Introduction

Service-Oriented Architecture (SOA) is rapidly becoming the dominant architectural style in many enterprises. The primary purpose of building an SOA solution is to empower the enterprise to be more responsive to business needs by loosely coupling its systems. One of the main goals of the design of your Web services within an SOA solution is to enable rapid construction of business processes. You also want to facilitate application integration both within the enterprise and with external business partners.

In the context of implementing SOA solutions, the structure of a service interface is very important. A poorly structured service interface can greatly complicate development of many service consumer applications that use that interface. From a business point of view, poorly structured service interfaces can complicate development and optimization of business processes. Conversely, well-designed service interfaces can accelerate development schedules and facilitate business-level flexibility.

Web services are a natural fit for constructing SOA solutions. Many existing and future industry standards in the Web services arena, such as SOAP, Java API for XML-based RPC (JAX-RPC), WSDL, and WS-* specifications) ensure interoperability. Standards-based tools included in
popular development environments, such as IBM® Rational® Application Developer and IBM WebSphere® Integration Developer, reduce development times and accelerate SOA projects.

This article focuses on high-level aspects of service interface design:

- Design and development approaches
- Service granularity
- Operation signature

**Development approaches**

Programming models and development tools based on XML and Web services define three approaches to building Web services:

**Bottom up**

Leading integrated development environments (IDEs) provide tools for creating Web service implementations from existing code (for example, Java™ or COBOL). With this approach, the developer usually selects an existing JavaBeans or an EJB component and invokes a wizard that generates a WSDL file that can be used to invoke the bean or EJB as a Web service.

**Top down**

Following this approach, the developer first defines the Web service interface using WSDL and XML Schema (XSD) constructs, then generates skeletal implementation code for the service. Next the developer completes the skeleton service implementation. Most leading IDEs, such as Rational Application Developer V6 and WebSphere Integration Developer V6, provide tooling support for this approach.

**Meet in the middle**

This approach involves a combination of the previous two. The developer first defines the service interface using WSDL and XSD, and generates a skeletal implementation for the service. If necessary, the developer may also use a bottom-up technique to expose existing code using a convenient application programming interface (API). Then the developer writes code that converts between the newly designed interface and the old interface.

Many skilled Java developers like to use bottom-up techniques to accelerate Web services development in SOA projects. They develop implementations for new services in Java first, then use the powerful code generation wizards to create WSDL interfaces for these services. Although this approach can speed the implementation of individual services, it frequently means problems for the SOA project as a whole.

Problems occur because bottom-up generation frequently results in type definitions that cannot be reused and multiple types defined to represent semantically equivalent information.

**Best practice:** Use the top down and meet in the middle development approach, rather than bottom-up techniques. Design your service interface using XSD and WSDL, then generate skeletal Java code.

The bottom up development approach is appropriate when there is an existing body of legacy code (for example JavaBeans, EJB, COBOL, and so on). With this approach, you should carefully
review interfaces of existing classes before generating the WSDL interface. If the Java interface contains any of the following, it can be considered weakly typed:

- java.lang.Object used as either parameter or return type for a method
- Collection classes (for example, java.util.Vector) used as either parameter or return type for a method (JAX-RPC constraint)

You should consider either refactoring your legacy code to ensure that interfaces are strongly typed or building a mediation that will wrapper the weakly typed interface with a strongly typed one.

**Java compared with WSDL**

You can use either Java or WSDL to describe your service interface. Web services-related specifications such as SOAP, JAX-RPC, and JAX-B define mappings that describe how types defined in Java can be mapped into WSDL/XSD and vice versa.

**Best practice:** Describe your service interfaces using WSDL and XSD instead of Java. Your service interface definition is a WSDL port type.

The XML Schema specification defines a broader range of constructs for describing message structures than Java. Some of these include choices, derivation by restriction, annotation, and others. Therefore, it is better to define your interface using WSDL and XSD, and generate skeletal Java code rather than doing it the other way around.

Together WSDL and XSD represent the technology-neutral interface definition language that can be used for SOA implementations. The WSDL/XSD interface definition can be used to generate skeletal implementations in many languages in addition to Java (for example COBOL and C++)

**Interface granularity**

A service interface should generally contain more than one operation. Operations defined as part of a single service interface should be semantically related. A large number of services, each containing a single operation or small number of operations, indicates inappropriate service granularity. Conversely, a very small number of services (or a single service) containing a large number of operations likewise indicates inappropriate service granularity.

Let's use an example to better understand how decisions about service interface granularity should be made. One of the most commonly encountered scenarios in SOA projects involves exposing existing transactions as Web services. In this example, an existing S390 mainframe hosts a CICS region that runs many COBOL transactions used to manage customer information, product pricing, and product availability.

**Best practice:** A service interface (WSDL port type) should generally contain more than one operation. Operations defined as part of a single service interface should be semantically related by data on which they operate.
Each COBOL transaction can be exposed as a single Web service operation. We could define a service called MyS390Service, for example, with a single interface that defines operations for all COBOL transactions running on that mainframe. This produces an interface with dozens of operations that client applications can use to invoke any transaction on that system, regardless of whether the transaction is related to customer management or product pricing.

This approach makes the service harder to understand and, as a result, hard to reuse in business processes -- ultimately resulting in many versions of the service. (More about service versioning in subsequent articles.) We encourage developers to not group operations based solely on the physical system that they are targeting.

Another approach involves defining a new interface for every transaction on the system. This results in many interfaces and eventually, many services that use these interfaces. Service proliferation, in turn, results in service governance problems, making it harder to pursue effective code reuse.

The best approach is to define interfaces (WSDL port types) in a way that groups semantically related transactions. In our example, COBOL transactions that operate on customer information are semantically related in that they operate on the same set of data.

If customer information resides on multiple enterprise information systems (EIS), rather than on one mainframe as in the previous example, you should first define physical system-specific interfaces grouping customer information-related transactions, and then aggregate these interfaces into a single interface for customer information management. Figure 1 shows the interface aggregation approach.

**Best practice:** If related information resides on multiple EISs, you should first define physical system-specific interfaces grouping information type-related transactions, and then aggregate these interfaces into a single interface.

In Figure 1 you see an example of aggregating system-specific interfaces into a generic interface. EIS1 provides access to customer information, such as addresses. EIS2 contains customer account data. The generic CustomerInfo interface combines operations from the two EIS-specific interfaces.
Figure 1. Aggregating system-specific interfaces into a generic one

Operation signature

This section discuss the following operation signature semantics:

- Synchronous compared with asynchronous interfaces
- Stateful compared with stateless design
- Use of faults
- Header compared with payload

Best practice guidelines in this section can help you design services that can be readily reused and incorporated into business processes.

Synchronous compared with asynchronous interfaces

A WSDL port type can contain one or more operations. Operations can be one-way or request-response. A one-way operation can define a request message but no response message. It is not possible to define faults messages for a one-way operation. (See Related topics for more information on WSDL.)

As soon as a client application invokes a one-way operation using, for example, a JAX-RPC-compliant Java proxy, it returns control immediately to the calling client application thread. There is no way for the client application to know whether or not the message was successfully delivered or even dispatched.

This may, or may not, be acceptable to the calling application. If it is acceptable, then the application can invoke one-way operations and rely on the message-oriented middleware, such as SIBus or WebSphere MQ, to ensure message delivery to its intended destination. If this is not acceptable, then the application can use a synchronous invocation technique to implement asynchronous semantics (described in a moment).
A request-response operation can define a request message, a response message, and any number of fault messages. When a client uses a synchronous protocol (such as HTTP) to send a request message (for example, a JAX-RPC-compliant Java proxy), the proxy blocks the calling thread until it receives either a response or a fault from the service.

Faults convey error information about failures that occur during the service invocation. In many processing scenarios, this information is just as important as the data returned during a "happy-path" invocation.

Services are frequently invoked by end-user-facing applications that need to convey error information to the end user. Many business processes need to immediately examine error information returned by a service invoked with a synchronous binding, allowing them to direct subsequent processing appropriately. In these cases you should always strive to design your interfaces using request-response operations that use faults. (Faults and error handling are covered in more detail in a subsequent article in this series.) Use a synchronous protocol with a request-response interaction pattern, and define faults that are understandable by the end user.

**Best practice:** Define faults in your service interfaces and use them in your service implementations.

Asynchronous interactions come in two different flavors:

**One-way invocations**
The service requestor does not expect or need a response. The application or the business process simply drops the message off to be delivered to the intended destination and continues processing.

**Asynchronous request with delayed response**
The service requestor dispatches the request message and subsequently polls the service for the response, or a callback is dispatched to the requestor.

**Best practice:** When designing a new service, do not mix synchronous and asynchronous invocation semantics in a single interface (WSDL port type). If it is advantageous to support both semantics, define separate interfaces for synchronous and asynchronous invocations.

Listing 1 shows an example of using synchronous operations to implement asynchronous invocation semantics. The transaction `debitAccount` does not have to return a value. By adding a return value to the operation you are allowing for fault handling in client applications.
Listing 1. Using synchronous operations to implement asynchronous invocation semantics

String transNumber;

Try {
    transNumber = debitAccount(amount);
} catch (SystemFault sysFault) {
    System.out.println(sysFault.getError());
    // React to system level fault
} catch(BusinessFault busFault) {
    System.out.println(sysFault.getError());
    // React to business level fault
}

Sometimes a combination of synchronous and asynchronous is used to achieve desired behavior, although this becomes complicated from a service interface standpoint. Unfortunately WSDL doesn't provide a good way to model asynchronous behavior.

In the example above, the calling application dispatches the request message using a request-response operation that returns a transaction number as the response message. The calling thread blocks until it receives a confirmation that the message has been successfully delivered to its intended destination. If problems are encountered, then both system-level and business-level faults can be raised by either the service provider or the Web services invocation engine, and caught by the calling application (synchronous behavior).

The calling application uses the transaction number to poll the service provider interface for the business response message at a later time (asynchronous behavior). You could also return a Boolean to simply indicate success, when the calling application is not interested in the response and to convey that the request message is successfully delivered.

Stateful compared with stateless interfaces

Exchanges between services can be stateful or stateless in nature. A stateful, or conversational, exchange between services occurs when the service provider retains knowledge of data that has been exchanged between the service consumer and the service provider during preceding operation invocations.

For example, a service interface could define operations called setCustomerNumber() and getCustomerInfo(). In a stateful exchange the service requestor calls the setCustomerNumber() operation first, passing in the customer number. The service provider retains the customer number in memory. Next the service requestor calls the getCustomerInfo() operation. The service provider then returns a customer information response that corresponds to the customer number set in the previous invocation.

In a stateless exchange, the service provider defines the getCustomerInfo() operation so that it takes the customer number as an input parameter. The service provider does not need to
define the `setCustomerNumber()` operation nor does the service requestor need to invoke it. Each operation invocation represents a separate transaction, with the request message containing all of the necessary information to complete it.

Stateless interfaces are considered superior in the context of building an SOA. A stateless interface can be readily reused by many service consumer applications that are free to manage state in the manner best suited to each application.

**Best practice:** Design your service interfaces for stateless interactions. The request message passed in to the operation should contain all information necessary to complete that operation, regardless of the sequence in which other interface operations are invoked.

**Header compared with payload**
The request messages contain data that will be used by the service to perform the business logic of the operation. These messages can also contain data that are more pertinent to system-level processing associated with the transaction, rather than business logic performed by the transaction. Examples of these data include:

- Identification of the service requestor application
- Service implementation version
- Dispatch and receipt timestamps

Similarly, the response message issued by the service operation can contain system-level data, such as:

- Identification of the responding application (service provider)
- Receipt and dispatch timestamps
- Computed response time

These system-level data have to be processed either by the service provider application, in addition to the business-level data, or by the enterprise service bus (ESB) infrastructure. In the context of building SOA solutions, it is much better to structure your service interface so that system-relevant data can be processed separately from the business-relevant data.

The SOAP specification stipulates that the SOAP message can contain the SOAP header, the body, and any number of user-defined headers. You should define and use custom headers to carry system-relevant information that is specific to your business or project. Avoid putting system-relevant information into the body of your message. This allows the ESB infrastructure to process the information without parsing the message body (performance intensive).

**Best practice:** Define and use custom headers to carry system-relevant information that is specific to your business or project. Avoid putting system-relevant information into the body of your message.

**Summary**
SOA allows enterprises to evolve their IT infrastructure in a flexible way. Web services provide an ideal technology for implementing an SOA. Well-designed service interfaces can facilitate an SOA
implementation, while poorly designed ones can greatly complicate it. In this article you explored best practices for the high-level design of your service interfaces.

Stay tuned for subsequent articles, which describe best practices for structuring your WSDL service definitions, techniques for error handling, partitioning information between message body and headers, and batch processing.
Related topics

- Read more about the IBM vision of SOA and IT infrastructure solutions.
- "Service-Oriented Architecture expands the vision of Web services, Part 1" (developerWorks, Apr 2004) has more information about implementing SOA with Web services.
- "Tip: Implement implicit and explicit SOAP headers" (developerWorks, Feb 2005) explains the difference between explicit and implicit headers, and how the differences might impact you when developing with JAX-RPC.
- Learn about wsdl:header and wsdl:headerfault elements from the Web Services Description Language (WSDL) 1.1 Specification.
- Web Services Description Language (WSDL) 1.1 provides information about WS-Addressing and WS-Security.
- Download an IBM trial version of Rational Application Developer.
- Download IBM product evaluation versions and get your hands on additional application development tools and middleware products from DB2®, Lotus®, Rational, Tivoli®, and WebSphere.

© Copyright IBM Corporation 2007
Trademark
(www.ibm.com/developerworks/ibm/trademarks/)