UNIT - IV

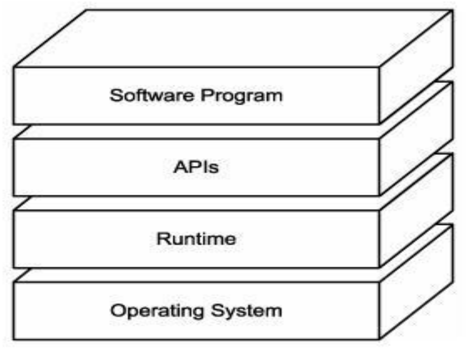
SOA platform basics – SOA support in J2EE – Java API for XML-based web services (JAX-WS) - Java architecture for XML binding (JAXB) – Java API for XML Registries (JAXR) - Java API for XML based RPC (JAX-RPC)- Web Services Interoperability Technologies (WSIT) - SOA support in .NET – Common Language Runtime - ASP.NET web forms – ASP.NET web services – Web Services Enhancements (WSE).

SOA platform basics

Before we begin to look at the specifics of the J2EE and .NET platforms, let's first establish some of the common aspects of the physical development and runtime environments required to build and implement SOA-compliant services.

Basic platform building blocks

Taking a step back from SOA for a moment, let's start by defining the rudimentary building blocks of a software technology platform. The realization of a software program puts forth some basic requirements, mainly:



Fundamental software technology architecture layers.

- 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 (because it provides the environment that will eventually host the software).

- We need APIs that expose features and functions offered by the runtime so that we can build our software program to interact with and take advantage of these features and functions.
- Finally, we need an operating system on which to deploy the runtime, APIs, and the software program. The operating system interfaces with the underlying hardware and likely will provide additional services that can be used by the software program (through the use of additional APIs).

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

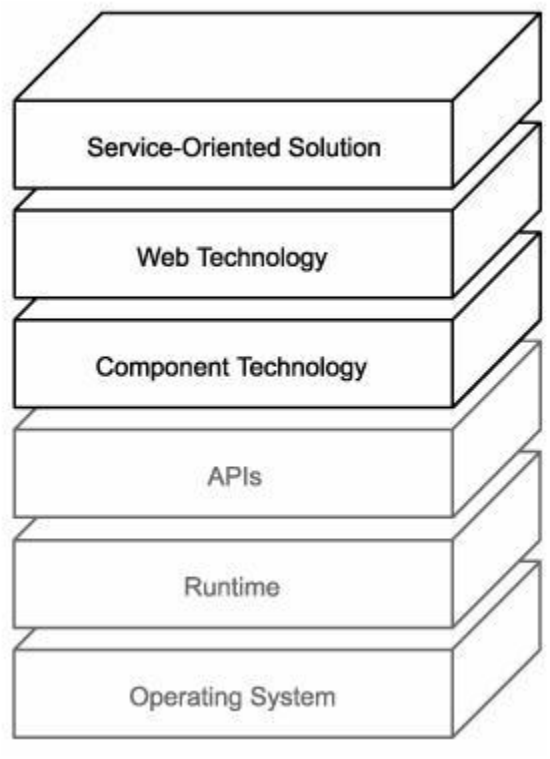
Common SOA platform layers

As we established early on in this book, 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.



Relationship between SOA layers and technologies

When we introduce components and Web services to our architecture model, we end up with a number of different relationships forged between the fundamental architecture layers and the specific technologies introduced by the Web services framework (namely, WSDL, SOAP, UDDI, and the WS-* specifications).

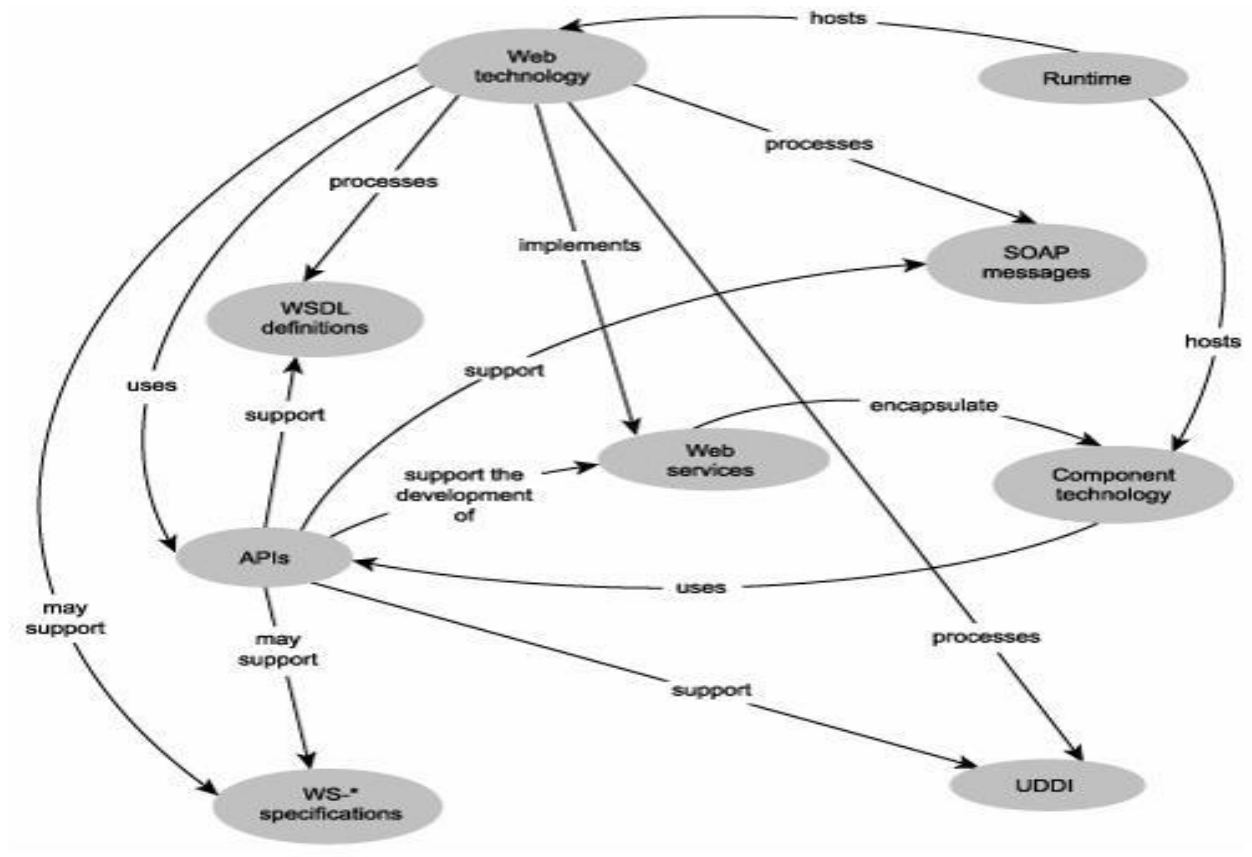
To better understand these dynamics, let's briefly review the requirements for each of the primary relationships.

- 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.

  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.

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



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

As we've established in previous chapters, service providers are commonly expected to performthe 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 providers are designed to facilitate service requestors. A service requestor can be any piece of software capable of communicating with a service provider. 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.

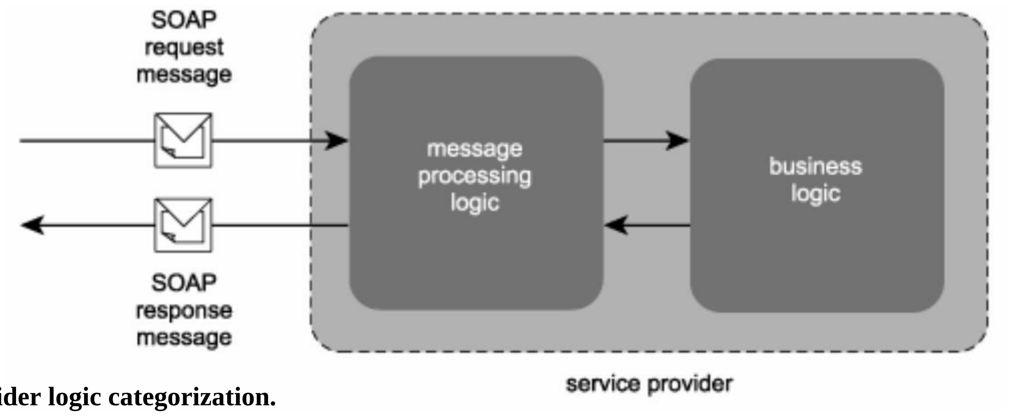
Service processing logic

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.

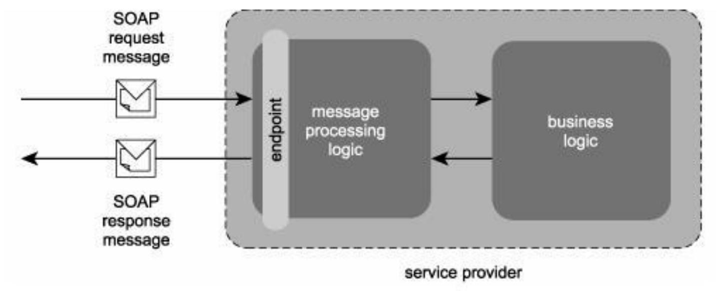


Service provider logic categorization.

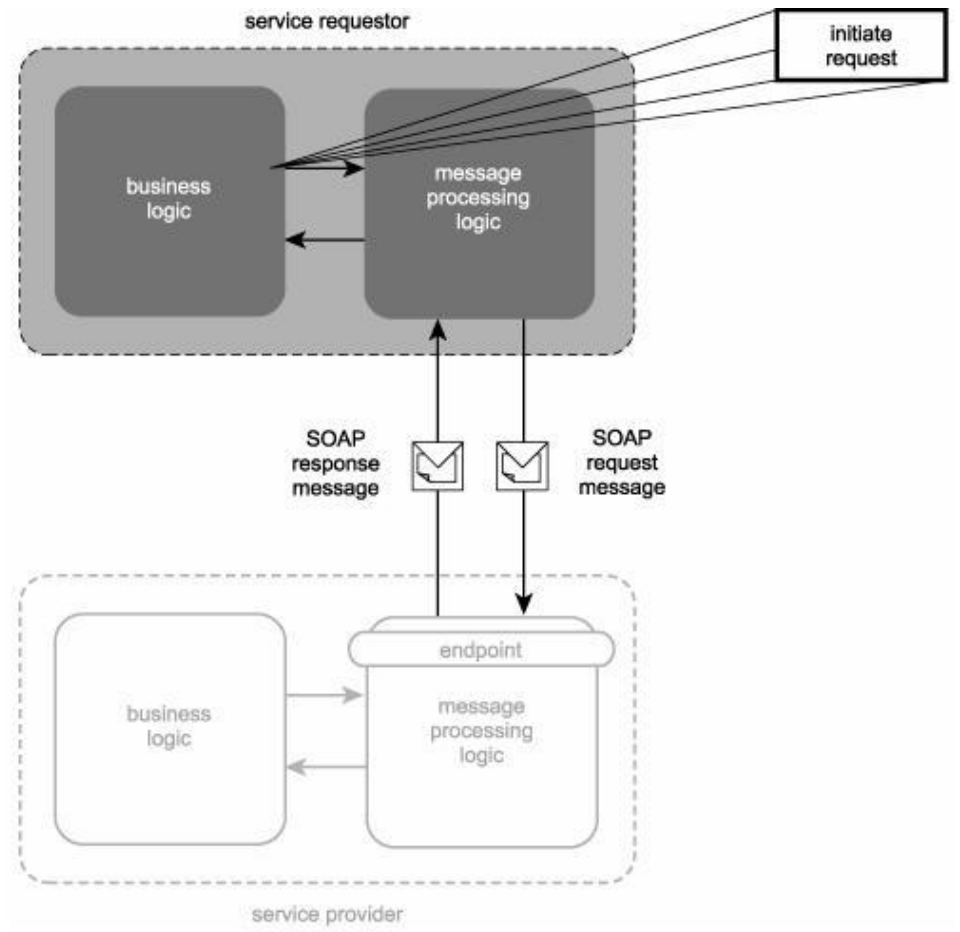
Message Processing Logic	Business Logic
SOAP message receipt and transmission.	Application-specific business processing logic.
SOAP message header processing.	
SOAP message payload validation and parsing.	
SOAP message payload transformation.	

These groups represent logic only, including the messaging logic required to process the WSDL definition. But what about the WSDL itself? This critical piece of a service needs to be distinctly identified, as it relates to and affects a great deal of the surrounding messaging processing logic. To keep things simple we will group the WSDL with other metadata documents (such as policies and schemas) and classify them collectively as the endpoint.

A revised service provider model now including an endpoint within the message processing logic.



A service requestor consisting of message processing and business logic.



Service requestor logic categorizatio

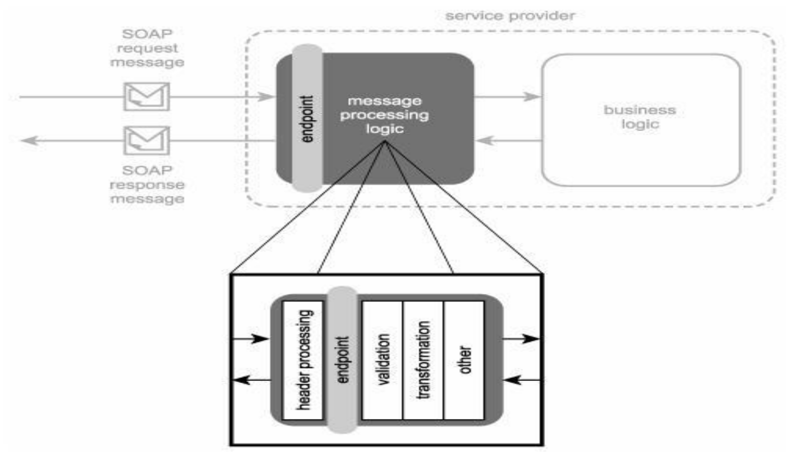
Message Processing Logic	Business Logic
WSDL interpretation (and discovery).	Application-specific business processing logic.
SOAP message transmission and receipt.	
SOAP message header processing.	

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.

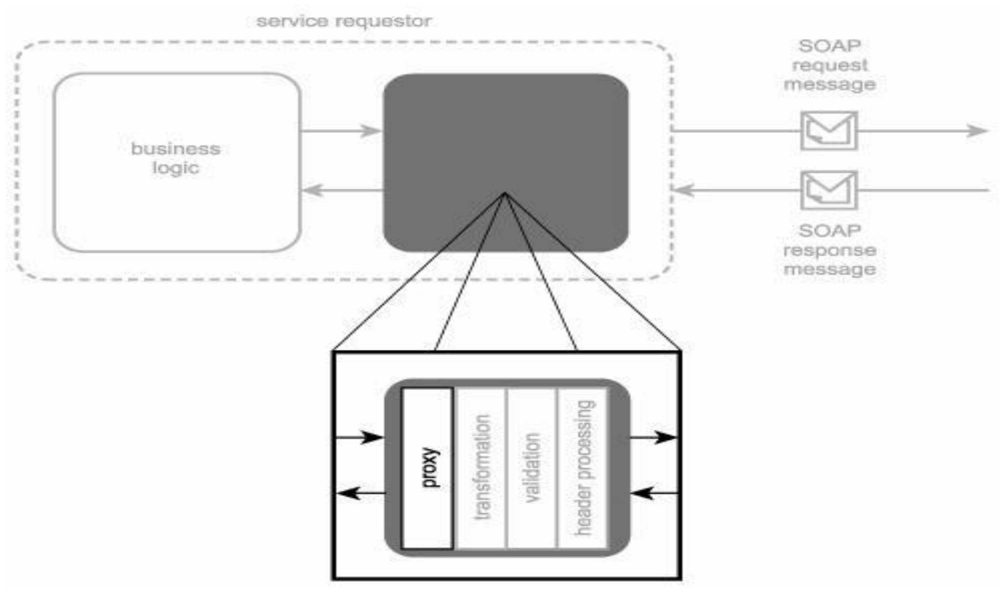
An example of the types of processing functions that can comprise the message processing logic of a service.



Although the message processing logic for service requestors and service providers may be similar, there is an important implementation-level difference. The service provider supplies an endpoint that expresses an interface and associated constraints with which all service requestors must comply.

Vendor platforms accomplish this by supporting the creation of proxy components. These components exist as part of the message processing logic and are commonly auto-generated from the service provider WSDL definition (and associated service description documents). They end up providing a programmatic interface that mirrors the WSDL definition but complies to the native vendor runtime environment.

The message processing logic part of a service requestor includes a proxy component.



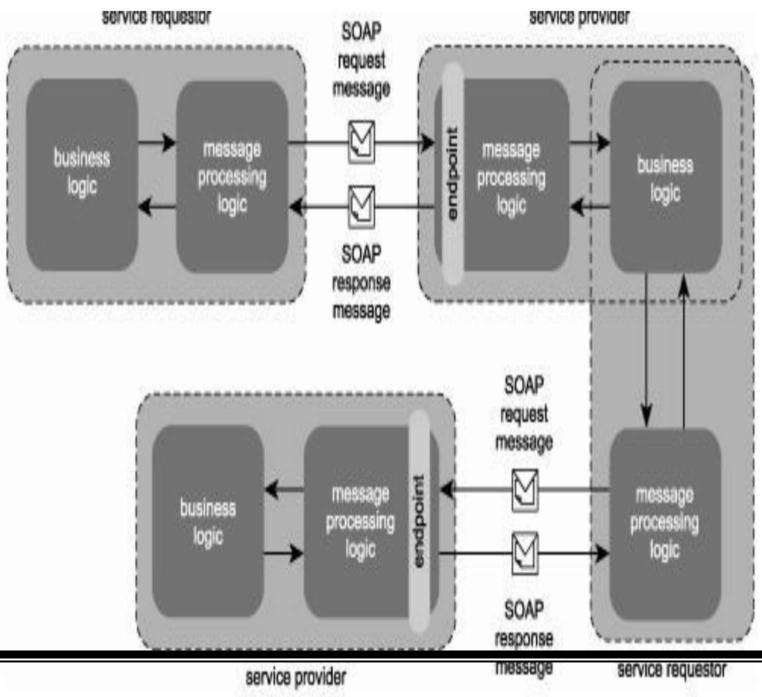
Proxies accept method calls issued from the regular vendor platform components that contain the service requestor business logic. The proxies then use vendor runtime services to translate these method calls and associated parameters into SOAP request messages. When the SOAP request is transmitted, the proxy is further able to receive the corresponding SOAP response from the service provider. It then performs the same type of translation, but in reverse.

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.

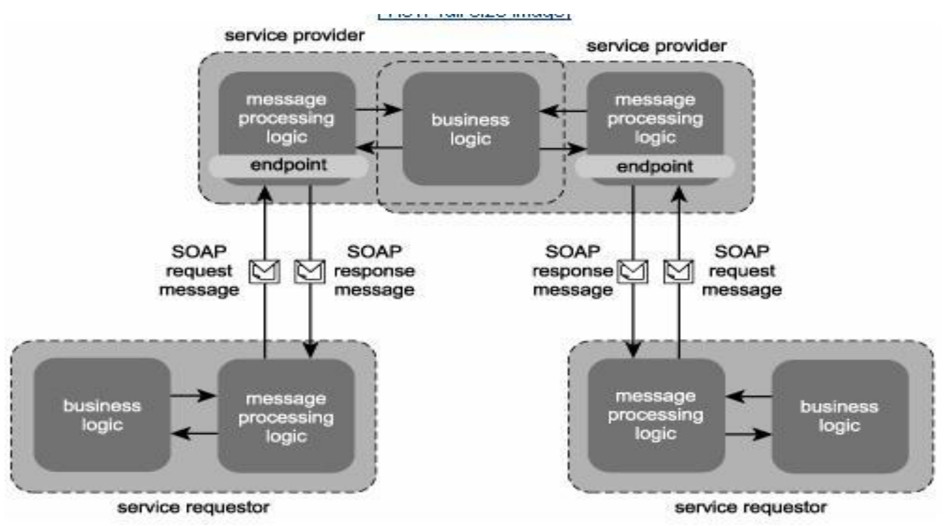
As an independent unit of logic, it is free to act in different roles

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



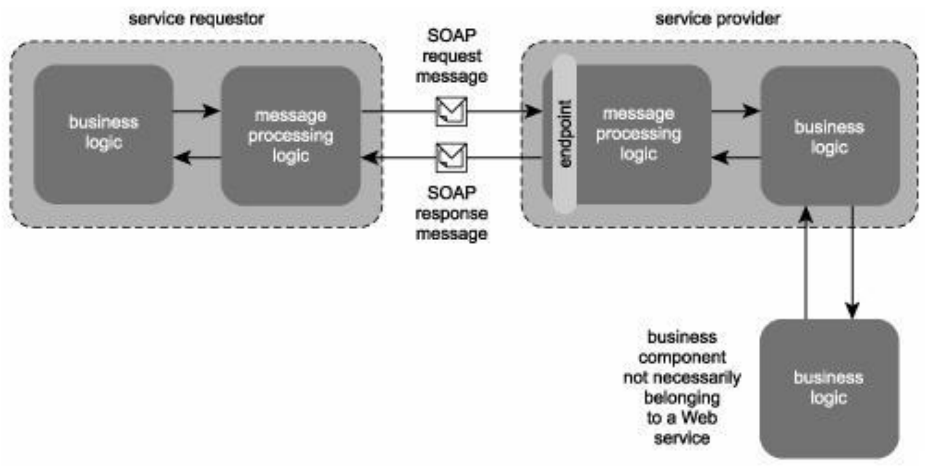
If units of business logic exist as physically separate components, the same business logic can be encapsulated by different service providers

One unit of business logic being encapsulated by two different service providers.



Because units of business logic can exist in their native distributed component format, they also can interact with other components that may not necessarily be part of the SOA. This, in fact, is a very common model in distributed environments where components (as opposed to services) are composed to execute specific tasks on behalf of the service provider.

The same unit of business logic facilitating a service provider and acting on its own by communicating independently with a separate component.



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.

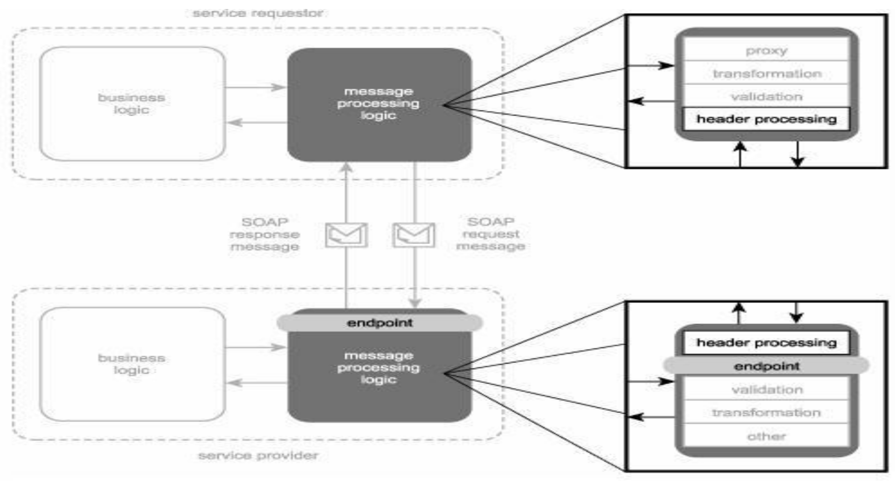
For example, service agents that reside alongside the service requestor will be engaged after a SOAP message request is issued by the service requestor and before it actually is transmitted to the service provider. Similarly, requestor agents generally kick in upon the initial receipt of a SOAP response,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

The same goes for service agents that act on the service provider's behalf. They typically pre-process SOAP request messages and intercept SOAP response messages prior to transmission.

Service agents usually address cross-cutting concerns, providing generic functions to alleviate the processing responsibilities of core Web service 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.

Even though every effort has been made to provide a balanced and equal documentation of each platform, it should be noted that the difference in vendor support required that some of the documentation be approached differently. For example, because J2EE is a platform supported by multiple vendors, multiple vendor products are mentioned. Because .NET is a platform provided by a single vendor, only that vendor's supporting products are referenced.

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. We then proceed to revisit the service-orientation principles and primary primitive and contemporary SOA characteristics established earlier in this book to discuss how these potentially can be realized using the previously explained parts of J2EE.

It is important to note that this section does not provide an in-depth explanation of the J2EE platform, nor do we get into how to program Web services using Java. There are already many comprehensive books that cover this vast subject. 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.

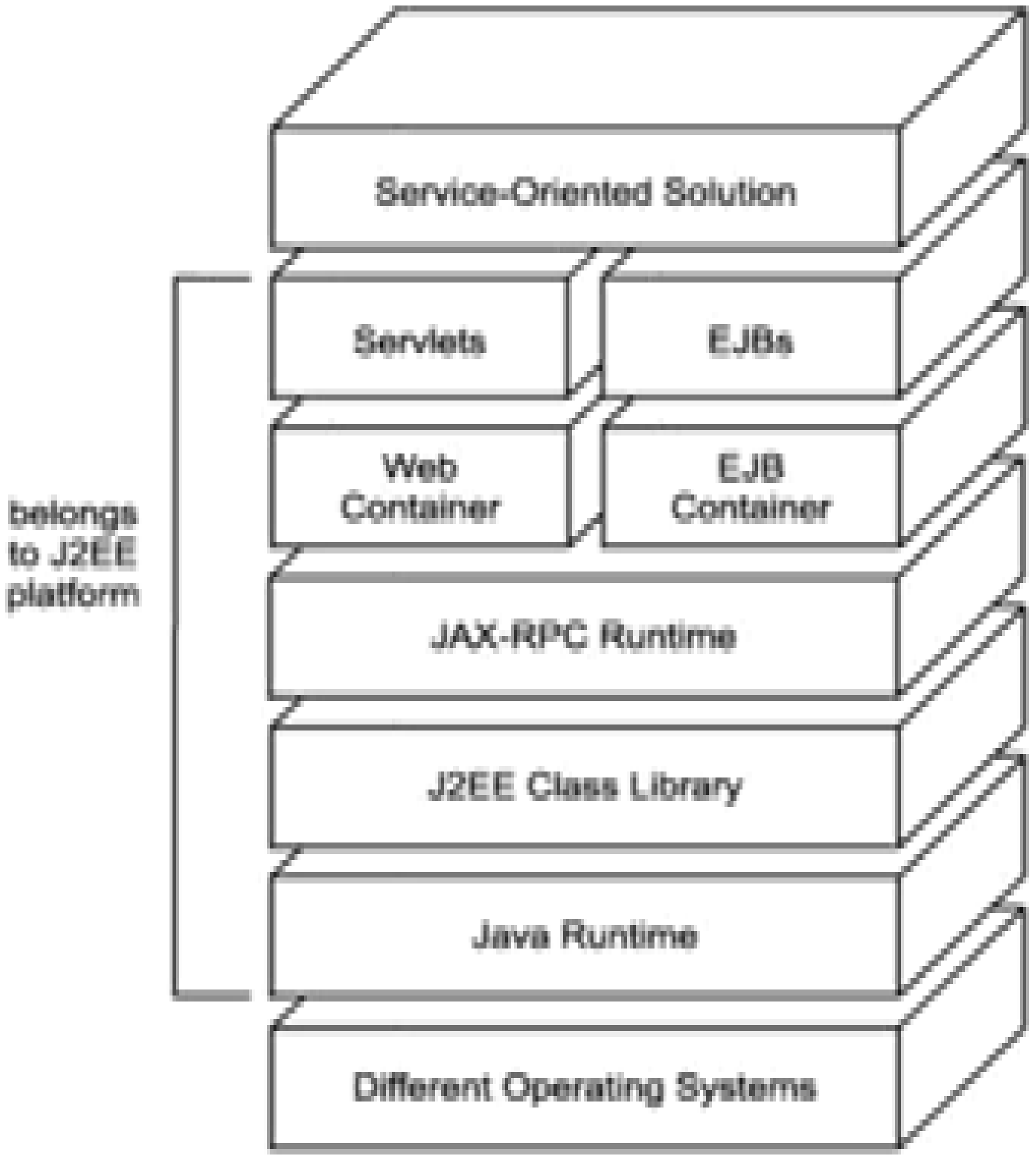
Platform overview

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.

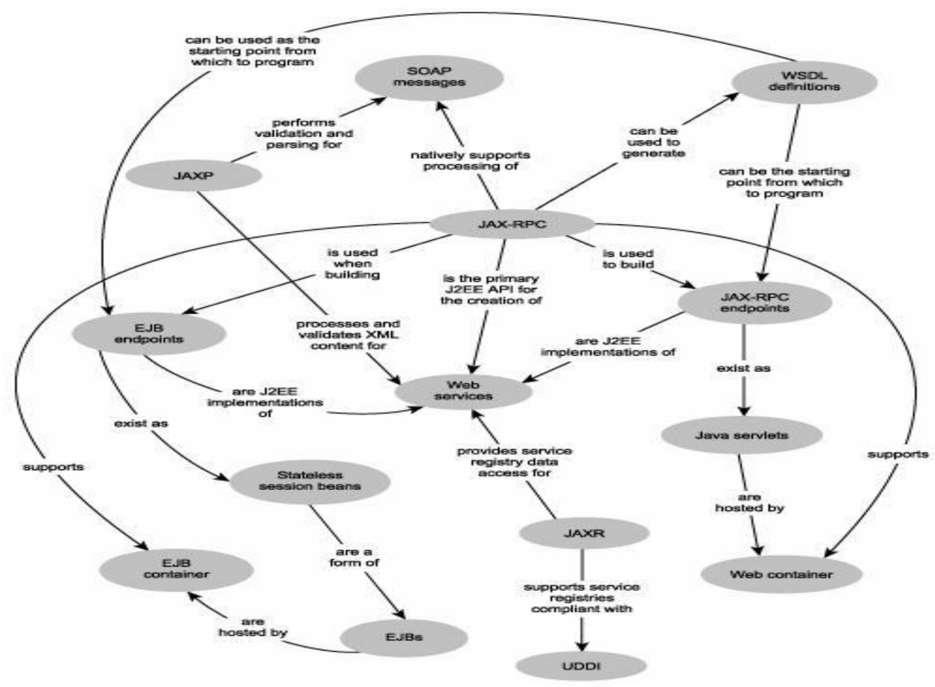
The J2EE development platform consists of numerous composable pieces that can be assembled into full-fledged Web solutions. Let's take a look at some of the technologies more relevant to Web services.

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.



can be used as the starting point from which to program
SOAP messages
WSDL definitions
performs validation and parsing for
JAXP
natively supports processing of
can be used to generate
can be the starting point from which to program
JAX-RPC
is used when building
is the primary J2EE API for the creation of
is used to build
JAX-RPC endpoints
EJB endpoints
processes and validates XML content for
are J2EE implementations of
Web services
are J2EE implementations of
exist as
Java servlets
exist as
supports
provides service registry data access for
Stateless session beans
are hosted by
supports
JAXR
EJB container
are a form of
Web container
supports service registries compliant with
are hosted by
EJBs
UDDI

Before we discuss each of these components individually, let's begin with an overview of some key J2EE specifications. There are many J2EE standards published by Sun Microsystems that establish the parts of the J2EE architecture to which vendors that implement and build products around this environment must conform. 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 geared toward Web services-centric applications that are generally EJB or servlet-based.

- 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.

As explained in the Service providers section, EJB and Web containers can host EJB-based or servlet-based J2EE Web services. Web service execution on both containers is supported by JAX-RPC runtime services. However, it is the vendor-specific container logic that generally determines the shape and form of the system-level message processing logic provided in support of Web services.

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.parsers`A package containing classes for different vendor-specific DOM and SAX parsers.

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

~ `javax.xml.transform`A 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.server`These packages contain a series of core functions for the JAX-RPC API.

~ `javax.xml.rpc.handler` and `javax.xml.rpc.handler.soap`API 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.soap`API 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.registry`A series of registry access functions that support the JAXR API.

~ `javax.xml.registry.infomodel`Classes 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.

Note that we do not discuss these APIs any further. They are mentioned here only to demonstrate the grouping of API functions in J2EE packages.

Service provider

As previously mentioned, 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:

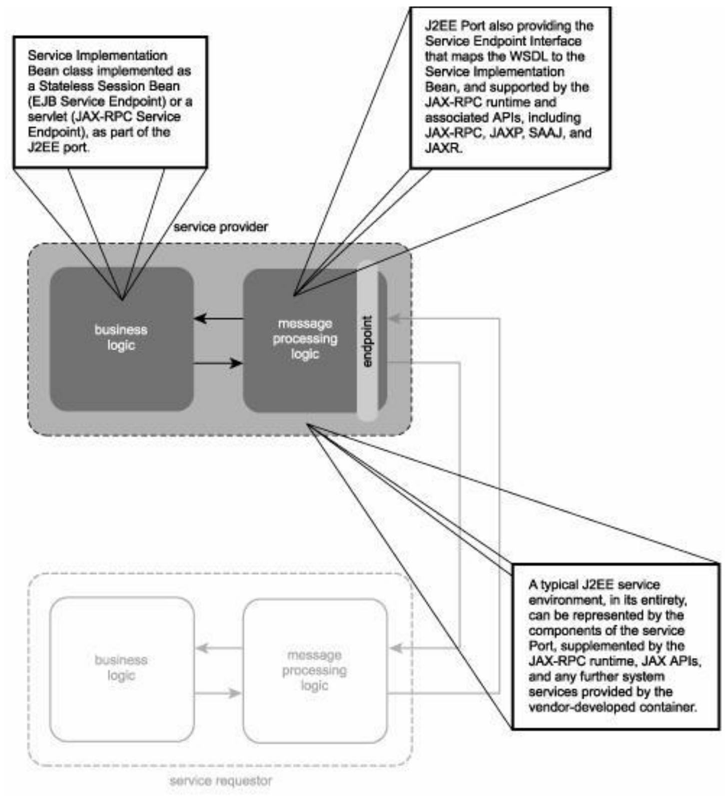
- JAX-RPC Service Endpoint When building Web services for use within a Web container, a JAX-RPC Service Endpoint is developed that frequently is implemented as a servlet by the underlying Web container logic. Servlets are a common incarnation of Web services within J2EE and most suitable for services not requiring the features of the EJB container.
- 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. To build an EJB Service Endpoint requires that the underlying EJB component be a specific type of EJB called a Stateless Session Bean.

Regardless of vendor platform, both types of J2EE Web services are dependent on the JAX-RPC runtime and associated APIs.

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. The Service Implementation Bean can be implemented as an EJB Endpoint (Stateless Session Bean) or a JAX-RPC Endpoint (servlet). For an EJB Endpoint, it is referred to as an EJB Service Implementation Bean and therefore resides in the EJB container. For the JAX-RPC Endpoint, it is called a JAX-RPC Service Implementation Bean and is deployed in the Web container.

A typical J2EE service provider.


Service Implementation Bean class implemented as a Stateless Session Bean (EJB Service Endpoint) or a servlet (JAX-RPC Service Endpoint), as part of the J2EE port.
J2EE Port also providing the Service Endpoint Interface that maps the WSDL to the Service Implementation Bean, and supported by the JAX-RPC runtime and associated APIs, including JAX-RPC, JAXP, SAAJ, and JAXR.
service provider
business logic
message processing logic
endpoint
A typical J2EE service environment, in its entirety, can be represented by the components of the service Port, supplemented by the JAX-RPC runtime, JAX APIs, and any further system services provided by the vendor-developed container.
business logic
message processing logic
service requestor

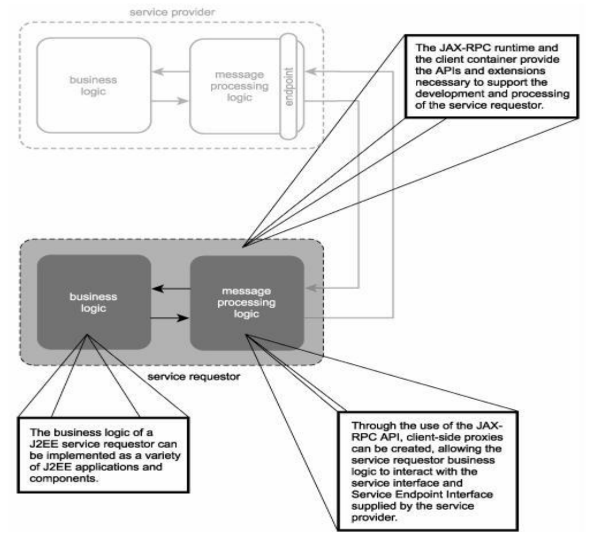
Service requestors

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. Specifically, the compiler creates a Java remote interface for every WSDL `portType` which exposes methods that mirror WSDL operations. It further creates a stub based on the WSDL `port` and `binding` constructs. The result is a proxy component that can be invoked as any other Java component. JAX-RPC takes care of translating communication between the proxy and the requesting business logic component into SOAP messages transmitted to and received from the service provider represented by the WSDL.
- 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.

The latter options are more suited for environments in which service interfaces are more likely to change or for which component interaction needs to be dynamically determined. For example, because a generated stub produces a static proxy interface, it can be rendered useless when the corresponding WSDL definition changes. Dynamic proxy generation avoids this situation.

A typical J2EE service requestor.

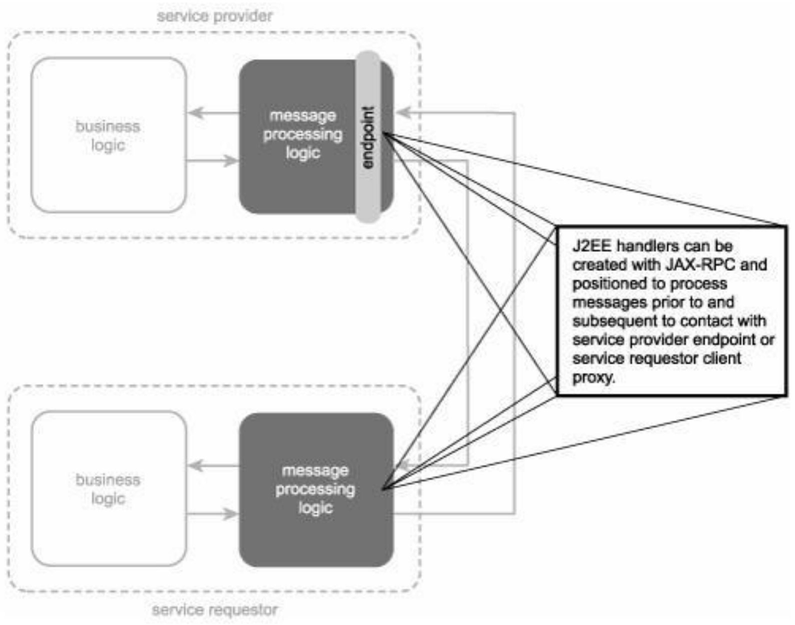

service provider
business logic
message processing logic
endpoint
The JAX-RPC runtime and the client container provide the APIs and extensions necessary to support the development and processing of the service requestor.
business logic
message processing logic
service requestor
The business logic of a J2EE service requestor can be implemented as a variety of J2EE applications and components.
Through the use of the JAX-RPC API, client-side proxies can be created, allowing the service requestor business logic to interact with the service interface and Service Endpoint Interface supplied by the service provider.

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.


service provider
business logic
message processing logic
endpoint
J2EE handlers can be created with JAX-RPC and positioned to process messages prior to and subsequent to contact with service provider endpoint or service requestor client proxy.
business logic
message processing logic
service requestor

Multiple handlers can be used to process different header blocks in the same SOAP message. In this case the handlers are chained in a predetermined sequence (appropriately called a handler chain).

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. The technologies supported by these toolkits, when sufficiently mature, can further support contemporary SOA. 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

The distributed nature of the J2EE platform allows for the creation of independent units of processing logic through Enterprise Java Beans or servlets. EJBs or servlets can contain small or large amounts of application logic and can be composed so that individual units comprise the processing requirements of a specific business task or an entire solution.

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. EJB and JAX-RPC Endpoints further establish a standard WSDL definition, supported by J2EE HTTP and SOAP runtime services. Therefore, loose coupling is a characteristic that can be achieved in support of SOA.

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. Within SOA this extension can be used by the business logic of a Web service, allowing SOAP messages to be passed through from the message process logic (which generally will rely on HTTP as the transport protocol). Either way, the J2EE platform provides the required messaging support for primitive SOA.

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.

It may therefore be easier for JAX-RPC Service Endpoints to retain complete autonomy, especially when they are only required to execute a small amount of business logic. EJB Service Endpoints are required to exist as Stateless Session Beans, which supports autonomy within the immediate endpoint logic. However, because EJB Service Endpoints are more likely to represent existing legacy logic (or a combination of new and legacy EJB components), retaining a high level of autonomy can be challenging.

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. Because service-orientation encourages services to be reusable and because a service can encapsulate one or more new or existing EJB components, reusability on a service level comes down to the design of a service's business logic and endpoint.

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.

As previously mentioned, one of the requirements for adapting an EJB component into an EJB Service Endpoint is that it be completely stateless. In the J2EE world, this means that it must be designed as a Stateless Session Bean, a type of EJB that does not manage state but that may still defer state management to other types of EJB components (such as Stateful Session Beans or Entity Beans).

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.
Note that even if JAXR is used to represent a UDDI registry, it does so by exposing an interface that differs from the standard UDDI API. (For example, a UDDI `Business-Entity` is a JAXR `Organization`, and a UDDI `BusinessService` is a JAXR `Service`.)

Contemporary SOA support

Extending an SOA beyond the primitive boundary requires a combination of design and available technology in support of the design. Because 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 Web services subset of the J2EE platform supports industry standard Web services specifications, including WSDL, SOAP, and UDDI. As explained later in the Intrinsically interoperable section, support for the WS-I Basic Profile also has been provided. Further, 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.

Further, by designing services to be WS-I Basic Profile compliant, vendor diversity beyond J2EE platforms is supported. For example, an organization that has built an SOA based on J2EE technology may choose to build another using the .NET framework. Both environments can interoperate if their respective services conform to the same open standards. This also represents vendor diversity.

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.

As of version 1.1, the JAX-RPC API is fully capable of creating WS-I Basic Profile-compliant Web services. This furthers the vision of producing services that are intrinsically interoperable.

Care must be taken, though, to prevent the use of handlers from performing runtime processing actions that could jeopardize this compliance.

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.

Also worth taking into consideration is the J2EE Connector Architecture (JCA), a structured, adapter-centric integration architecture through which resource adapters are used to bridge gaps between J2EE platforms and other environments. As with JMS, JCA is traditionally centered around the use of proprietary messaging protocols and platform-specific adapters. Recently, however, support for asynchronous and SOAP messaging has been introduced. Further, service adapters have been made available to tie JCA environments into service-oriented solutions.

Numerous integration server platforms also are available to support and implement the overall concept of enterprise-wide federation. Depending on the nature of the integration architecture, service-oriented integration environments are built around orchestration servers or enterprise service bus offerings (or both).

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.

With regards to taking advantage of the composable contemporary SOA landscape, the J2EE platform, in its current incarnation, does not yet provide native support for WS-* specifications. Instead, extensions are supplied by product vendors that implement and build upon J2EE standards. The extent to which the WS-* features of an SOA based on the J2EE platform can be composed is therefore currently dependent upon the vendor-specific platform used.

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.

Because J2EE environments are implemented by different vendors, extensibility can sometimes lead to the use of proprietary extensions. While still achieving extensibility within the vendor environment, this can limit the portability and openness of Java solutions.

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. Instead, orchestration services and design tools are provided by vendors to supplement the J2EE Web services development and runtime environment. Service-oriented business modeling and the service layers we've discussed in this book can therefore be created with the right vendor tools.

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.

Depending on the vendor server platform used, some limitations may be encountered when building service compositions that require message-level security measures. These limitations may inhibit the extent of feasible logic-level abstraction.

Organizational agility and enterprise-wide loose coupling

In past chapters we explored the enablement of enterprise-wide agility through the implementation of abstraction via service sub-layers. As discussed in the previous section, 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.

For example, if a vendor server platform is not satisfying current business needs and requires replacement, application, entity-centric, and task-centric services likely will be sufficiently mobile so that they can be used in the replacement environment. The orchestration logic may or may not be portable, depending on whether a common orchestration language, such as WS-BPEL, was used to express the process logic.

To attain a state where business and technology domains of an enterprise are loosely coupled and achieve full, two-way agility, requires the fulfillment of a number of contemporary SOA characteristics identified in this book. The J2EE platform provides a foundation upon which to build a standardized and extensible SOA. Enterprise features offered by vendor platforms need to be incorporated to add layers on top of this foundation necessary to driving service-orientation across the enterprise.

SOA support in .NET

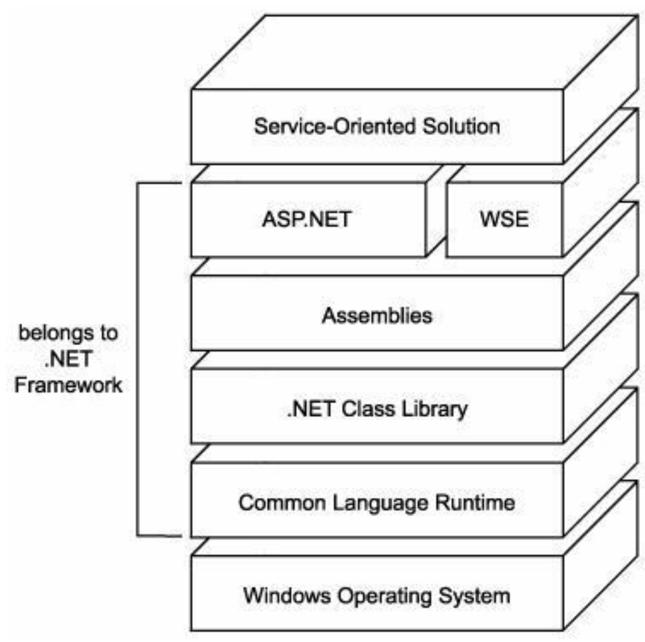
The .NET framework is the second of the two platforms for which we discuss SOA support in this book. As with the previous section, we first introduce the primary parts of the .NET platform and then delve into our familiar primitive SOA characteristics, service-orientation principles, and contemporary SOA characteristics. For each we explore .NET features that provide direct or indirect support.

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-relatea

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 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. An assembly can contain multiple classes that further partition code using object-oriented principles. The application logic behind a .NET Web service is typically contained within an assembly (but does not need to be).

ASP.NET 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.

Various supplementary runtime layers can be added to the CLR. ASP.NET itself provides a set of runtime services that establish the HTTP Pipeline, an environment comprised of system service agents that include HTTP modules and HTTP handlers (see the Service agents section for more information). Also worth noting is that the established COM+ runtime provides a further set of services (including object pooling, transactions, queued components, and just-in-time activation) that are made available to .NET applications.

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:

~ The `XmlReader` and `XmlWriter` classes that provide functionality for retrieving and generating XML document content.

~ Fine-grained classes that represent specific parts of XML documents, such as the `XmlNode`, `XmlElement`, and `XmlAttribute` classes.

- 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:

~ WSDL documents are represented by a series of classes that fall under the `System.Web.Services.Description` namespace.

~ Communication protocol-related functionality (including SOAP message documents) are expressed through a number of classes as part of the `System.Web.Services.Protocols` namespace.

~ The parent `System.Web.Services` class that establishes the root namespace also represents a set of classes that express the primary parts of ASP.NET Web service objects (most notably, the `System.Web.Services.WebService` class).

~ Also worth noting is the `SoapHeader` class provided by the `System.Web.Services.Protocols` namespace, which allows for the processing of standard SOAP header blocks.

In support of Web services and related XML document processing, a number of additional namespaces provide class families, including:

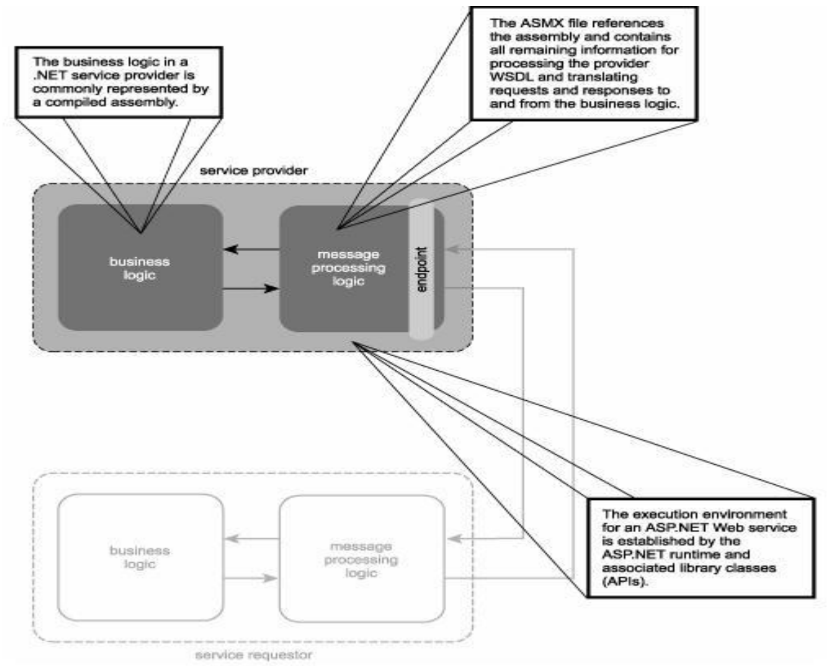
- `System.Xml.Xsl` Supplies documentation transformation functions via classes that expose XSLT-compliant features.
- `System.Xml.Schema` A set of classes that represent XML Schema Definition Language (XSD)-compliant features.
- `System.Web.Services.Discovery` Allows for the programmatic discovery of Web service metadata.

Note that we do not discuss these class libraries any further. They are only provided here to demonstrate the organization of .NET APIs into the .NET class hierarchy.

Service providers

.NET service providers are Web services that exist as a special variation of ASP.NET applications, called ASP.NET Web Services. You can recognize a URL pointing to an ASP.NET Web Service by the ".asmx" extension used to identify the part of the service that acts as the endpoint. ASP.NET Web Services can exist solely of an ASMX file containing inline code and special directives, but they are more commonly comprised of an ASMX endpoint and a compiled assembly separately housing the business logic.

A typical .NET service provider.


The business logic in a .NET service provider is commonly represented by a compiled assembly.
The ASMX file references the assembly and contains all remaining information for processing the provider WSDL and translating requests and responses to and from the business logic.
service provider
business logic
message processing logic
endpoint
The execution environment for an ASP.NET Web service is established by the ASP.NET runtime and associated library classes (APIs).
business logic
message processing logic
service requestor

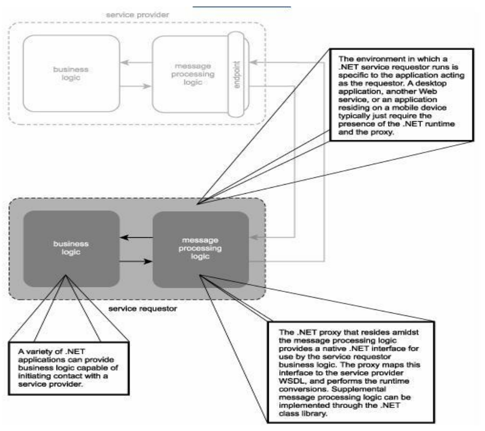
Service requestors

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.

The .NET proxy translates method calls into HTTP requests and subsequently converts the response messages issued by the service provider back into native method return calls.

The code behind a proxy class is auto-generated using Visual Studio or the WSDL.exe command line utility. Either option derives the class interface from the service provider WSDL definition and then compiles the proxy class into a DLL.

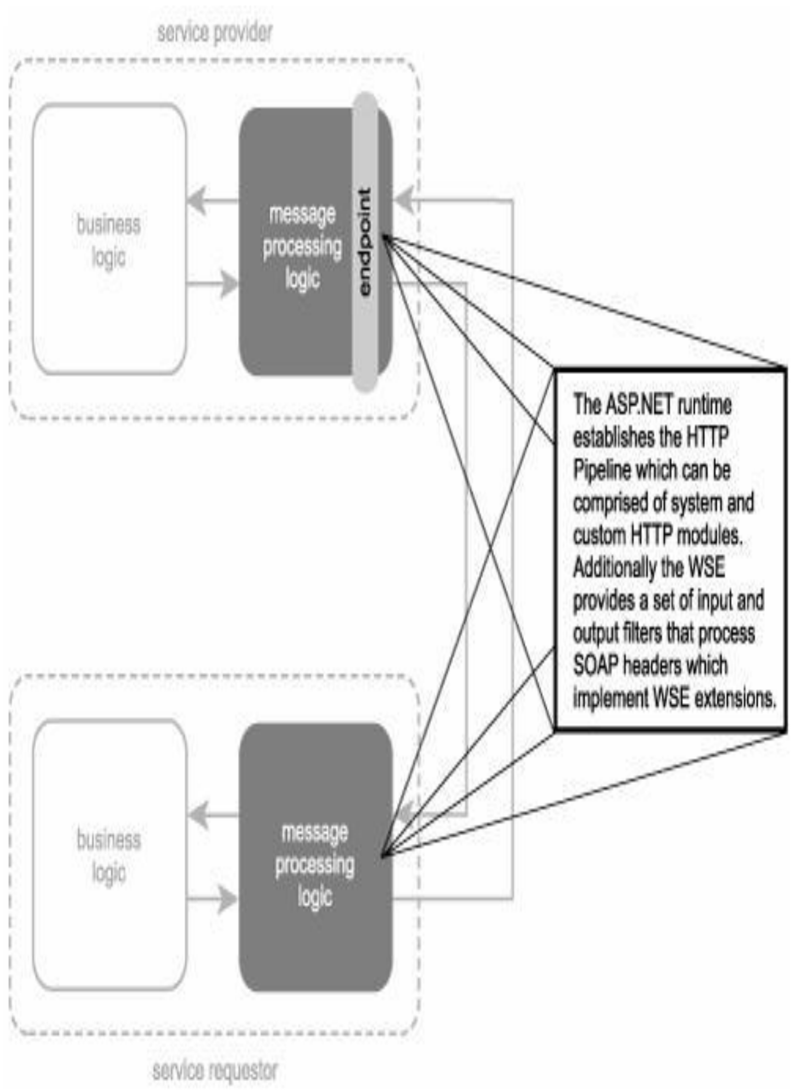
A typical .NET service requestor.


service provider
business logic
message processing logic
endpoint
The environment in which a .NET service requestor runs is specific to the application acting as the requestor. A desktop application, another Web service, or an application residing on a mobile device typically just require the presence of the .NET runtime and the proxy.
business logic
message processing logic
service requestor
A variety of .NET applications can provide business logic capable of initiating contact with a service provider.
The .NET proxy that resides amidst the message processing logic provides a native .NET interface for use by the service requestor business logic. The proxy maps this interface to the service provider WSDL, and performs the runtime conversions. Supplemental message processing logic can be implemented through the .NET class library.

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.

Types of .NET service agents.


service provider
business logic
message processing logic
endpoint
The ASP.NET runtime establishes the HTTP Pipeline which can be comprised of system and custom HTTP modules. Additionally the WSE provides a set of input and output filters that process SOAP headers which implement WSE extensions.
business logic
message processing logic
service requestor

Also worth noting are HTTP Handlers, which primarily are responsible for acting as runtime endpoints that provide request processing according to message type. As with HTTP Modules, HTTP Handlers can also be customized. (Other parts of the HTTP Pipeline not discussed here include the HTTP Context, HTTP Runtime, and HTTP Application components.)

Another example of service agents used to process SOAP headers are the filter agents provided by the WSE toolkit (officially called WSE filters). The feature set of the WSE is explained in the next section (Platform extensions), but let's first briefly discuss how these extensions exist as service agents.

WSE provides a number of extensions that perform runtime processing on SOAP headers. WSE therefore can be implemented through input and output filters that are responsible for reading and writing SOAP headers in conjunction with ASP.NET Web proxies and Web services. Much like the pre-processing scenarios we established in the Service agents sub-section of the SOA platforms section at the beginning of this chapter, WSE filters position themselves to intercept SOAP messages after submission on the service provider's end and prior to receipt on the service requestor's side.

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. Through the introduction of Web services support, .NET assemblies can be composed and encapsulated through ASP.NET Web Services. Therefore, the creation of independent services via .NET supports the service encapsulation required by primitive SOA.

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.

Further, the use of ASP.NET Web Services establishes service interfaces represented as WSDL descriptions, supported by a SOAP messaging framework. This provides the foremost option for achieving loose coupling in support of SOA.

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.

MSMQ messaging offers a proprietary alternative to the native SOAP messaging capabilities provided by the .NET framework. SOAP, however, is the primary messaging format used within contemporary .NET SOAs, as much of the ASP.NET environment and supporting .NET class libraries are centered around SOAP message communication and processing.

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.

However, building autonomous ASP.NET Web Services is achieved more easily when creating a new service-oriented solution, as the supporting application logic can be designed to support autonomy requirements. Further, self-contained ASP.NET Web Services that do not share processing logic with other assemblies are naturally autonomous, as they are in complete control of their logic and immediate runtime environments.

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.

- A UDDI Services extension is offered on newer releases of the Windows Server product, allowing for the creation of private registries.

- Also worth noting is that Visual Studio contains built-in UDDI support used primarily when adding services to development projects.

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.

As mentioned earlier, the WSE extension to .NET provides additional support for a distinct set of WS-* specifications. Finally, as described later in the Intrinsically interoperable section, version 2.0 of the .NET framework and Visual Studio 2005 provide native support for the WS-I Basic Profile.

Also worth noting is that Microsoft itself has provided significant contributions to the development of several key open 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).

The .NET framework provides limited vendor diversity with regard to its development or implementation. This is because it is a proprietary technology that belongs to a single vendor (Microsoft). However, a third-party marketplace exists, providing numerous add-on products.

Additionally, several server product vendors support the deployment and hosting of .NET Web Services and assemblies.

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. (Previous versions of Visual Studio can be used to develop Basic Profile compliant Web services, but they require the use of third-party testing tools to ensure compliance.) Additional design efforts to increase generic interoperability also can be implemented using standard .NET first-generation Web services features.

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. It supplies a series of native adapters and is further supplemented by a third-party adapter marketplace. BizTalk also provides an orchestration engine with import and export support for BPEL process definitions.

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.

With regard to providing support for composable Web specifications, the WSE supplies its own associated class library, allowing only those parts required of the WSE (and corresponding WS-* specifications) to be pulled into service-oriented solutions.

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.

Functional extensibility also can be achieved by extending .NET SOAs through compliant platform products, such as the aforementioned BizTalk server.

Supports service-oriented business modeling

Service-oriented business modeling concepts can be implemented with .NET by creating the standard application, entity-centric, and task-centric service layers described earlier in this book. The orchestration layer requires the use of an orchestration engine, capable of executing the process definition that centralizes business workflow logic. Orchestration features are not a native part of the .NET framework. However, they can be implemented by extending a .NET solution environment with the BizTalk server platform.

Logic-level abstraction

.NET SOAs can position ASP.NET Web Services and service layers to abstract logic on different levels. Legacy and net-new application logic can be encapsulated and wholly abstracted through proper service interface design. Service compositions can be built to an extent through the use of custom SOAP headers and correlation identifiers or by taking advantage of WSE extensions, such as the support provided for WS-Addressing and WS-Referral. (WSE also provides fundamental support for message-level security through extensions that implement portions of the WS-Security framework.)

Organizational agility and enterprise-wide loose coupling

Because the .NET framework supports the development of industry-standard Web services, the proper positioning and application of service layers allows for the creation of the required layers of abstraction that promote fundamental agility. The use of an orchestration layer can further increase corporate responsiveness by alleviating services from business process-specific logic and reducing the need for task-centric services.

The ultimate state of a service-oriented enterprise is to attain a level of bi-directional agility between business and technology domains. This can be achieved through contemporary SOA and requires that many of the discussed characteristics be fully realized. The .NET framework provides a foundation for SOA, capable of fully manifesting the primitive SOA characteristics and select qualities associated with contemporary SOA. Microsoft extensions to the .NET framework and third-party products are required to implement enterprise-wide loose coupling and realize the benefits associated with organizational agility.