecture



The circles represent the various interfaces implemented by the JAXR client and the JAXR provider:

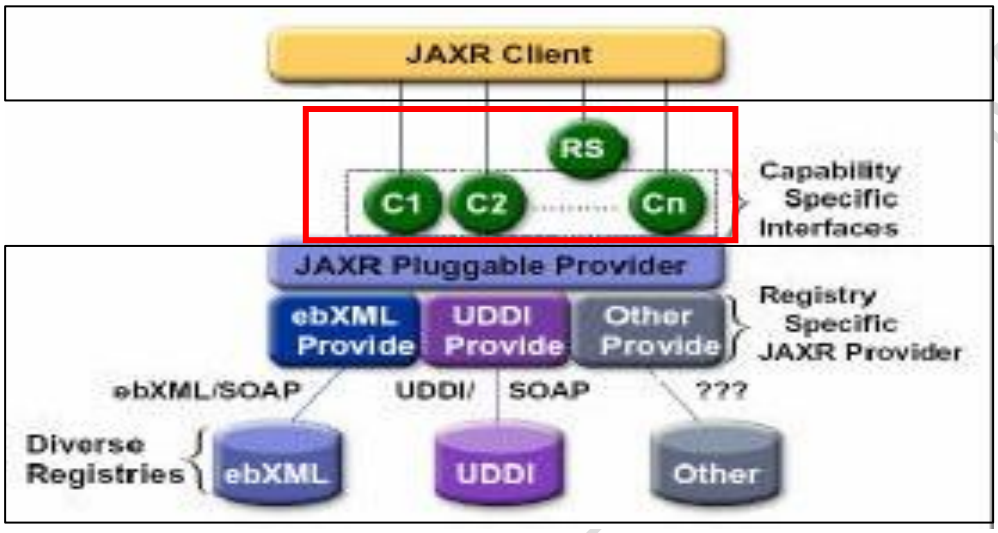
- RS represents the RegistryService interface implemented by the JAXR provider.
- C1, C2 through Cn represent the JAXR interfaces implemented by the JAXR provider that provide the various registry capabilities.

JAXR Client



The JAXR client may be any standalone Java application or an enterprise component based on J2EE technology. The JAXR client uses the JAXR API to access a registry via a JAXR provider.

Interface Connection & Interface Registry Service

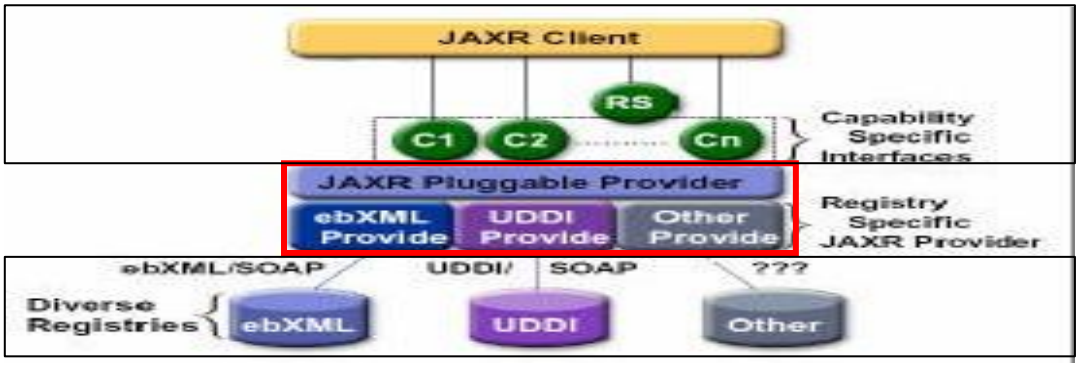


Client must create a JAXR Connection to a registry provider using an appropriate JAXR provider in order to employ the services of that registry using the JAXR API.

The *RegistryService* interface is the principal interface implemented by a JAXR provider. A registry client can get this interface from its connection to a JAXR provider.

The *RegistryService* interface provides the getter methods that are used by the client to discover various capability-specific interfaces implemented by the JAXR provider. It also provides a *getCapabilityProfile* method that allows the JAXR client to access the capability profile that describes the capabilities supported by the JAXR provider.

The JAXR Provider



The JAXR provider is an implementation of the JAXR API.
A JAXR client accesses a registry via a JAXR provider.

JAXR Pluggable provider

The JAXR Pluggable provider implements features of the JAXR API that are independent of any specific registry type.

registry-specific JAXR

The registry-specific JAXR providers implement the JAXR API in a registry- specific manner. A registry-specific JAXR provider plugs into the JAXR Pluggable provider and is used by the JAXR Pluggable provider in a delegation pattern.

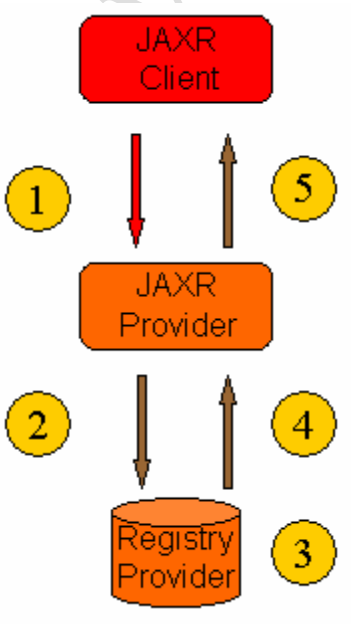
Registry Provider

These are implementations of various registry specifications such as ebXML and UDDI.

Benefits of JAXR Architecture

1. If a new registry standard is developed, it can be included in the JAXR provider implementation without having to change the JAXR client.
2. A JAXR client is capable to communicate with several registries in one session.
3. Additional capability profiles allow the limitation or expansion of the functionality of a JAXR provider.

JAXR Interaction steps



1. JAXR Client sends a request

A JAXR Client uses JAXR interfaces and classes to access a registry. The client sends the request to a JAXR Provider.

2. When a JAXR Provider receives a request from a JAXR client, he transforms the request into an equivalent request, which is based upon the specification of the target registry. The JAXR provider now sends this request to a registry provider.
3. The registry provid er receives the request from the JAXR provider and processes it.
4. The registry provider sends a reply to the JAXR

provider which is transformed into a JAXR reply.

5. The JAXR provider sends the JAXR reply to the JAXR client.

JAXR provider

The JAXR provider implements two main packages:

- *javax.xml.registry*

This package includes API interfaces and classes which define interfaces for accessing the registry.

- *javax.xml.registry.infomodel*

This package includes interfaces which define the JAXR information model.

javax.xml.registry

The javax.xml.registry packet has the following basic interfaces:

- *Connection*

The client needs to establish a connection to the JAXR Provider in order to use the registry. The connection represents a client session with a registry provider.

- *RegistryService*

The client receives a *RegistryService* object from his connection. Furthermore the *RegistryService* provides other interfaces to the client for accessing the registry.

The javax.xml.registry packet has the following primary interfaces:

- *BusinessQueryManager*

This interface allows the client to search the registry for information.*BusinessLifeCycleManager*

This interface allows the client to modify information in a registry through saving or deleting.

- *DeclarativeQueryManager*

This interface enables the use of SQL syntax in queries for the client (not implemented at the moment).

javax.xml.registry.infomodel

The registry javax.xml.registry.infomodel contains the interfaces which define the JAXR information modelThese interfaces define the types of objects in a registry and their relations. The elemental interfaces in this package is the *RegistryObject* interface with the subinterfaces *Organization*, *Service* and *ServiceBinding*.

JAXR Exception handling

If an error occurs a JAXR API method or one of its subclasses is called. Some methods in the JAXR API use a collection as argument or return value. This enables the parallel use of several registry objects.

JAVA API FOR XML BASED RPC(JAX-RPC)

Overview of JAX-RPC

An RPC-based Web service is a collection of procedures that can be called by a remote client over the Internet. For example, a typical RPC-based Web service is a stock quote service that takes a SOAP (Simple Object Access Protocol) request for the price of a specified stock and returns the price via SOAP.

A Web service, a server application that implements the procedures that are available for clients to call, is deployed on a server-side container. The container can be a servlet container such as Tomcat or a Java™ 2 Platform, Enterprise Edition (J2EE™) container that is based on Enterprise JavaBeans™ (EJB™) technology.

A Web service can make itself available to potential clients by describing itself in a Web Services Description Language (WSDL) document. A WSDL description is an XML document that gives all the pertinent information about a Web service, including its name, the operations that can be called on it, the parameters for those operations, and the location of where to send requests.

A consumer (Web client) can use the WSDL document to discover what the service offers and how to access it. How a developer can use a WSDL document in the creation of a Web service is discussed later.

Interoperability

Perhaps the most important requirement for a Web service is that it be interoperable across clients and servers. With JAX-RPC, a client written in a language other than the Java programming language can access a Web service developed and deployed on the Java platform.

Conversely, a client written in the Java programming language can communicate with a service that was developed and deployed using some other platform.

What makes this interoperability possible is JAX-RPC's support for SOAP and WSDL. SOAP defines standards for XML messaging and the mapping of data types so that applications adhering to these standards can communicate with each other.

JAX-RPC adheres to SOAP standards, and is, in fact, based on SOAP messaging. That is, a JAX-RPC remote procedure call is implemented as a request-response SOAP message.

The other key to interoperability is JAX-RPC's support for WSDL. A WSDL description, being an XML document that describes a Web service in a standard way, makes the description portable. WSDL documents and their uses will be discussed more later.

Ease of Use

Given the fact that JAX-RPC is based on a remote procedure call (RPC) mechanism, it is remarkably developer friendly. RPC involves a lot of complicated infrastructure, or "plumbing," but JAX-RPC mercifully makes the underlying implementation details invisible to both the client and service developer. For example, a Web services client simply makes Java method calls, and all the internal marshalling, unmarshalling, and transmission details are taken care of automatically. On the server side, the Web service simply implements the services it offers and, like the client, does not need to bother with the underlying implementation mechanisms.

Largely because of its ease of use, JAX-RPC is the main Web services API for both client and server applications. JAX-RPC focuses on point-to-point SOAP messaging, the basic mechanism that most clients of Web services use.

Although it can provide asynchronous messaging and can be extended to provide higher quality support, JAX-RPC concentrates on being easy to use for the most common tasks. Thus, JAX-RPC is a good choice for applications that wish to avoid the more complex aspects of SOAP messaging and for those that find communication using the RPC model a good fit. The more heavy-duty alternative for SOAP messaging, the Java™ API for XML Messaging (JAXM), is discussed later in this introduction.

Advanced Features

Although JAX-RPC is based on the RPC model, it offers features that go beyond basic RPC. For one thing, it is possible to send complete documents and also document fragments. In addition, JAX-RPC supports SOAP message handlers, which make it possible to send a wide variety of messages. And JAX-RPC can be extended to do one-way messaging in addition to the request-response style of messaging normally done with RPC. Another advanced feature is extensible type mapping, which gives JAX-RPC still more flexibility in what can be sent.

The Java API for XML-based RPC (page 305) (JAX-RPC) makes it possible to write an application in the Java programming language that uses SOAP to make a remote procedure call (RPC).

JAX-RPC can also be used to send request-response messages and, in some cases, one-way messages.

In addition to these conventional uses, JAX-RPC makes it possible for an application to define its own XML schema and to use that schema to send XML documents and XML fragments.

The result of this combination of JAX-RPC and XML Schema is a powerful computing tool.The Java programming language already has two other APIs for making remote procedure calls, Java IDL and Remote Method Invocation (RMI).

All three have an API for marshalling and unmarshalling arguments and for transmitting and receiving procedure calls. The difference is that JAX-RPC is based on SOAP and is geared to Web services. Java IDL is based on the Common Object Request Broker Architecture (CORBA) and uses the Object Management Group's Interface Definition Language (OMG IDL).

RMI is based on RPC where both the method calls and the methods being invoked are in the Java programming language—although with RMI over IIOP, the methods being invoked may be in another language. Sun will continue its support of CORBA and RMI in addition to developing JAX-RPC, as each serves a distinct need and has its own set of users.

All varieties of RPC are fairly complex underneath, involving the mapping and reverse mapping of data types and the marshalling and unmarshalling of arguments. However, these take place behind the scenes and are not visible to the user.

JAX-RPC continues this model, which means that a client using XML-based RPC can invoke remote procedures or do SOAP messaging by simply making Java method calls.

Using JAX-RPC

JAX-RPC makes using a Web service easier, and it also makes developing a Web service easier, especially if you use the J2EE platform. An RPC-based Web service is basically a collection of procedures that can be called by a remote client over the Internet.

The service itself is a server application deployed on a server-side container that implements the procedures that are available for clients to call. For example, a typical RPC-based Web service is a stock quote service that takes a SOAP request for the price of a specified stock and returns the price via SOAP.

A Web service needs to make itself available to potential clients, which it can do, for instance, by describing itself using the Web Services Description Language (WSDL). A consumer (Web client) can then do a lookup of the WSDL document to access the service.

Interoperability across clients and servers that have been described using WSDL is key to JAX-RPC. A consumer using the Java programming language can use JAX-RPC to send its request to a service that may or may not have been defined and deployed on a Java platform.

The converse is also possible, that is, a client using another programming language can send its request to a service that has been defined and deployed on a Java platform. This interoperability is a primary strength of JAX-RPC.

Although JAX-RPC implements a remote procedure call as a request-response SOAP message, a user of JAX-RPC is shielded from this level of detail. So, underneath the covers, JAX-RPC is based on SOAP messaging.

JAX-RPC is the main client and server Web services API, largely because of its simplicity. The JAX-RPC API is simple to use and requires no set up.

Also, JAX-RPC focuses on point-to-point SOAP messaging, the basic mechanism that most Web services clients use. Although it can provide asynchronous messaging and can be extended to provide higher quality support, JAX-RPC concentrates on being easy to use for the most common tasks.

Thus, JAX-RPC is a good choice for applications that wish to avoid the more complex aspects of SOAP messaging and for those that find communication using the RPC model a good fit.

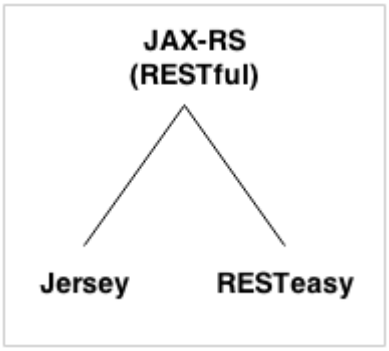
JAX-RPC is not yet final. The specification is still being fine tuned, and the latest draft includes features such as interceptors and Servlet JAX-RPC endpoints. In future releases of the Java WSDP, this introductory overview will be expanded to reflect JAX-RPC more fully.

JAVA API FOR RESTFUL WEB SERVICES(JAX-RS)

Java API for RESTful Web Services (**JAX-RS**), is a set if APIs to developer REST service. JAX-RS is part of the Java EE6, and make developers to develop REST web application easily.

There are two main implementation of JAX-RS API.

1. Jersey
2. RESTEasy



JAX-RS Example Jersey

We can create JAX-RS example by jersey implementation. To do so, you need to load jersey jar files or use maven framework.

Sample JAX-RS Server code:

```
1.  import javax.ws.rs.GET;
2.  import javax.ws.rs.Path;
3.  import javax.ws.rs.Produces;
4.  import javax.ws.rs.core.MediaType;
5.  @Path("/hello")
6.  public class Hello {
7.    // This method is called if HTML and XML is not requested
8.    @GET
9.    @Produces(MediaType.TEXT_PLAIN)
10.   public String sayPlainTextHello() {
11.     return "Hello Jersey Plain";
12.   }
13.   // This method is called if XML is requested
14.   @GET
15.   @Produces(MediaType.TEXT_XML)
16.   public String sayXMLHello() {
17.     return "<?xml version=\"1.0\"?>" + "<hello> Hello Jersey" + "</hello>";
18.   }
19.
```

In this example, we are using jersey jar files for using jersey example for JAX-RS.

JAX-RS Client Code

The ClientTest.java file is created inside the server application. But you can run client code by other application also by having service interface and jersey jar file.

Sample Client code:

```
1.  import java.net.URI;
2.  import javax.ws.rs.client.Client;
3.  import javax.ws.rs.client.ClientBuilder;
4.  import javax.ws.rs.client.WebTarget;
5.  import javax.ws.rs.core.MediaType;
6.  import javax.ws.rs.core.UriBuilder;
7.  import org.glassfish.jersey.client.ClientConfig;
8.  public class ClientTest {
9.    public static void main(String[] args) {
10.     ClientConfig config = new ClientConfig();
11.     Client client = ClientBuilder.newClient(config);
12.     WebTarget target = client.target(getBaseURI());
13.     //Now printing the server code of different media type
14.     System.out.println(target.path("rest").path("hello").request().accept(MediaType.TEXT
_PLAIN).get(String.class));
15.     System.out.println(target.path("rest").path("hello").request().accept(MediaType.TEXT
_XML).get(String.class));
16.     System.out.println(target.path("rest").path("hello").request().accept(MediaType.TEXT
_HTML).get(String.class));
17.   }
```

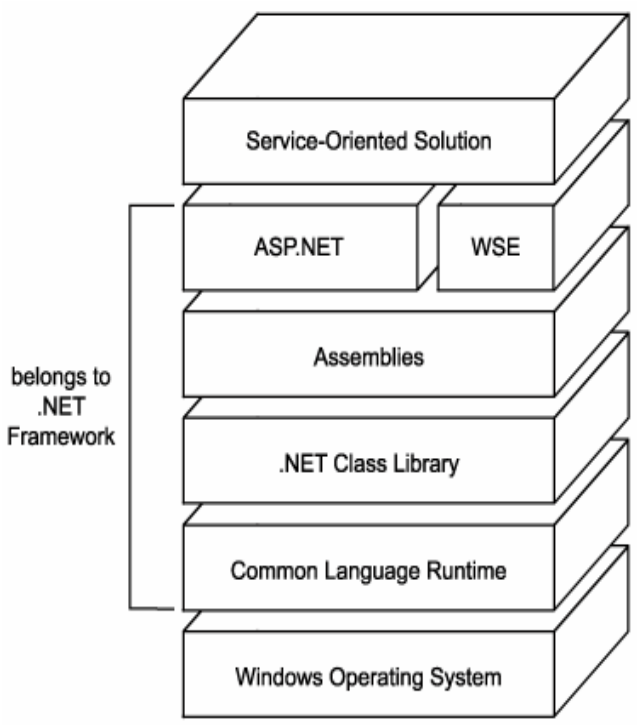
SOA SUPPORT IN .NET

Platform overview

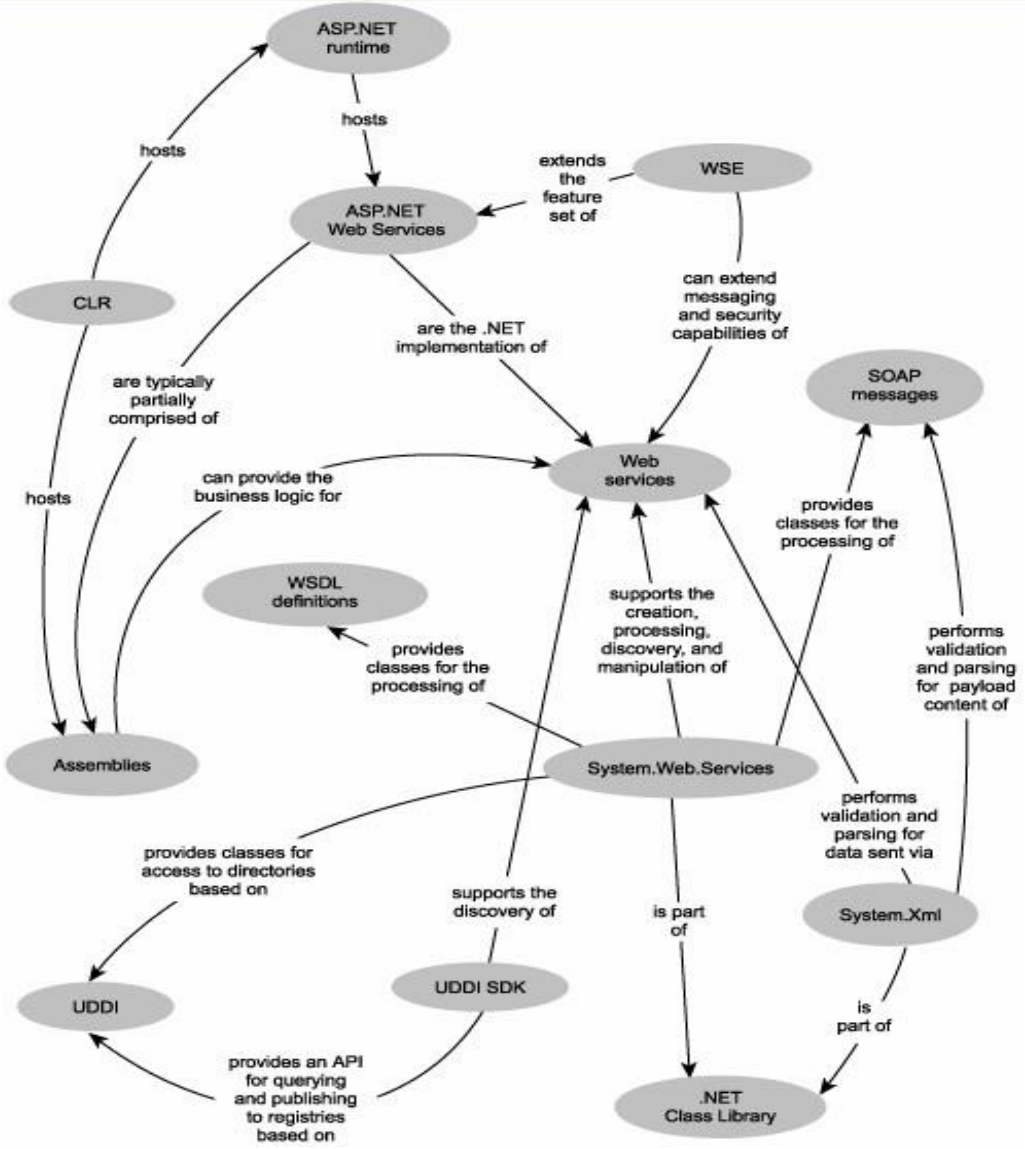
The .NET framework is a proprietary solution runtime and development platform designed for use with Windows operating systems and server products. The .NET platform can be used to deliver a variety of applications, ranging from desktop and mobile systems to distributed Web solutions and Web services.

A primary part of .NET relevant to SOA is the ASP.NET environment, used to deliver the Web Technology layer within SOA (and further supplemented by the Web Services Enhancements (WSE) extension).

Relevant layers of the .NET framework, as they relate to SOA.



How parts of the .NET framework inter-relate.



Architecture components

The .NET framework provides an environment designed for the delivery of different types of distributed solutions. Listed here are the components most associated with Web-based .NET applications:

- ASP.NET Web Forms - These are dynamically built Web pages that reside on the Web server and support the creation of interactive online forms through the use of a series of server-side controls responsible for auto-generating Web page content.

- ASP.NET ASP.NET Web Services An ASP.NET application designed as a service provider that also resides on the Web server.

- Assemblies An assembly is the standard unit of processing logic within the .NET environment.
  Web Forms can be used to build the presentation layer of a service-oriented solution, but it is the latter two components that are of immediate relevance to building Web services.

Runtime environments

The architecture components previously described rely on the Common Language Runtime (CLR) provided by the .NET framework. CLR supplies a collection of runtime agents that provide a number of services for managing .NET applications, including cross-language support, central data typing, and object lifecycle and memory management.

Programming languages

The .NET framework provides unified support for a set of programming languages, including Visual Basic, C++, and the more recent C#. The .NET versions of these languages have been designed in alignment with the CLR. This means that regardless of the .NET language used, programming code is converted into a standardized format known as the Microsoft Intermediate Language (MSIL). It is the MSIL code that eventually is executed within the CLR.

APIs

.NET provides programmatic access to numerous framework (operating system) level functions via the .NET Class Library, a large set of APIs organized into namespaces. Each namespace must be explicitly referenced for application programming logic to utilize its underlying features.

Following are examples of the primary namespaces that provide APIs relevant to Web services development:

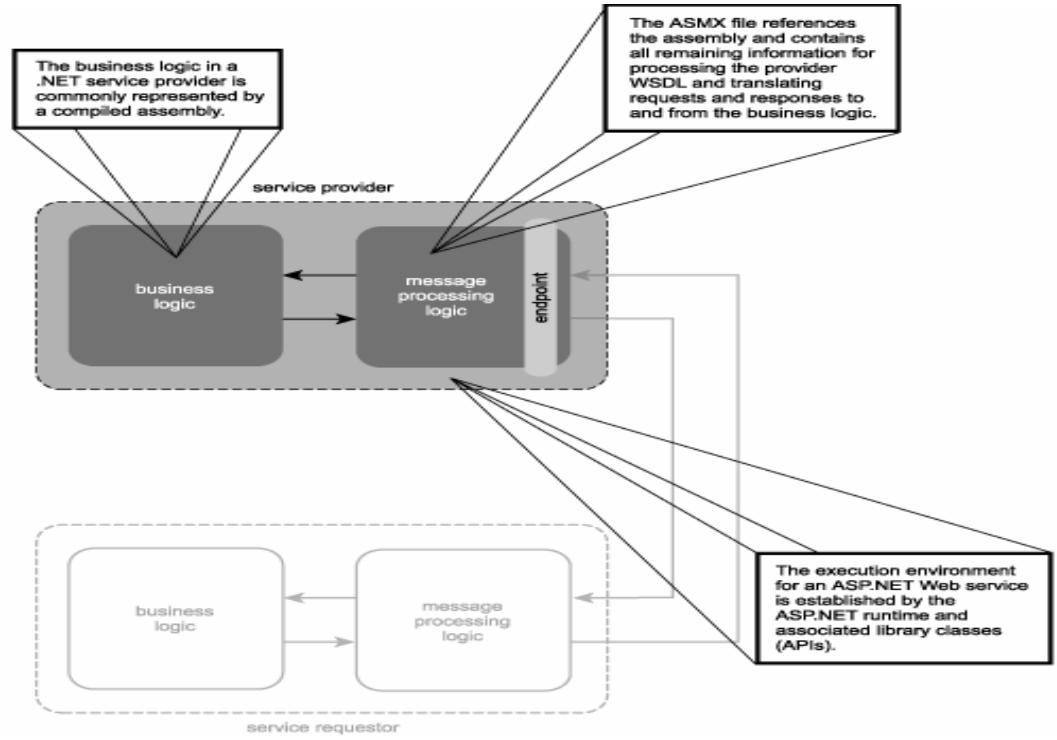
- System.Xml Parsing and processing functions related to XML documents are provided by this collection of classes. Examples include:

- System.Web.Services This library contains a family of classes that break down the various documents that comprise and support the Web service interface and interaction layer on the Web server into more granular classes. For example:

Service providers

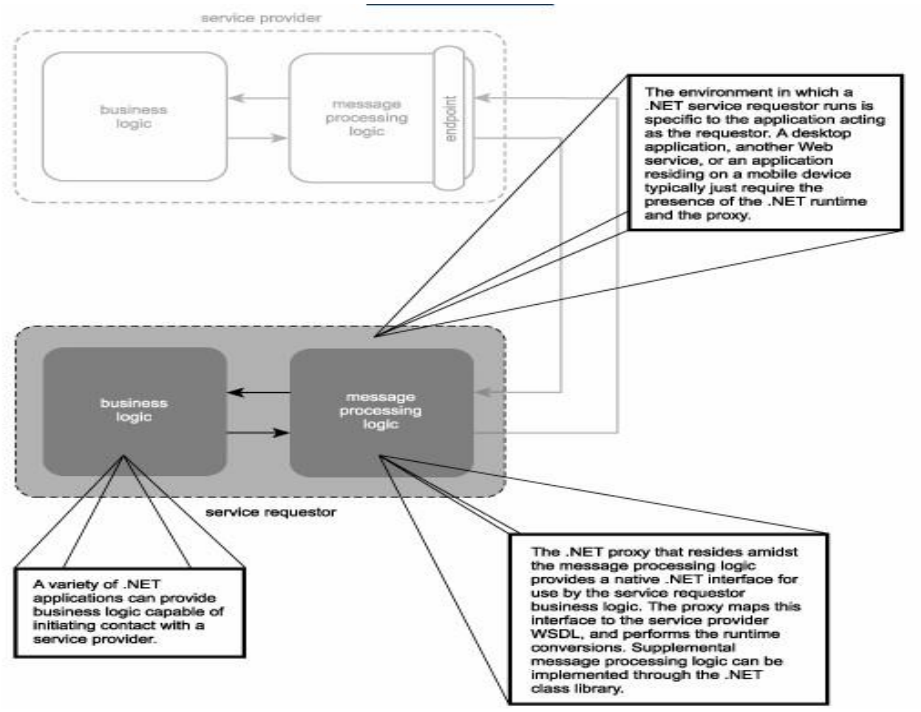
.NET service providers are Web services that exist as a special variation of ASP.NET applications, called ASP.NET Web Services.

A typical .NET service provider.



Service requestors

To support the creation of service requestors, .NET provides a proxy class that resides alongside the service requestor's application logic and duplicates the service provider interface. This allows the service requestor to interact with the proxy class locally, while delegating all remote processing and message marshalling activities to the proxy logic.



Service agents

The ASP.NET environment utilizes many system-level agents that perform various runtime processing tasks. As mentioned earlier, the ASP.NET runtime outfits the HTTP Pipeline with a series of HTTP Modules These service agents are capable of performing system tasks such as authentication, authorization, and state management. Custom HTTP Modules also can be created to perform various processing tasks prior and subsequent to endpoint contact.

Platform extensions

The Web Services Enhancements (WSE) is a toolkit that establishes an extension to the .NET framework providing a set of supplementary classes geared specifically to support key WS-* specification features. It is designed for use with Visual Studio and currently promotes support for the following WS-* specifications: WS-Addressing, WS-Policy, WS-Security (including WS-SecurityPolicy, WS-SecureConversation, WS-Trust), WS-Referral, and WS-Attachments

and DIME (Direct Internet Message Encapsulation).

Primitive SOA support

The .NET framework natively supports primitive SOA characteristics through its runtime environment and development tools, as explained here.

Service encapsulation

Through the creation of independent assemblies and ASP.NET applications, the .NET framework supports the notion of partitioning application logic into atomic units. This promotes the componentization of solutions, which has been a milestone design quality of traditional distributed applications for some time.

Loose coupling

The .NET environment allows components to publish a public interface that can be discovered and accessed by potential clients. When used in conjunction with a messaging framework, such as the one provided by Microsoft Messaging Queue (MSMQ), a loosely coupled relationship between application units can be achieved.

Messaging

When the .NET framework first was introduced, it essentially overhauled Microsoft's previous distributed platform known as the Distributed Internet Architecture (DNA). As part of both the DNA and .NET platforms, the MSMQ extension (and associated APIs) supports a messaging framework that allows for the exchange of messages between components.

Support for service-orientation principles

The four principles we identified in Chapter 8 as being those not automatically provided by first- generation Web services technologies are the focus of this section, as we briefly highlight relevant parts of the .NET framework that directly or indirectly provide support for their fulfillment.

Autonomy

The .NET framework supports the creation of autonomous services to whatever extent the underlying logic permits it. When Web services are required to encapsulate application logic already residing in existing legacy COM components or assemblies designed as part of a traditional distributed solution, acquiring explicit functional boundaries and self-containment may be difficult.

Reusability

As with autonomy, reusability is a characteristic that is easier to achieve when designing the Web service application logic from the ground up. Encapsulating legacy logic or even exposing entire applications through a service interface can facilitate reuse to whatever extent the underlying logic permits it. Therefore, reuse can be built more easily into

ASP.NET Web Services and any supporting assemblies when developing services as part of newer solutions.

Statelessness

ASP.NET Web Services are stateless by default, but it is possible to create stateful variations. By setting an attribute on the service operation (referred to as the WebMethod) called EnableSession, the ASP.NET worker process creates an HttpSessionState object when that operation is invoked. State management therefore is permitted, and it is up to the service designer to use the session object only when necessary so that statelessness is continually emphasized.

Discoverability

Making services more discoverable is achieved through proper service endpoint design. Because WSDL definitions can be customized and used as the starting point of an ASP.NET Web Service, discoverability can be addressed, as follows:

- The programmatic discovery of service descriptions and XSD schemas is supported through the classes that reside in the System.Web.Services.Discovery namespace. The

.NET framework also provides a separate UDDI SDK.

- .NET allows for a separate metadata pointer file to be published alongside Web services, based on the proprietary DISCO file format. This approach to discovery is further supported via the Disco.exe command line tool, typically used for locating and discovering services within a server environment.

Contemporary SOA support

Keeping in mind that one of the contemporary SOA characteristics we identified that SOA is still evolving, a number of the following characteristics are addressed by current and maturing .NET framework features and .NET technologies.

Based on open standards

The .NET Class Library that comprises a great deal of the .NET framework provides a number of namespaces containing collections of classes that support industry standard, first-generation Web services specifications.

Supports vendor diversity

Because ASP.NET Web Services are created to conform to industry standards, their use supports vendor diversity on an enterprise level. Other non-.NET SOAs can be built around a .NET SOA, and interoperability will still be a reality as long as all exposed Web services comply to common standards (as dictated by the Basic Profile, for example).

Intrinsically interoperable

Version 2.0 of the .NET framework, along with Visual Studio 2005, provides native support for the WS-I Basic Profile. This means that Web services developed using Visual Studio 2005 are Basic Profile compliant by default.

Promotes federation

Although technically not part of the .NET framework, the BizTalk server platform can be considered an extension used to achieve a level of federation across disparate enterprise environments.

Architecturally composable

The .NET Class Library is an example of a composable programming model, as classes provided are functionally granular. Therefore, only those functions actually required by a Web service are imported by and used within its underlying business logic.

Extensibility

ASP.NET Web Services subjected to design standards and related best practices will benefit from providing extensible service interfaces and extensible application logic (implemented via assemblies with service-oriented class designs). Therefore, extensibility is not a direct feature of the .NET framework, but more a common sense design approach to utilizing .NET technology.

ASP.NET WEB SERVICES

A web service is a web-based functionality accessed using the protocols of the web to be used by the web applications. There are three aspects of web service development:

- Creating the web service
- Creating a proxy
- Consuming the web service

Creating a Web Service

A web service is a web application which is basically a class consisting of methods that could be used by other applications. It also follows a code-behind architecture such as the ASP.NET web pages, although it does not have a user interface.

To understand the concept let us create a web service to provide stock price information. The clients can query about the name and price of a stock based on the stock symbol. To keep this example simple, the values are hardcoded in a two-dimensional array. This web service has three methods:

- A default HelloWorld method
- A GetName Method
- A GetPrice Method

Take the following steps to create the web service:

Step (1) : Select File -> New -> Web Site in Visual Studio, and then select ASP.NET Web Service.

Step (2) : A web service file called Service.asmx and its code behind file, Service.cs is created in the App_Code directory of the project.

Step (3) : Change the names of the files to StockService.asmx and StockService.cs.

Step (4) : The .asmx file has simply a WebService directive on it:

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/StockService.cs"
Class="StockService" %>
```

Step (5) : Open the StockService.cs file, the code generated in it is the basic Hello World service. The default web service code behind file looks like the following:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Linq;

using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;

using System.Xml.Linq;

namespace StockService
```

```
{
  // <summary>
  // Summary description for Service1
  // <summary>

  [WebService(Namespace = "http://tempuri.org/")]
  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
  [ToolboxItem(false)]

  // To allow this Web Service to be called from script,
  // using ASP.NET AJAX, uncomment the following line.
  // [System.Web.Script.Services.ScriptService]

  public class Service1 : System.Web.Services.WebService
  {
    [WebMethod]

    public string HelloWorld()
    {
      return "Hello World";
    }
  }
}
```

Step (6) : Change the code behind file to add the two dimensional array of strings for stock symbol, name and price and two web methods for getting the stock information.

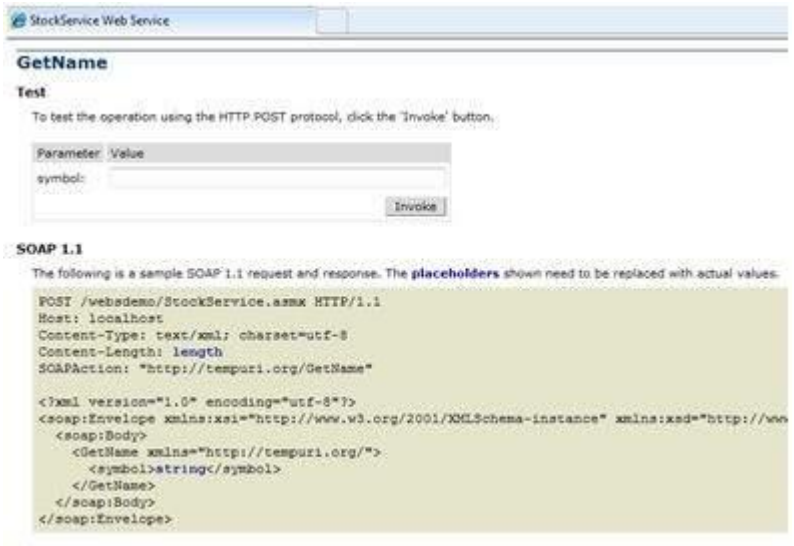
```
using System;
using System.Linq;

using System.Web;
using System.Web.Services;
```

```
using System.Web.Services.Protocols;

using System.Xml.Linq;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]

// To allow this Web Service to be called from script,
// using ASP.NET AJAX, uncomment the following line.
// [System.Web.Script.Services.ScriptService]

public class StockService : System.Web.Services.WebService
{
  public StockService () {
    //Uncomment the following if using designed components
    //InitializeComponent();
  }

  string[,] stocks =
  {
    {"RELIND", "Reliance Industries", "1060.15"},
    {"ICICI", "ICICI Bank", "911.55"},
    {"JSW", "JSW Steel", "1201.25"},
    {"WIPRO", "Wipro Limited", "1194.65"},
    {"SATYAM", "Satyam Computers", "91.10"}
  };

  [WebMethod]
  public string HelloWorld() {
    return "Hello World";
  }

  [WebMethod]
  public double GetPrice(string symbol)
  {
    //it takes the symbol as parameter and returns price
    for (int i = 0; i < stocks.GetLength(0); i++)
    {
      if (String.Compare(symbol, stocks[i, 0], true) == 0)
      return Convert.ToDouble(stocks[i, 2]);
    }

    return 0;
  }

  [WebMethod]
```

```
  public string GetName(string symbol)
  {
    // It takes the symbol as parameter and
    // returns name of the stock
    for (int i = 0; i < stocks.GetLength(0); i++)
    {
      if (String.Compare(symbol, stocks[i, 0], true) == 0)
      return stocks[i, 1];
    }

    return "Stock Not Found";
  }
}
```

Step (7) : Running the web service application gives a web service test page, which allows testing the service methods.

Step (8) : Click on a method name, and check whether it runs properly.



Step (9) : For testing the GetName method, provide one of the stock symbols, which are hard coded, it returns the name of the stock

Consuming the Web Service

For using the web service, create a web site under the same solution. This could be done by right clicking on the Solution name in the Solution Explorer. The web page calling the web service should have a label control to display the returned results and two button controls one for post back and another for calling the service.

The content file for the web application is as follows:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="wsclient._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

  <head runat="server">
    <title>
      Untitled Page
    </title>
```

```
  </head>

  <body>

    <form id="form1" runat="server">
      <div>

        <h3>Using the Stock Service</h3>

        <br /> <br />

        <asp:Label ID="lblmessage" runat="server"></asp:Label>

        <br /> <br />

        <asp:Button ID="btnpostback" runat="server" onclick="Button1_Click" Text="Post Back"
style="width:132px" />

        <asp:Button ID="btnservice" runat="server" onclick="btnservice_Click"  Text="Get
Stock" style="width:99px" />

      </div>
    </form>

  </body>
</html>
```

The code behind file for the web application is as follows:

```
using System;
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;

using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;

using System.Xml.Linq;

//this is the proxy
using localhost;
```

```
namespace wsclient
{
  public partial class _Default : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      if (!IsPostBack)
      {
        lblmessage.Text = "First Loading Time: " +  DateTime.Now.ToLongTimeString
      }
      else
      {
        lblmessage.Text = "PostBack at: " + DateTime.Now.ToLongTimeString();
      }
    }

    protected void btnservice_Click(object sender, EventArgs e)
    {
      StockService proxy = new StockService();
      lblmessage.Text = String.Format("Current SATYAM Price:{0}",
      proxy.GetPrice("SATYAM").ToString());
    }
  }
}
```

Creating the Proxy

A proxy is a stand-in for the web service codes. Before using the web service, a proxy must be created. The proxy is registered with the client application. Then the client application makes the calls to the web service as it were using a local method.

The proxy takes the calls, wraps it in proper format and sends it as a SOAP request to the server. SOAP stands for Simple Object Access Protocol. This protocol is used for exchanging web service data.

When the server returns the SOAP package to the client, the proxy decodes everything and presents it to the client application.

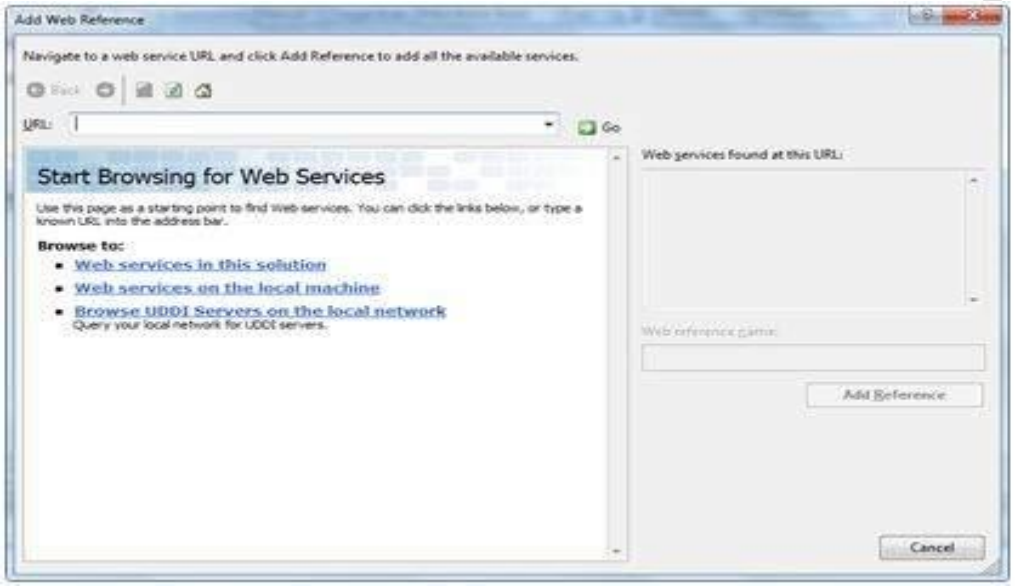
Before calling the web service using the btnservice_Click, a web reference should be added to the application. This creates a proxy class transparently, which is used by the btnservice_Click event.

```
protected void btnservice_Click(object sender, EventArgs e)
```

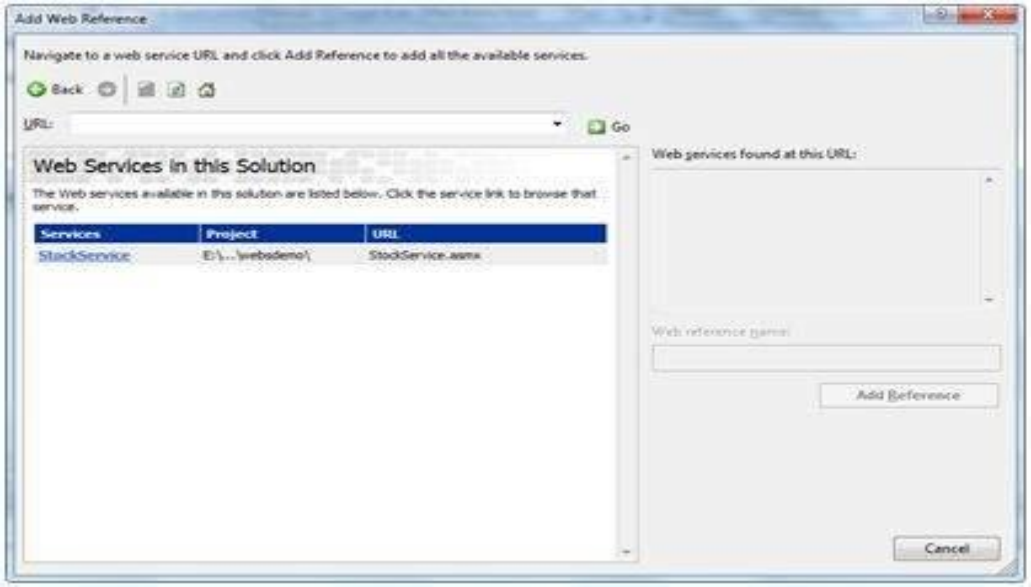
```
{
  StockService proxy = new StockService();
  lblmessage.Text = String.Format("Current SATYAM Price: {0}",
  proxy.GetPrice("SATYAM").ToString());
}
```

Take the following steps for creating the proxy:

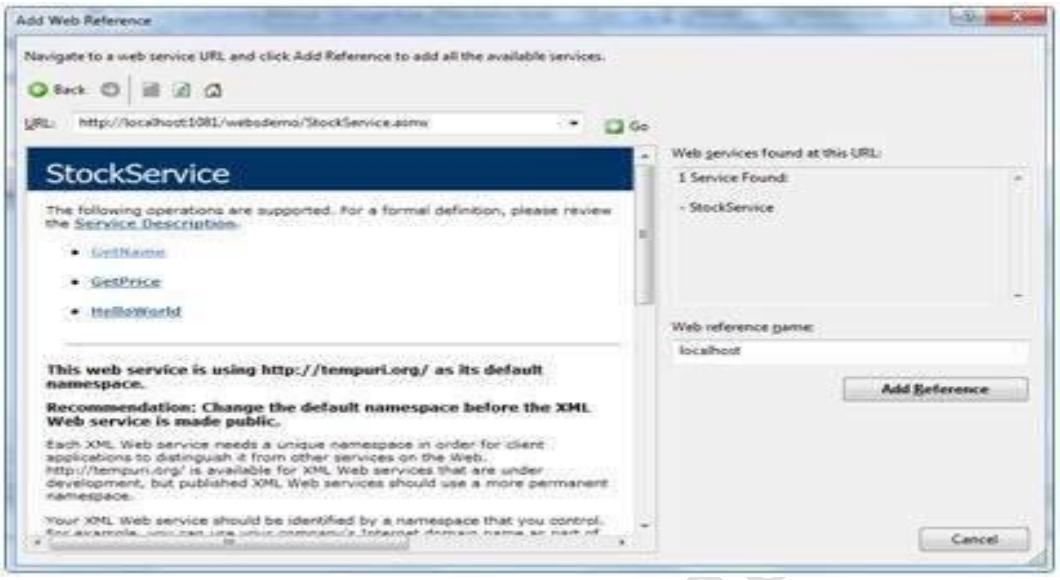
Step (1) : Right click on the web application entry in the Solution Explorer and click on 'Add Web Reference'.



Step (2) : Select 'Web Services in this solution'. It returns the Stock Service reference.



Step (3) : Clicking on the service opens the test web page. By default the proxy created is called 'localhost', you can rename it. Click on 'Add Reference' to add the proxy to the client application.



Include the proxy in the code behind file by adding:

```
using localhost;
```