UNIT - IV

4.1. SOA platform basics

Before looking the specifics of the J2EE and .NET platforms, first discuss about some of the common aspects of the physical development and runtime environments required to build and implement SOA-compliant services.

4.1.1. Basic platform building blocks

The building blocks of a software technology platform which has the realization of a software program that puts forth some basic requirements, mainly:

- We need a development environment with which to program and assemble the software program and it 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 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.

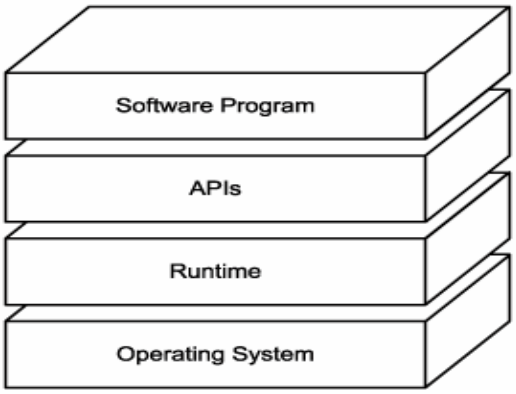


Fig 4.1. Fundamental software technology architecture layers.

4.1.2. Common SOA platform layers

The 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.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.

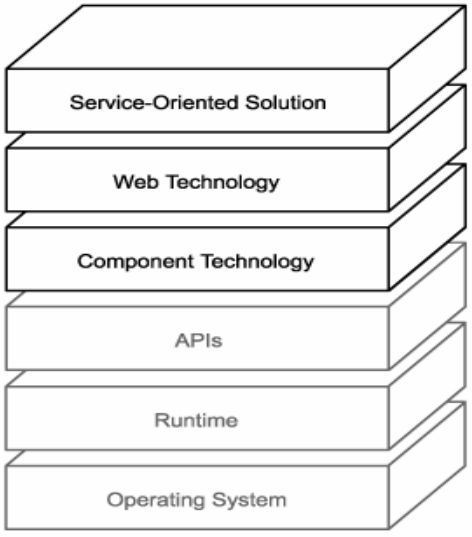


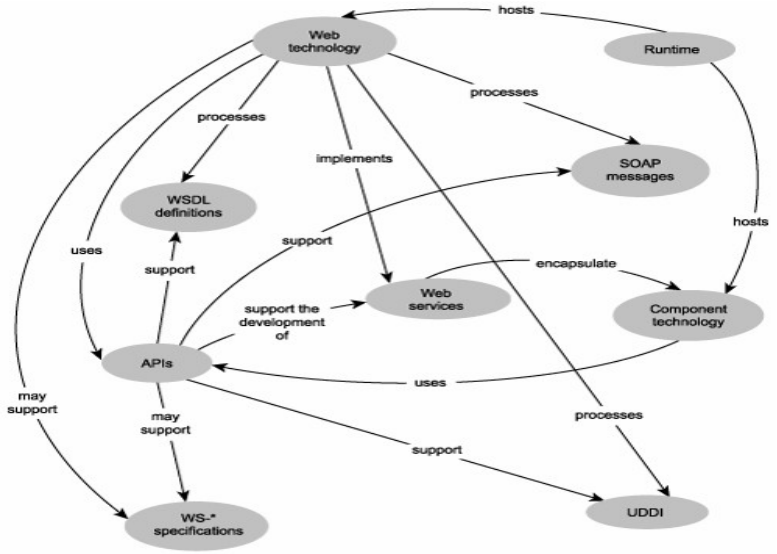
Fig 4.2. The common layers required by a development and runtime platform for building SOA.

- We need the ability to encapsulate and expose application logic through industry standard Web services technologies.

4.1.3. 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.

Fig 4.3. A logical view of the basic relationships between the core parts of a service-oriented architecture.



technologies introduced by the Web services framework (namely, WSDL, SOAP, UDDI, and the WS-* specifications).

4.1.4. Fundamental service technology architecture

The 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.

4.1.4.1. 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 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.

4.1.4.2. 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.

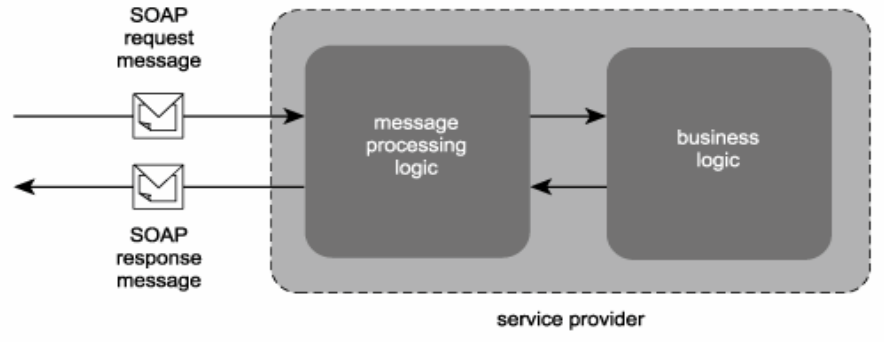
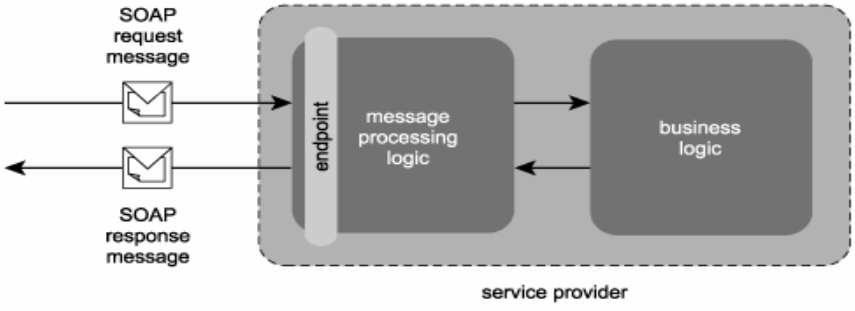


Fig 4.4. A service provider consisting of message processing and business logic.

Table 4.1. 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.	

Fig 4.5. A revised service provider model now including an endpoint within the message processing logic.



The primary difference between how service logic is used in requestors and providers is related to the role of business logic. The business logic part of a service requestor is responsible for initiating an

activity (and the resulting SOAP message exchange), whereas the business logic within a service provider responds to an already initiated activity.

Fig 4.6. A service requestor consisting of message processing and business logic.

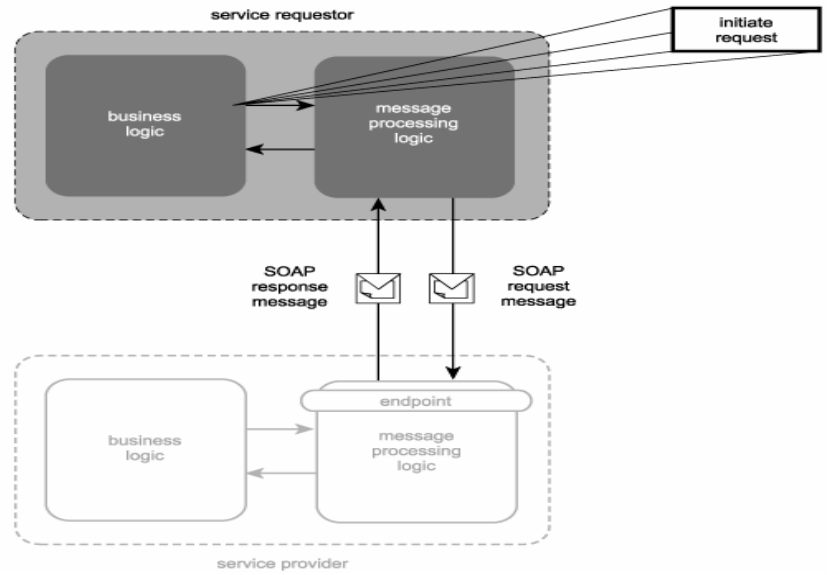


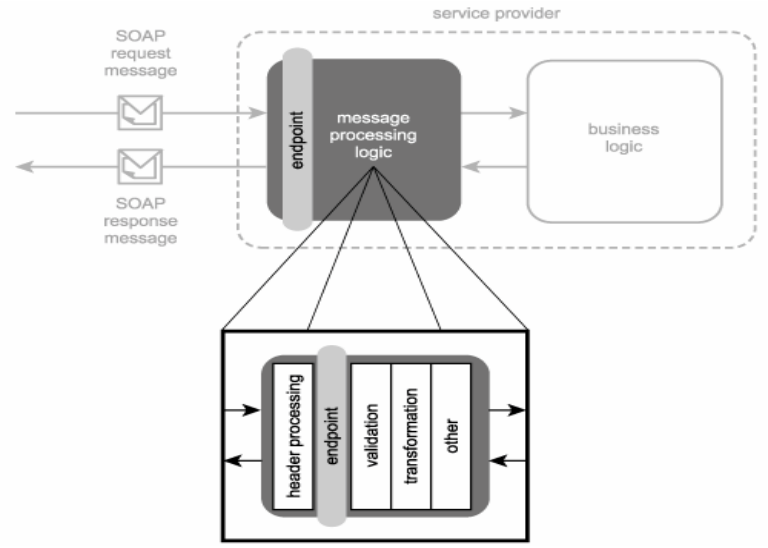
Table 4.2. Service requestor logic categorization.

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.	

4.1.4.3. Message processing logic

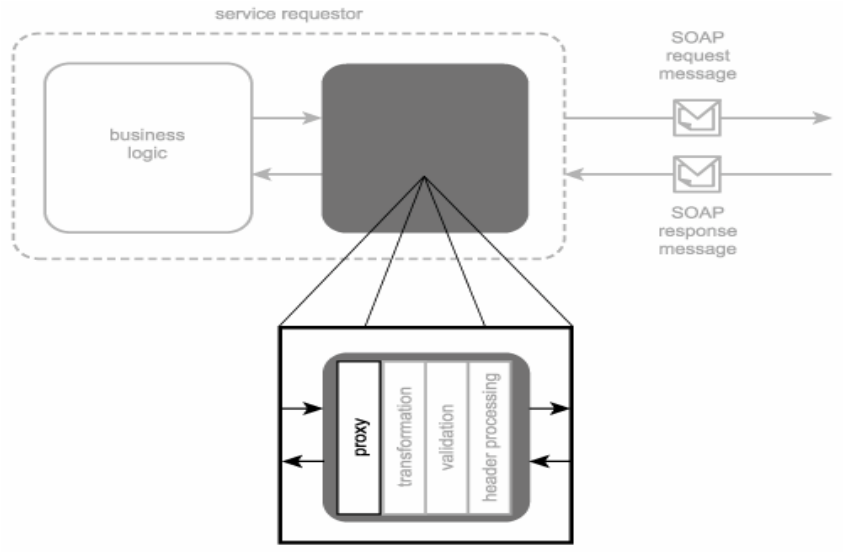
The characteristics of the message processing logic of a service provider and service requestor consists of functions or tasks performed by a combination of runtime services and application-specific extensions. It is therefore not easy to nail down which elements of the message processing logic belong exclusively to the service. Among the processing layers are tasks, such as header processing, that are generic and applied to all service providers. Validation or transformation tasks, on the other hand, may involve service-specific XSD schemas and XSLT stylesheets and therefore may be considered exclusive to the service provider.

Fig 4.7. 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.

Fig 4.8. The message processing logic part of a service requestor includes a proxy component.

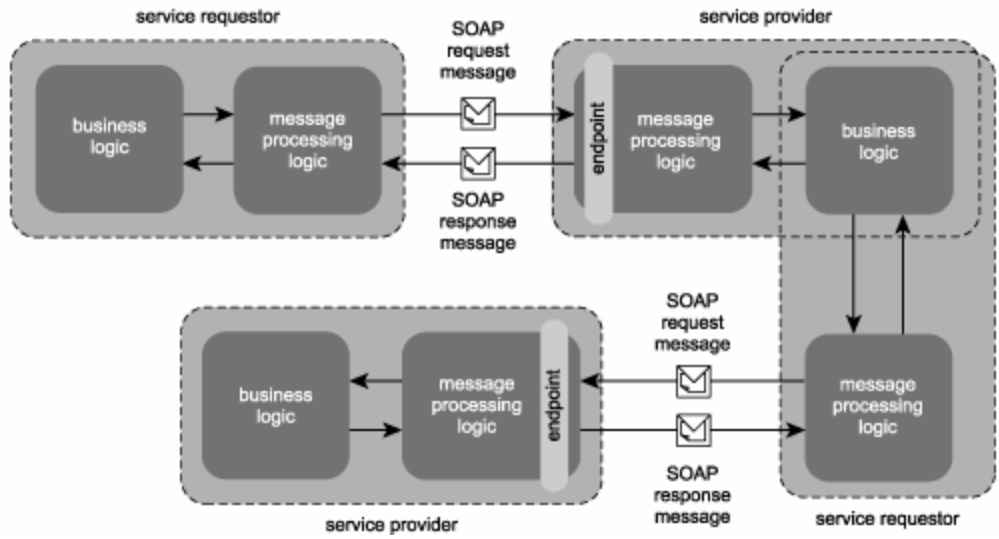


Proxies accept method calls issued from the regular vendor platform components that contain the service requestor business logic. The proxies then use vendor runtime services to translate these method calls and associated parameters into SOAP request messages. When the SOAP request is transmitted, the 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.

4.1.4.4. Business logic

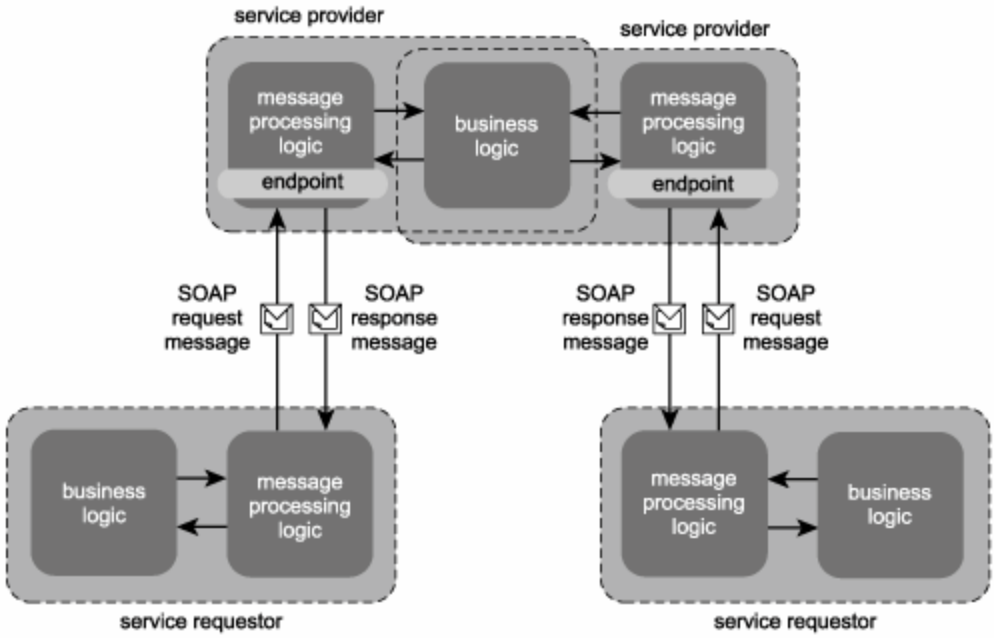
Business logic can exist as a standalone component, housing the intelligence required to either

Fig 4.9. The same unit of business logic participating within a service provider and a service requestor.



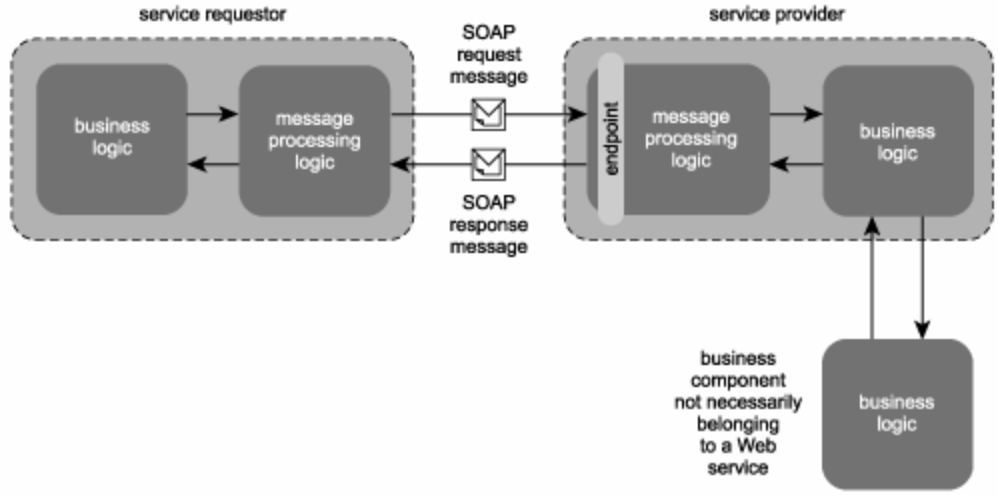
invoke a service provider as part of a business activity or to respond to a request in order to participate in such an activity. As an independent unit of logic, it is free to act in different roles. If units of business logic exist as physically separate components, the same business logic can be encapsulated by different service providers.

Fig 4.10. 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.

Fig 4.11. The same unit of business logic facilitating a service provider and acting on its own by communicating independently with a separate component.



In this case

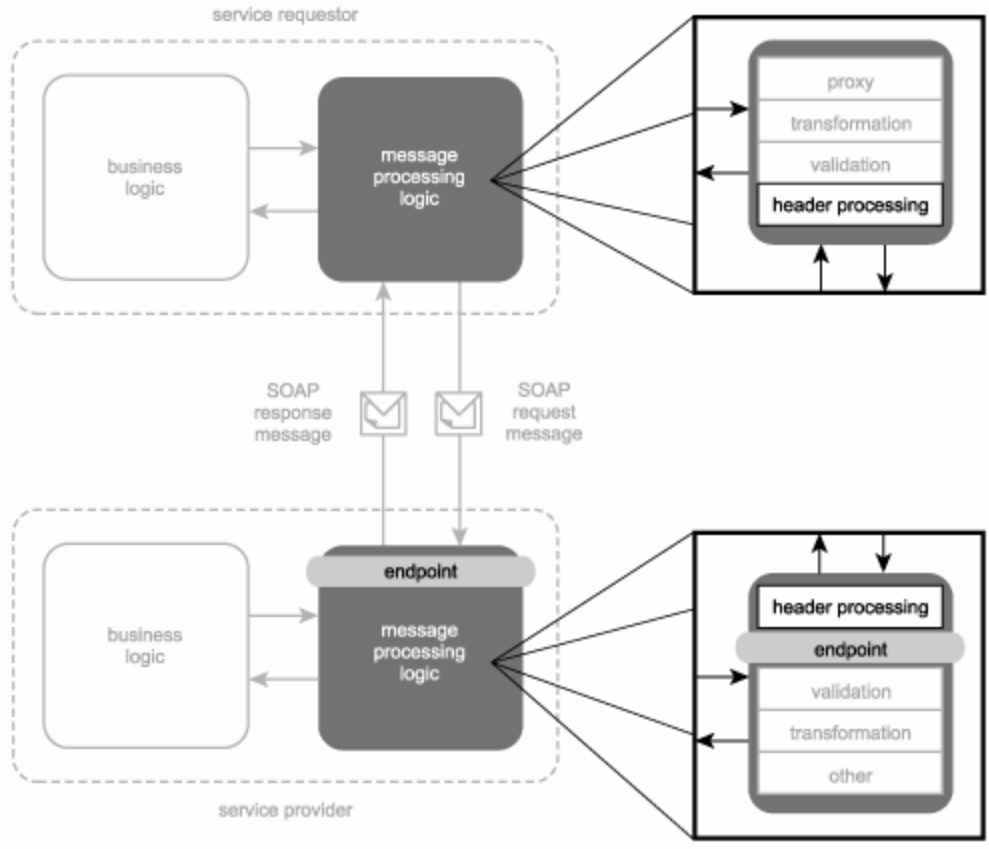
the second component can be considered as belonging to the overall automation logic encapsulated by the service provider.

4.1.4.5. 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. 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:

1. SOAP header processing
2. filtering (based on SOAP header or payload content)
3. authentication and content-based validation
4. logging and auditing
5. routing

Fig 4.12. 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. What's the difference between a service

agent intermediary and an intermediary Web service? The determining factor is typically the availability of a WSDL endpoint. Service agents don't generally have or require one, as they are designed to intercept message traffic automatically. An intermediary that is also a Web service will supply a published WSDL definition, establishing itself as a legitimate endpoint along the message path. Note that a service agent intermediary can be designed to also be a Web service intermediary.

4.2 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.

4.2.1. 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.

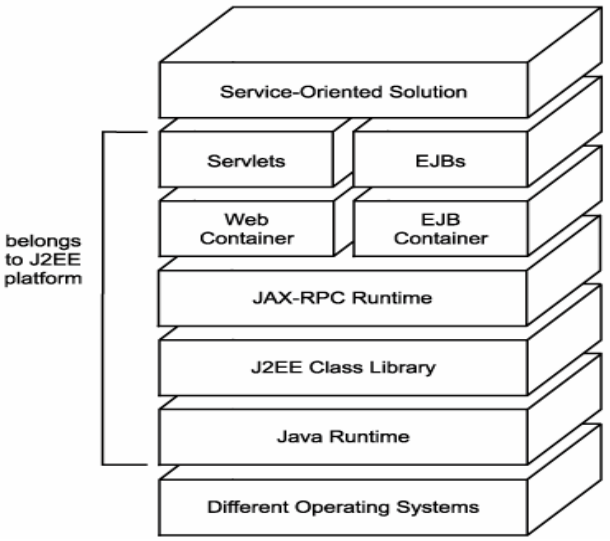


Fig 4.13. Relevant layers of the J2EE platform as they relate to SOA.

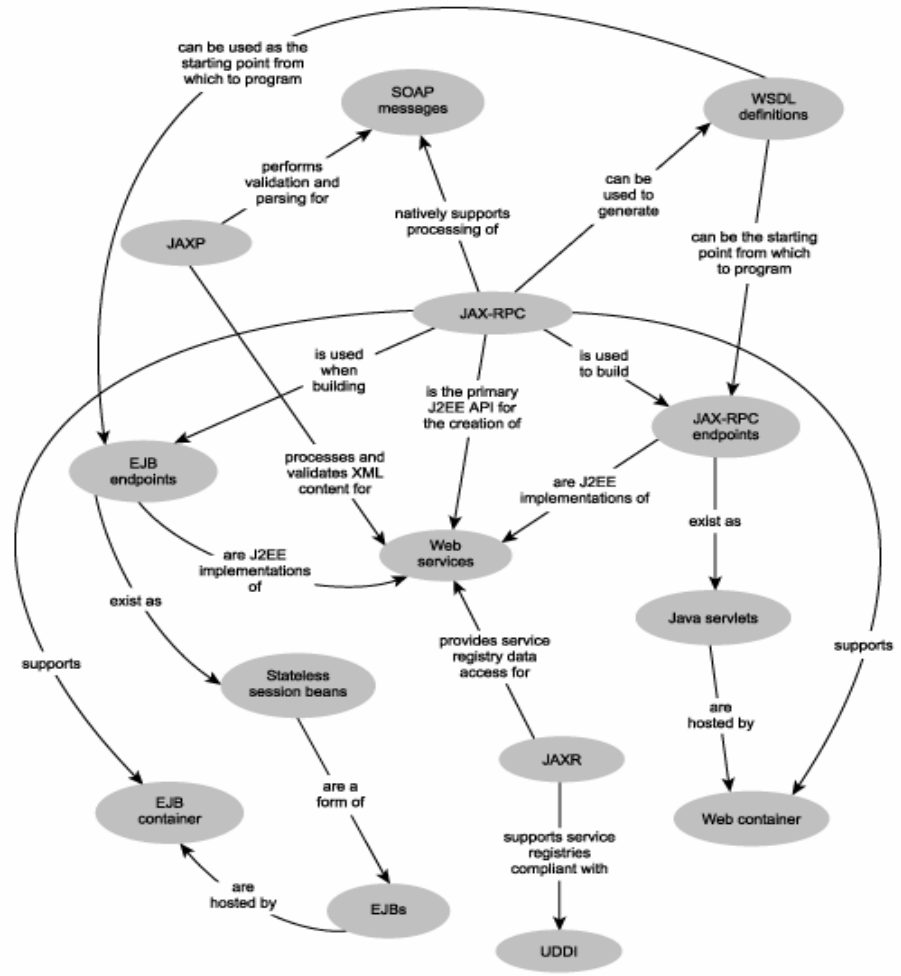
The Servlets + EJBs and Web + EJB Container layers (as well as the JAX-RPC Runtime) relate to the Web and Component Technology layers.Web technology is incorporated is largelydependent on how a vendor chooses to implement this part of a J2EE architecture.

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.

Fig 4.14. How parts of the J2EE platform inter-relate.



- *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.
- *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.

4.2.2. Architecture components

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.

4.2.3. 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.

4.2.4. 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.

4.2.5. 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.saxThese 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.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.

  `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.registryA 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.

4.2.6. 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. 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.

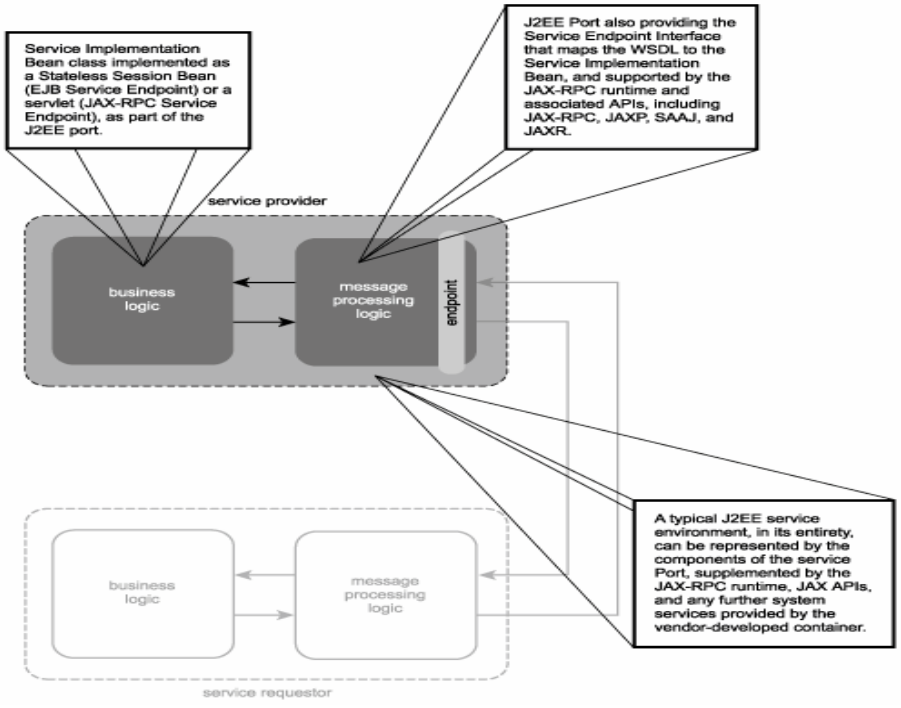


Fig 4.15. A typical J2EE service provider.

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.

4.2.7. 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.

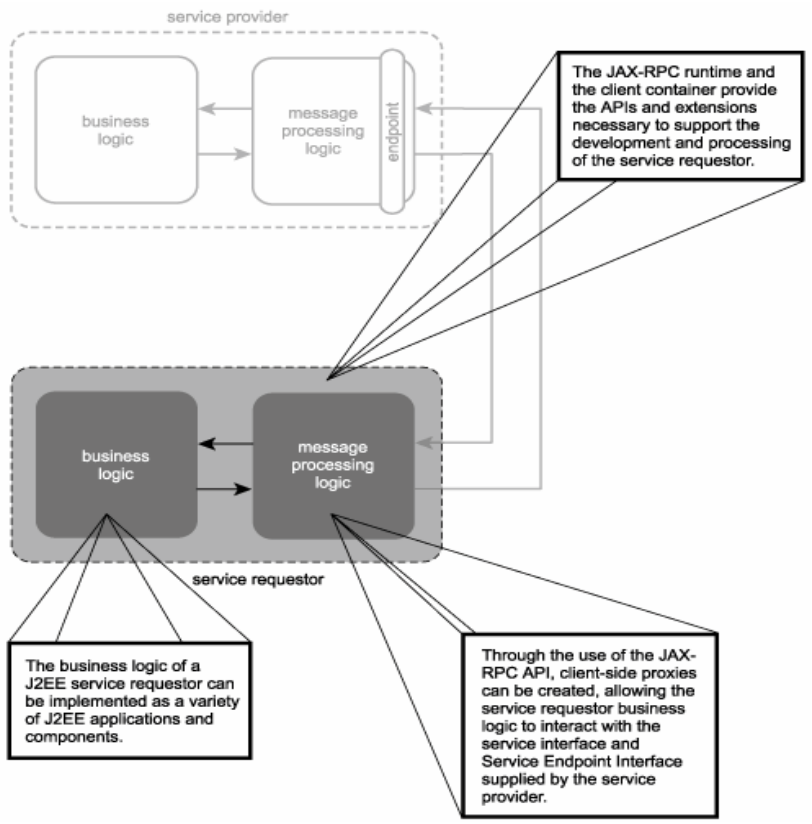


Fig 4.16. A typical J2EE service requestor.

4.2.8. 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.

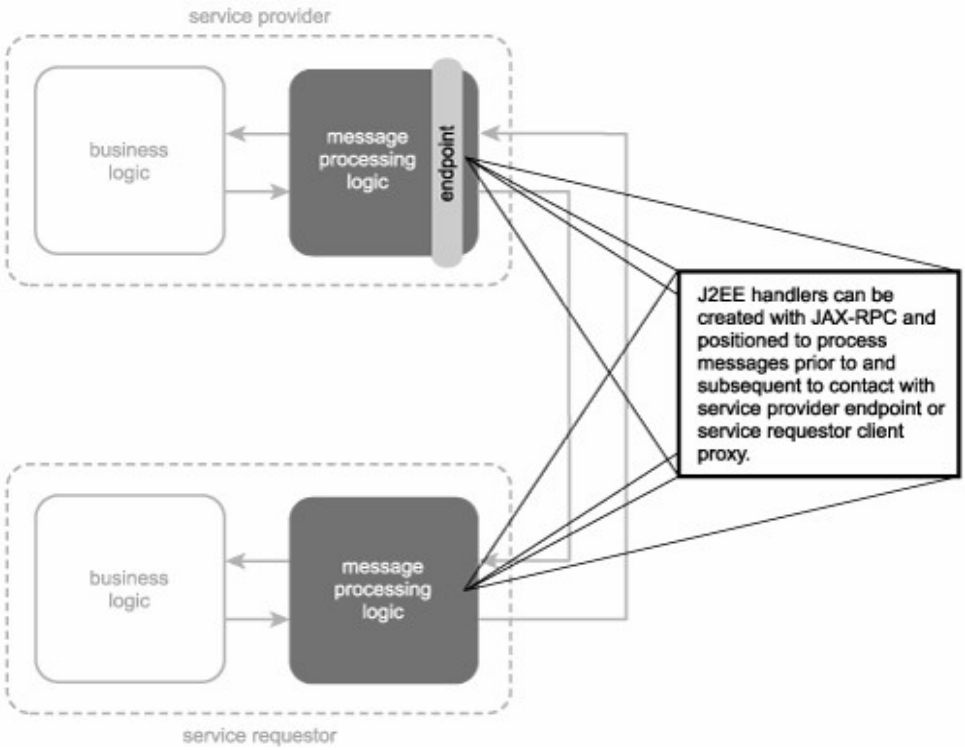


Fig 4.17. J2EE handlers as service agents.

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.

4.2.9. 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. The two examples currently available platform extensions are *IBM Emerging Technologies Toolkit* for WS-* extensions, including WS-Addressing, WS-ReliableMessaging, WS-MetadataExchange, and WS-Resource Framework and *Java Web Services Developer Pack* include WS-Security (along with XML-Signature), and WS-I Attachments.

4.2.10. Primitive SOA support

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

4.2.10.1. 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.

4.2.10.2. 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.

4.2.10.3. 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.

4.2.11. Support for service-orientation principles

We've established that the J2EE platform supports and implements the first-generation Web services technology set.

4.2.11.1. 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.

4.2.11.2. 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.

4.2.11.3. 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*).

4.2.11.4. 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.

4.2.12. 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.

4.2.12.1. Based on open standards

The Web services subset of the J2EE platform supports industry standard Web services specifications, including WSDL, SOAP, and UDDI. The API specifications that comprise the J2EE platform are themselves open standards, which further promotes vendor diversity, as described in the next section.

4.2.12.2. Supports vendor diversity

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.

4.2.12.3. 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.

4.2.12.4. 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 (and an accompanying orchestration service layer) further increases the potential of unifying and standardizing integrated logic. (This is discussed in the Supports service-oriented business modeling section as well.)

4.2.12.5. 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.

4.2.12.6. 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.

4.3. SOA support in .NET

The .NET framework is the second of the two platforms for which it supports SOA.

4.3.1. 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

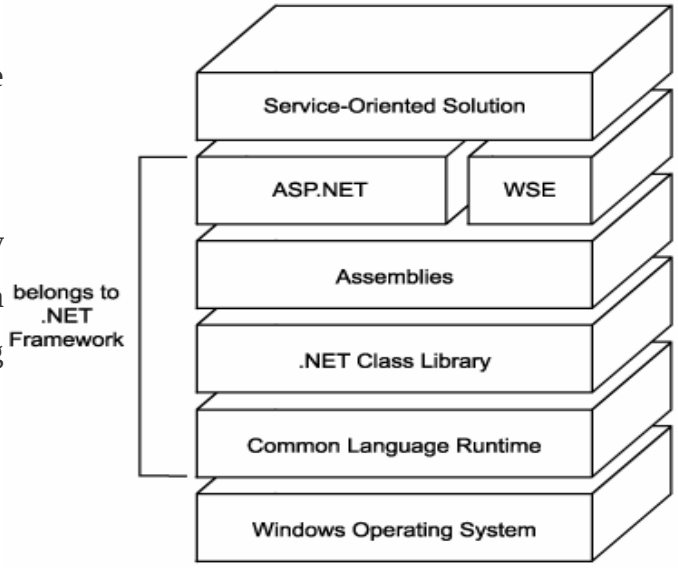
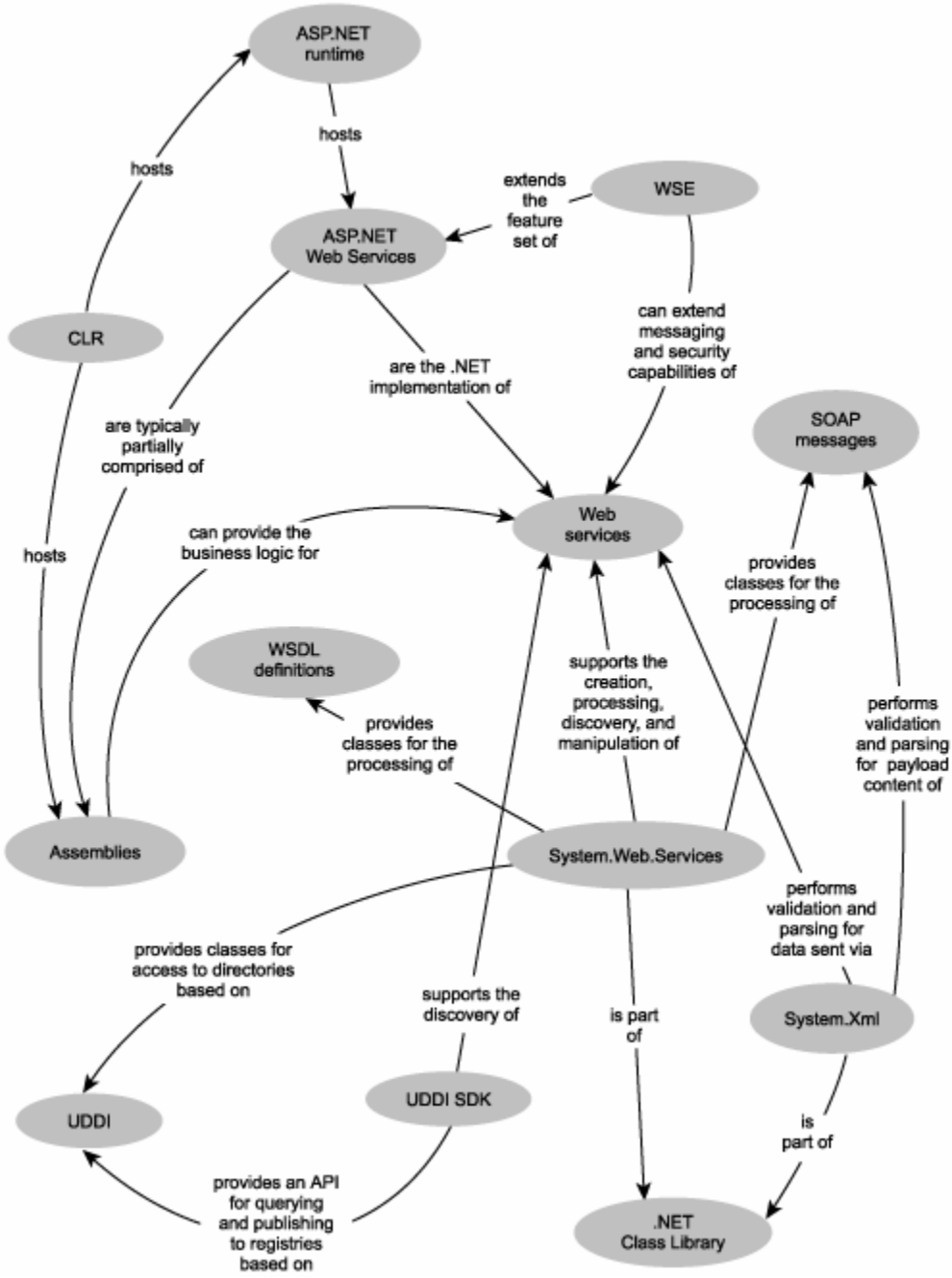


Fig 4.18. Relevant layers of the .NET

framework, as they relate to SOA.

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).

Fig 4.19. How parts of the .NET framework inter-relate.



4.3.2. 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.

4.3.3. 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.

4.3.4. 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.

4.3.5. 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.

4.3.6. 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.

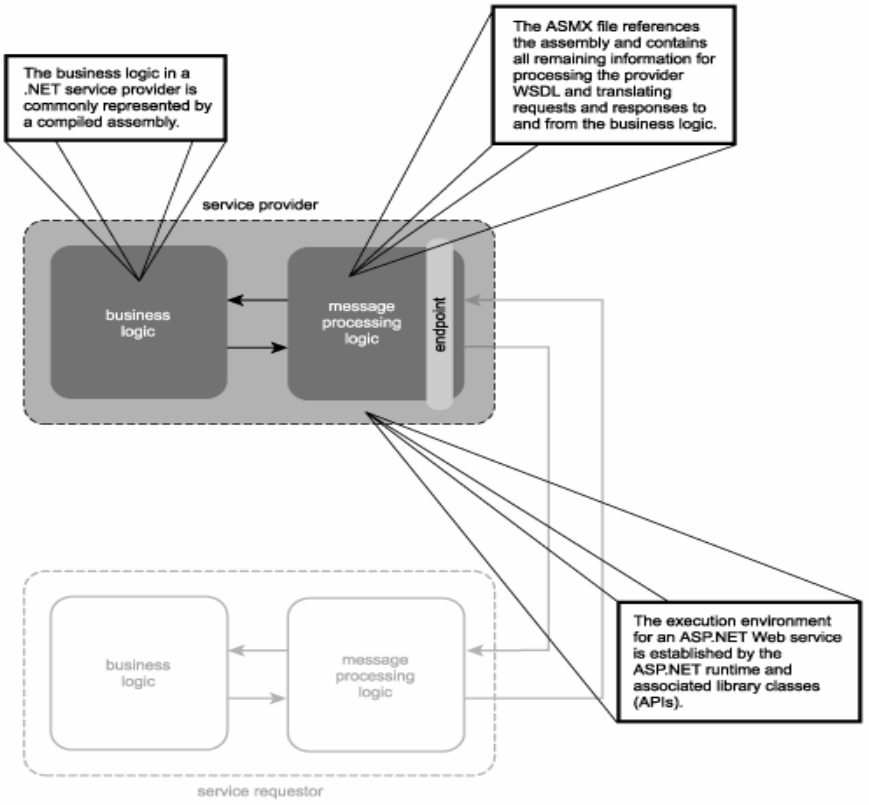


Fig 4.20. A typical .NET service provider.

4.3.7. 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

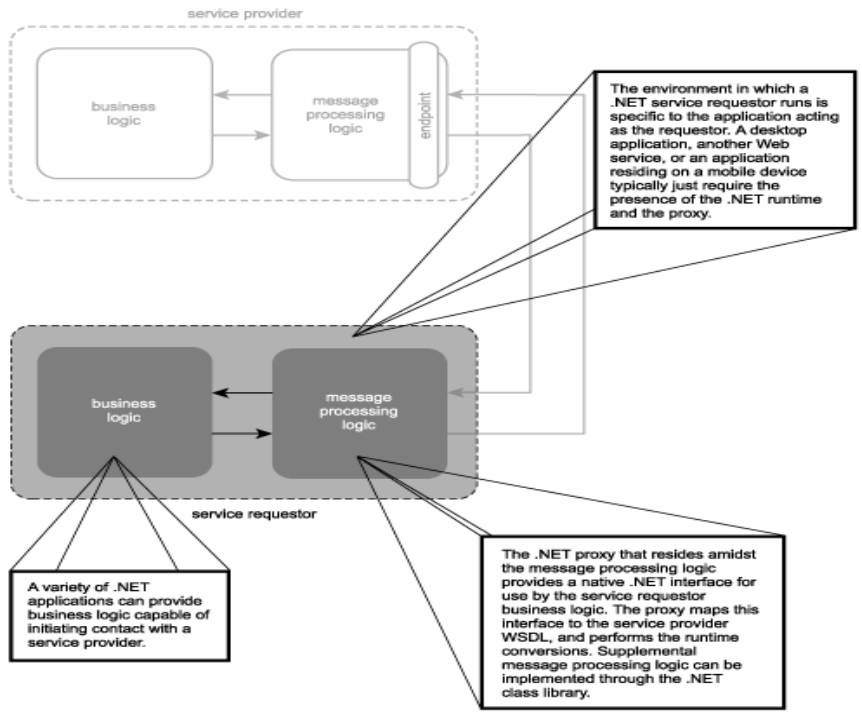


Fig 4.21. A typical .NET service requestor.

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.

4.3.8. 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

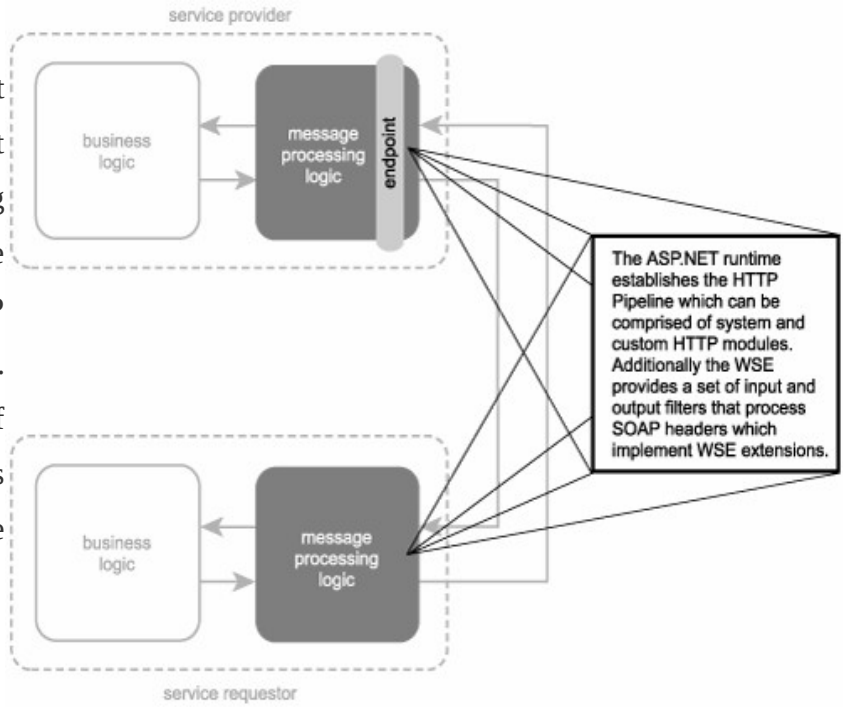


Fig 4.22. Types of .NET service agents.

also can be created to perform various processing tasks prior and subsequent to endpoint contact. 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.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.

4.3.9. 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).

4.3.10. Primitive SOA support

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

4.3.10.1. 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.

4.3.10.2. 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.

4.3.10.3. 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.

4.3.11. Support for service-orientation principles

The four principles 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.

4.3.11.1. 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.

4.3.11.2. 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.

4.3.11.3. 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.

4.3.11.4. 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.

4.3.12. Contemporary SOA support

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

4.3.12.1. 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. 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.

4.3.12.2. 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.

4.3.12.3. 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.

4.3.12.4. 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.

4.3.12.5. 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.

4.3.12.6. 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.

4.4. Integration considerations

Every automation solution, regardless of platform, represents a collection of features and functions designed to execute some form of business process in support of one or more related tasks. The requirements for which such a system is built are generally well-defined and relevant at the time of construction. But, as with anything in life, they are eventually subject to change. When the extent of change is so broad that it affects multiple processes and application environments, it tests an organization's ability to adapt, for example:

- *Cross-platform interoperability* The ability of previously standalone applications to be integrated with other applications that reside on and were developed with different vendor platforms.

- *Changes to cross-platform interoperability requirements* The ability of existing integration channels to be augmented or replaced entirely in response to technical or business-related changes.
- *Application logic abstraction* The ability for existing application logic to be re-engineered or even replaced entirely (often with new underlying technology).

The agility contemporary SOA brings to an organization can be fully leveraged when building integration architectures. The many benefits and characteristics we identified in this book as being attainable via SOA outfit the enterprise with the ability to meet the challenges we just explained. Service-oriented integration therefore empowers organizations to become highly responsive to change, all the while building on the service foundation established by SOA. (Service-oriented integration is explored in the companion guide to this book, Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services.)

Fig 4.23. Disparate solutions communicating freely across an open communications platform. A testament to the inherent interoperability established by SOA.