Service Oriented Architecture

GETTING IT RIGHT

Contributors:

AmberPoint
BearingPoint
Composite Software
MomentumSI
Progress Software

Editor:

Jim Green

Service Oriented Architecture

GETTING IT RIGHT

Contributing Authors:

David Besemer
Paul Butterworth
Luc Clément
Jim Green
Hemant Ramachandra
Jeff Schneider
Hub Vandervoort

Editor:

Jim Green

An Implementor's Guide to Service Oriented Architecture Getting It Right

www.SOAguidebook.com

ISBN-13: 978-0-9799304-0-9
ISBN-10: 0-9799304-0-5

First Edition
Printed April 2008

Printed and designed by Westminster Promotions.

CHAPTER 2

Designing Services

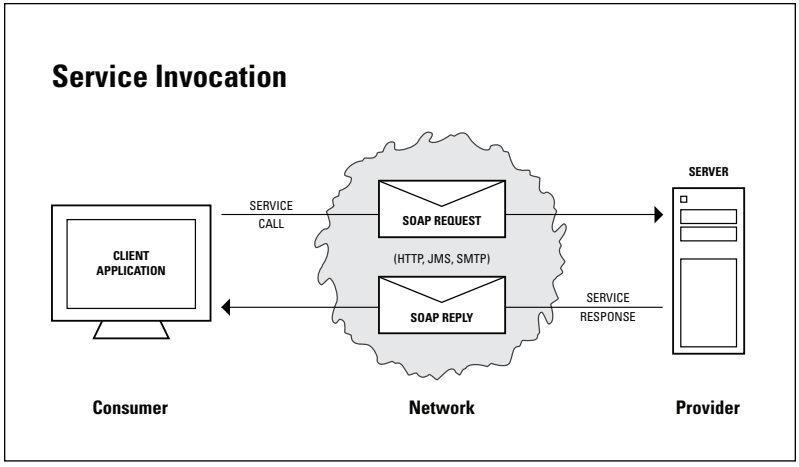
David Besemer
Chief Technology Officer
Composite Software

2.1 Services Introduction

In a service oriented architecture, *services* provide the basis for communications between systems and technologies. Services are well-defined units of functionality that are accessible over the network via standard protocols. They are invoked by software, and are not accessed by a human user. In other words, services are more like a remote procedure calls. The system that implements a service is called a *provider*, while the system that uses the service is called a *consumer*.

Services can be built in a variety of ways, but standards and guidelines exist to promote interoperability and reuse in an enterprise-class service oriented architecture. The central standards relevant to service implementation and deployment are *XML, SOAP, WSDL, and UDDI (refer to the following illustration),* and services that conform to these standards are called *web services*. A web service is actually a collection of individual service operations, each of which can be thought of as an individual procedure.

Figure 2.1: Service Invocation



KEY RECOMMENDATIONS:

- Base your services on vendor independent industry standards to ensure the best reuse and interoperability.
- Create and deploy your services in an appropriate and best-of-breed infrastructure to ensure operational efficiencies (e.g. an *information server* for data services; an *application server* for transaction services.)
- Design service interfaces that are simple, consistent, well-documented, and motivated by business requirements to ensure adoption, reusability, and expandability.
- Employ security policies to meet the business needs of your enterprise.

INDIVIDUAL SERVICE OPERATION

An individual service operation is invoked using a SOAP call, which encapsulates the service request message (and subsequently, a response message) for transport over the network – you can think of it as the envelope that contains a letter. The SOAP call can be transported between consumer and provider over a variety of mechanisms such as HTTP, SMTP, or a message bus. Because of the wide availability of HTTP infrastructures within enterprises, most web service calls today are transported via HTTP. Recently, however, the use of message buses (ESBs) has been increasing for transporting web service calls.

SERVICE REQUEST AND MESSAGES

The service request and response messages themselves are written in XML. The SOAP standard defines two possible XML message formats, RPC and document, and two encodings, SOAP and literal. Most experts agree that the best way to ensure interoperability is to use the document format with literal encoding.

WEB SERVICE SPECIFICATION LANGUAGE DOCUMENT

The complete specification of a web service (i.e., the location of the service on the network, the specific operations available, and the request and response message formats, etc.) is embodied in a WSDL document, which service consumers consult to figure out how to use the service. The WSDL can be considered the API definition for a web service, and as such, it defines the contract between provider and consumer.

UDDI DIRECTORY

WSDLs are often catalogued in a UDDI directory that consumers consult to discover services and their providers.

Unfortunately standards alone are not enough to ensure service interoperability. Additional guidelines have been created by an organization called the *Web Services Interoperability Organization* (WS-I). WS-I's *Basic Profile* defines best practices within the Web service standards and promotes the highest possibility for reuse and platform independence. Organizations can benefit greatly from following recommendations of the WS-I Basic Profile for their service development and deployment.

Services generally either provide data to the consumer, or they create or modify data in an underlying system. The former are called data services, and the later are called *transaction services*. An example of a data service might be *retrieveOrdersForCustomer*, which might take a customer number as an input parameter. An example of a transaction service might be *updateOrderShippingStatus*, which might take an order number and the updated shipping status as input parameters. These services present separate challenges to the service provider and they are generally created and deployed using different infrastructures. Data services are created and deployed in an *information server,* while transaction services are created and deployed in an *application server*. These different types of services and their associated infrastructures are described in detail later in this chapter.

Getting started with service development and deployment in your enterprise does not have to be difficult or expensive. Rather than following a 'boil the ocean' approach that seeks to define all enterprise-wide services needs in advance, it is commonly recommended to take an incremental, organic approach to service development and deployment. Choose a project that will benefit from a service-oriented approach and begin creating a collection of services needed for that specific project. Once the first project is in production, select another project can reuse some of the services from the first project. You will more than likely need to create new services for your second project, but you will probably be able to reuse one or more of the services created for the first project. When reusing services, you may discover that the services you created for the first project require modification or augmentation to facilitate reuse, which is perfectly normal. Because the collection of consumers is limited at this point, you will usually be able to modify them with little effort. More important, you will have learned what it takes to create reusable and scalable services for your enterprise. This pragmatic, incremental methodology allows you to show value quickly and to refine your strategy as your service usage grows.

Securing service calls can be a complex topic, but the good news is that there are relatively straightforward approaches to security that can be implemented easily. As with services standards, there are both standards and best practices that can be combined to prescribe an approach that we will explore later in this chapter.

2.2 Data Services

An estimated two thirds of all services will be data services, making them the most prevalent form of services in an enterprise. Data services provide data to a consumer in a form that addresses current and ongoing business demands. The focus of data services is to make it easy for consumers to access and use enterprise data in support of their business processes. However, in many cases this requested form of the data does not match how the data is stored in legacy systems, so the data must be transformed, aggregated, combined, or otherwise modified to support current business needs. This is the primary role of a data service: To virtualize (abstract) data from its native form for use (and reuse) in the modern enterprise, while hiding (encapsulating) the complex work of getting the data into a form for consumption. However, providing data to a service consumer in an appropriate form can be challenging for a variety of reasons, including:

- Data required to satisfy demand may be distributed amongst two or more systems. For example, the bulk of information about an order might be stored in the ERP system (e.g., SAP), but customer interactions regarding the order might be stored in a CRM system (e.g., SalesForce.com).

- Protocols for getting data out of the underlying systems are vendor specific and highly varied. You may be able to retrieve customer data directly from your customer master using SQL, but you might have to use a web service call—or worse—a vendor-specific API to get the order information from the ERP system.

- The format of the data from the underlying systems is probably not XML, and as a result, will require transformation prior to supporting a web service call. The native format possibilities for the underlying data are numerous (e.g. relational, delimited, proprietary, hierarchical, etc.) and manually mapping these to XML is not practical.

- Legacy semantics of the data will not necessarily match current use cases. For example, prior to the dot-com era, an internal data source might have been created to hold information about a customer. At that time, it was reasonable to establish fields regarding 'marketing opportunities', In the current usage, however, that same data might be presented to a customer in a self-service portal as 'privacy preferences'.

- Approximately ten percent of enterprise data is replicated in data warehouses and data marts, while the remaining ninety percent is in operational systems. It is important to maintain high levels of performance in these operational systems. Data services need to optimize data access performance as well as utilize intelligent caching and other advanced techniques.

2.2.1 Data Services Levels

Transforming data from its native 'physical' environment to its required 'virtual' form can comprise a complex and difficult set of operations. One recommended approach to address these data transformation challenges is to break the problem into smaller pieces (see Figure 2.2), which manifests itself as layered services of varied granularity, including:

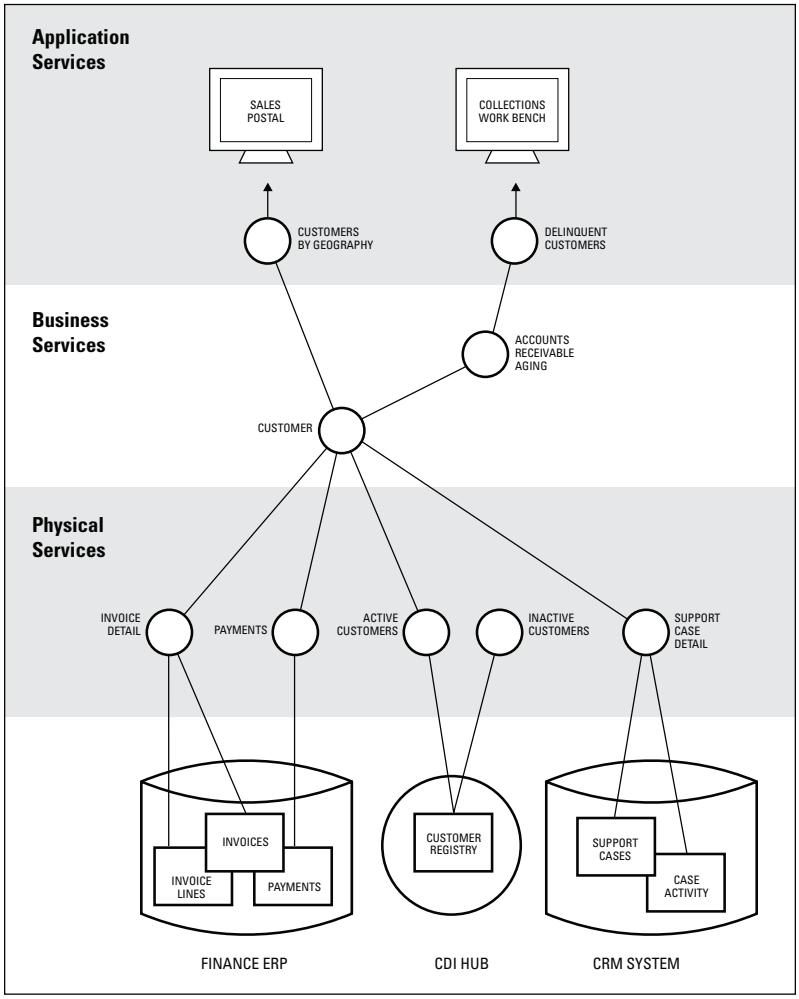
- **Physical Services.** Physical services lie just above the data source and they transform the data into a form that is easily consumed by higher-level services. For a well-designed database, these services may be unnecessary because the data can be understood and used as is. However, many packaged applications store their data in a form that is designed for optimal use within that application, and that form of the data does not lend itself well to direct and transparent access. For this kind of data, it is very useful to layer a collection of light transformation services just above the physical layer. These services can change element names, cast data types, and augment record contents. The output of these services can still be considered relatively raw, physical data, but it has been put into a form that is cleaner and more useful.

- **Business Services.** Business services embody the bulk of the transformation logic that converts data from its physical form into its required business form. These services should be thought of as a provider of the canonical data representations for your business (e.g., customer, supplier, product, order, shipment, etc.). There may be several 'layers' of business services—especially if intermediate transformations are useful as business services in their own right. For example, if your company sells cellular and residential phone service, you may have a 'customer' business service, and above it you might also have a 'cellularCustomer' business service (which leverages but refines the 'customer' service). Business services can be seen as providing master data and transaction data to the rest of your processes.

- **Application Services.** Application services leverage business services to provide data optimally to the consuming applications. Application services are lightweight wrappers that match the business services with their actual usage in the application layer. If the application layer is a modern BPM environment, no transformation may be necessary – that is, it may be possible to use the business services directly via SOAP invocations. On the other hand, if the application layer is a business intelligence platform, it probably needs to access the data as if it were stored in a database. So an application service that looks like a virtual database table will be necessary. As application services are created and used, discipline should be applied to avoid business logic creeping into this layer. If data is transformed with business logic, that logic should reside in the business services layer.

The elimination of duplicated enterprise data and increased opportunities for reuse are the main advantages of establishing logical layers within the pool

Figure 2.2: Data Services Levels



of data services. With these logical levels of service granularity in use, you will find that the business services can be reused throughout the enterprise with few additional transformations required.

2.2.2 Data Services Infrastructure

The challenges associated with providing data services, beyond the usual scalability and high-availability production needs, dictate the need for an environment designed specifically to easily create, deploy, and maintain data services. This infrastructure environment is called an 'information server' and several vendors offer products in this category. An information server is distinctly different from an application server (which will be discussed in the

Figure 2.3: Comparing Information and Application Servers

Data Services Information Server	**Transaction Services** Application Server
Key Features	
Container for Data Services	Container for Transaction Services
Data Access Standards	J2EE Standards
Data Federation	Session Management
Data Retrieval Performance	Memory/Thread Management
Data Transformation	Concurrency
Data Caching	Security
Data Security	

next section on transaction services). Most mature SOA infrastructures will have both an information server and an application server. (see Figure 2.3)

When selecting information server infrastructure software on which to build your data services layer, there are many things to consider, including:

- **Adherence to Standards.** The key tenets of a service oriented architecture are loose coupling and reusability. It is impossible to achieve either of these if your services do not conform to standards and best practices.

- **Performance and Scalability.** The run-time execution of individual data services must be intelligent and efficient, and the overall infrastructure must provide massive scalability. Advanced query planning and optimization are the keys to intelligent execution – it's not enough to simply throw more processing power at the problem.

- **Ease-of-Use.** One reason to use an infrastructure that focuses specifically on data services is to eliminate work that would otherwise be done elsewhere. If the environment is not easy for developers to use and maintain, adoption will be slow and efficiencies will be lost.

- **Data Caching.** In addition to being a virtualization layer, the data services infrastructure is also an insulation/buffering layer. This cannot be effectively accomplished without providing a caching mechanism. There should be both implicit and explicit caching opportunities, and it should be possible to cache both query results and procedure calls.

- **Access to Data Sources.** An enterprise's data services layer must provide access to all structured enterprise data. This includes relational databases,

third-party data services, packaged applications (e.g., SAP, Siebel), files (e.g., Excel), directories (e.g., LDAP), and legacy mainframes (e.g., VSAM). It should also provide the capability to expand its reach through custom development, allowing even the most obscure data source to participate in the data services layer.

- **Data Quality Management.** A significant amount of enterprise data is dirty and incomplete. Some of the messiness can be addressed with straightforward transformation capabilities, but some of it must be attacked with robust data cleansing functionality.

- **Strong and Flexible Security Mechanisms.** Exactly what your enterprise needs will be determined by your industry and business requirements, but the infrastructure software should provide general purpose mechanisms to implement a variety of security measures.

- **Vision and Focus.** The challenges associated with the data services infrastructure comprise a discipline that is unique. The vendor you choose to provide this capability to your enterprise should be clearly focused on this problem, and have a vision for advancing the state of the art. Several vendors claim data services functionality as part of their broad offerings, but that slice of the platform will never get the focus it needs to be effective. We recommend that you select a vendor that offers best-of-breed in data services technology.

2.2.3 Enterprise-wide Data Services Layer

As the collection of reusable data services in your enterprise grows and the production requirements of the service consumers become more demanding, the information server will expand to form an enterprise-wide data services layer. This clustered and highly available infrastructure establishes a virtualization layer between enterprise systems that store data and enterprise applications that use data. The presence of this data services layer in an enterprise provides several long-term benefits, including:

- Consumers of a particular type of data will get that data from the same shared service, ensuring consistency of data across the enterprise.

- New business application requirements are less daunting since the IT organization can now provide the application developer with the exact data they need to be most effective—regardless of how the data exists in the underlying systems. This sort of data access agility is unheard of in today's enterprise IT environment.

- Data consumers will be decoupled from the underlying physical systems, allowing legacy systems to be changed, migrated, or retired without affecting the consuming applications. Only the data services layer will need to be modified to accommodate the underlying physical changes.

- As data capacity requirements grow, the data services layer can be scaled to accommodate increasing demand. And because caching is available in this layer, it may not be necessary to add corresponding capacity to the underlying physical data source.

- System consolidation will require data to be grafted from only one of the affected systems into the data services layer without affecting the high-level business applications. This efficiency overcomes the consolidation chaos commonly resulting from mergers and acquisitions.

2.3 Transaction Services

Transaction services implement individual business operations that are executed as part of a larger business process. The effect of invoking a transaction service is the creation or modification of data in an underlying data repository. The logic encapsulated in a transaction service represents your enterprise's definition of what it means to, for example, create a customer or update an inventory level.

Some transaction services will be provided inherently as part of a packaged application (e.g., SAP), and a user indirectly invokes them when a user employs the application's user interface. Although many packaged application vendors do not yet provide their functionality as standard services for use outside their user interface, most are moving in this direction.

Other transaction services will need to be developed to implement specific, unique business logic. These services are generally built by IT developers in a software development environment like an application server (e.g., IBM WebSphere). These environments offer powerful development tools and efficient deployment environments. They also provide standard security and transaction frameworks.

Transaction services generally modify data in a single underlying data source, and they are therefore generally connected directly to that data source (rather than relying on the data services layer as an abstraction). This tight coupling is acceptable because a collection of transaction services normally 'owns' the data source it is modifying. However, transaction services also often need access to data to carry out their business logic. For this data they should invoke the same data services that everybody else uses (through the data services infrastructure).

As the names imply, transaction services implement the logical equivalent of a business transaction (e.g., place an order). As such, an important characteristic of a transaction service is that it either completely succeeds or completely fails, leaving no artifacts or incorrect data behind. This is not difficult if the transaction service is modifying a single relational database that implements transaction semantics, but it can be more challenging if it is working with a set of underlying (finer grained) transaction services that are inherently stateless, or if its transaction data is split among more than

one data source. The application server environment usually provides strong transactional models that will assist the developer with this challenge, but the developer needs to use them.

Just as important as the transactional integrity of the service, it is critical to define the scope of the service at the appropriate granularity: Your transaction services should provide business-level granularity so the consumer is not required to think about the interplay between fine-grained physical data components. For example, if you wanted to provide a service for updating the on-hand inventory level for a product, the service should simply take the increment or decrement amount as input, and then internally handle the possibilities for concurrency. As another example, if you wanted to provide a service that deletes a customer, the service should also delete the customer's associated orders, payments, service calls, etc. In other words, the consumer of the service should not have to know the business rules associated with deleting a customer; the service should simply encapsulate the rules and offer the comprehensive service to the consumer.

A transaction is not a substitute for application integration which would be accomplished with an ESB layer or a similar system with traditional EAI capability. That is, it should not be the responsibility of the transaction service to update the same data in multiple underlying sources. The transaction service should modify its system(s) of record only. Any required propagation of new or modified data to other systems should be done after the transaction is completed, and it should be performed by an appropriate infrastructure that is designed for this kind of pattern.

2.3.1 Transaction Services Infrastructure

Important considerations when choosing a transaction server development environment are a superset of the those when choosing an application server environment. In most cases, an enterprise will already have at least one in-house application server environment which IT is familiar with, and that same environment can probably be effectively used to create and deploy transaction services. Since application servers are well understood by most IT departments, the following list comprises only additional considerations that should influence the selection of an application server for building transaction services.

- **Service Standards Support.** The transaction environment should offer built-in support for XML manipulation, SOAP semantics, and automatic WSDL creation. In addition, it should be easy to implement services that conform to the WS-I Basic Profile for web services.

- **Vendor Neutrality.** Make sure the services that are created in the environment do not require software from the same vendor on the consumer side of the interaction. This is a key point in guaranteeing truly reusable and loosely coupled services.

- **Robust Transaction Semantics.** The environment should support various transaction implementation models, from two-phase commit to compensation models, and it should be easy for the software developer to wrap his work in a reliable transaction scope.

- **Easy and Efficient Service Invocation.** Transaction service developers will need to access data from the data services infrastructure, so it is important that service invocation be easy and efficient for the developer to accomplish. Otherwise, the developer will be tempted to access data directly, thereby compromising the abstraction provided by the data services layer.

- **Strong and Flexible Security Mechanisms.** Exactly what your enterprise needs will be determined by your requirements, but the software vendor should provide general purpose mechanisms to implement a variety of security measures. Later in this chapter there is a section that describes service security.

2.4 Service Interface Design

The web service standards and recommendations leave service creators with broad latitude for designing service interfaces. From one viewpoint, this is a very positive situation: You can design service interfaces that exactly meet the needs of your enterprise. From another viewpoint, however, this broad latitude creates a problem because it will be easy to inadvertently create service interfaces that have no relationship with one another, are difficult to use, and the resulting services will embody no unified design vision. In other words, it will be a mess.

You may be able to take service design guidance from the dominant packaged application vendor in your enterprise. Some of the application vendors have made significant progress in providing service-based APIs. SAP currently provides the most complete treatment, although it requires their installed base to upgrade to take full advantage of their offering. Other vendors have not made significant public commitments to service-based APIs, so it's not clear what direction they will take. If you are a customer of one of these vendors you should demand to see their plans so that you can begin your own planning. When you learn more about the APIs that your vendors provide, you can consider modeling your own APIs along the same lines, or wrap those APIs in your own to extend or elevate their interfaces. The guidelines below will help you determine whether the vendor-provided APIs are appropriate for your needs.

You should think of your service interfaces as the public API into your enterprise data. As such, care should be taken to make them useful, easy to learn, well documented, consistent, supportable, and extendable. If you have ever been on the consumer side of a poorly designed API, you can appreciate the need for simplicity and elegance – it should all hang together and make sense to the consumer.

Fortunately, we can learn something about how to do this from another software development paradigm: Object oriented programming. In this paradigm, a developer creates a class for a specific type of data, and the class implements methods (procedures) for manipulating that data. Related classes that work with each other to accomplish something broader are usually grouped into packages, and multiple packages that form a comprehensive framework are packaged and distributed together.

An analogous paradigm can be used as a guideline for developing your services.

- **Categorize Your Data.** Design a collection of services for manipulating a particular category of data. For example, *customer*. Services should be provided for creating, updating, deleting, and retrieving customers. There may be several services for each of these activities. For example, you might provide multiple ways to retrieve customers.

- **Group Services by Category.** Create these sets of services for each category of data in your enterprise. The categories of data will either be master data (e.g., employee, customer, and product) or operational data (e.g., order, PO, shipment). The collection of services might be slightly different for these two categories of data, but the differences should be motivated by requirements of your service consumers.

- **Judiciously Provide Cross-Category Services.** Where necessary, create services that operate on multiple categories of data, but leverage the service interfaces you designed for the individual data types. For example, you might need to provide a service that retrieves a customer and all of their orders. Make sure the input parameter to specify the customer matches the input parameter for specifying a customer in the collection of customer-specific services. Furthermore, make sure the schema of the returned data (customer and orders) match the schemas for customer and order data returned in the data-specific services.

- **Package Related Services Together.** Finally, group related services together in a single WSDL to provide consumers access to the whole framework at once.

The important thing is to avoid designing individual service interfaces in isolation. If consumers are familiar with your services for manipulating an employee, it should be natural and easy for them to begin using your services that manipulate a customer. It sounds like common sense, but it will make a huge difference in the adoption rate of your shared services.

2.4.1 Individual Service Design

With this service framework in mind, we can turn our attention to the design of the individual services, beginning with some guidelines, including:

- Keep interfaces as simple as possible. Service consumers do not want a comprehensive service that does everything possible on a particular kind of data, but requires an overly complex service call simply to, for example, change a phone number of a customer. Service consumers want it to be easy and obvious how to accomplish their task.

- A service that modifies data should either completely succeed or cleanly fail: Without leaving corrupted or incomplete data behind. Exactly how the service accomplishes this will depend on the implementation, but the consistency contract with the customer should not be compromised.

- Try not to provide services to consumers that would allow them to unwittingly do harm to enterprise data. For example, if you provide a service that sets the inventory level of a product, a service consumer could retrieve the existing inventory level, add some recently received product to the count, and then update the inventory to the new count. Unfortunately, if two different consumers perform this sequence at roughly the same time, it would be possible to lose inventory because one of the consumers can overwrite the other consumer's change. It is preferable to provide a service that increments or decrements inventory, and the service's implementation should employ a locking strategy to ensure correct and consistent behavior.

- Establish and use a standard error reporting scheme for all services (refer to the following section for more details).

2.4.2 Error Handling

Reporting errors that occur during service invocation should be done in a way that allows the client to handle errors in a consistent way. There are four kinds of errors that can occur during service invocation, including:

- **Communication Errors.** These happen when the service infrastructure is unavailable to complete the invocation (e.g., the network is down). These errors will usually manifest as something outside the SOAP standard (e.g., an HTTP connect error). As a service implementer, you don't have control over how to report them. You should, however, perform internal testing with your own infrastructure to see how errors will be reported to your consumers. This will enable you to provide direction for handling errors effectively in the service orchestration environment.

- **System Errors.** These occur inside the execution of the service, but they are related to the system rather than to the application logic. For example, temporary disk space for assembling results might become full, or a required data source is currently unavailable. These errors are usually not correctable by the caller. This class of errors should be reported to the caller as a *fault* in the SOAP invocation, with the standard fault code of *soap:Server*. SOAP faults are like exceptions, and they are returned to the caller instead of the return message. The caller can *catch* the SOAP fault and process it accordingly.

- **Application Errors.** These are errors in processing business logic that defines the service. For example, when a user attempts to set a phone number to an invalid string. Application errors should also be reported using SOAP faults, but with the standard fault code of *soap:Client,* which distinguishes them from system errors. It is useful to establish a convention for reporting additional information in the detail element of the fault. The WS-I Basic Profile for interoperability allows arbitrary sub-elements underneath the detail element so a schema snippet can be created and included in every SOAP fault. This will result in offering additional information that will be useful to the client (e.g., an application error code).

- **Application Warnings.** These are non-fatal errors that are discovered during the processing of the business logic. They are not severe enough to cause the request to fail, but you might like to tell the caller about them. For example, there might be a service that updates a customer's address, and the service caller might provide a zip code that does not match the city and state in the address. While it may be reasonable to allow this service to succeed (your own business rules will determine this), it will be useful to issue a warning that the customer's address data is not internally consistent. If you plan to issue warnings with your services you should create a standard part of the document schema for reporting them. All return messages should include the warning component as an optional part of the return message. The caller can choose to ignore it, but the information is available if they want to process the error.

2.4.3 Example

With these guidelines in mind, designing specific service interfaces required for a type of data can be straightforward. Here is a typical set of services you might create.

- Design an XML representation (schema) for the data.
- Design CRUD (Create, Retrieve, Update, Delete) service operations for the data (leveraging the XML schema).
- Design supplemental services to further manipulate the data, as required by the business.

To make these service development activities clearer, let's apply them to an example of customer data:

- **Design a XML representation (schema) for the type of data that the services will work with.** A customer represented in XML might begin something like this:

```
<customer>
    <id>123456</id>
    <creationTimestamp>2007-01-13 14:35:22.345</
    <creationTimestamp>
    <modificationTimestamp>2007-02-09 08:30:55.127</
    <modificationTimestamp>
```

```
    <firstName>John</firstName>
    <lastName>Smith</lastName>
    <gender>M</gender>
    <birthDate>1962-07-10</birthDate>
    …
</customer>
```

- **Create a service for creating a new customer.** The input document should be the schema designed above. The service should confirm that all required data elements have been provided. It is possible to specify required and optional elements in the XML schema, but different uses of the same schema will have different requirements, so it is better to embody this business logic in the service itself. The service should automatically create some of the fields for the consumer (e.g., the customer id should be uniquely generated, and the timestamps should be handled automatically). The service should return the complete customer (as created) using the same schema.

- **Create a service for easily modifying an existing customer.** The input document should be the customer schema designed above. The id element specifies which customer is to be updated and the other elements will be used to update (overwrite) the customer's data.

- **Provide a service for easily deleting a customer.** The input document for this service should simply contain the customer's id—no other data should be required. The service itself should implement all business logic required to delete a customer from the enterprise. For example, it may be desirable to remove a customer's orders, payments, and service calls as well. Whatever the business logic is, it should be performed in a manner that can guarantee integrity of the underlying data.

- **Provide a service for easily retrieving (querying) customers.** The input document should be the customer schema, and then only the provided fields will be used to match existing customers. For example, if an id is provided, a single customer will be matched. If a last name is provided, multiple customers may be matched. The service should return a list of customers that match the input values.

- **Provide additional services for retrieving customers in other useful ways, as dictated by the requirements of the consumers.** The input document should be designed to accommodate the necessary input data. The service should return a list of customers (using the same schema as all the other services). For example, somebody may want a service that retrieves all customers who placed orders since a given date (or between two dates).

- **Provide additional services for operating on customers, as dictated by the requirements of the consumers.** Again, the input document should be designed to accommodate the necessary input data. In this case, the output document should be designed to accommodate the service requirements. For example, somebody may want a service that counts

customers by geography, returning a list of countries, states, or zip codes, and the corresponding customer count for each geography. The important thing about designing these services is to wait until they are needed. Designing these services in the absence of real business requirements is usually time wasted.

This methodology can be applied repeatedly to all the data in your enterprise. It is not recommended that you do it all at once, however, because when consumers begin to use services you have provided, you will learn lessons that can be applied to future efforts. Expanding your service collection incrementally, as needed by the consumer community, is the most efficient way to proceed.

2.5 Security Considerations

Security of enterprise data is always a priority, and introducing services as an access and manipulation paradigm adds new challenges. Since each enterprise has its own philosophies on security, the best approach to service security is to extend your enterprise's current security strategies to these new paradigms. In keeping with that idea, this section is not a service security cookbook, but is provided to help educate the readers about the available security alternatives.

There are three areas of security to be considered when deploying services:

- User Authentication
- Access Authorization
- Message Privacy

Several standards exist that contribute to service security implementation (HTTP Authorization, WS-Security, SSL, SAML, etc). However, as with other web services standards, there is quite a bit of latitude, and therefore broad variability, as to how security is actually implemented. The WS-I has formulated the *WS-I Basic Security Profile* in an attempt to narrow the range, and increase both security and interoperability, and we urge readers to consult this recommendation to assist with security questions.

2.5.1 User Authentication

Services are essentially executable modules, available to other consumers over the network. But who gets to execute them? It is possible to provide services that are open to anyone, but this is not the usual situation in an enterprise. Rather, access to a service usually requires a user to be identified and authenticated so that authorization can be performed. With web services, this can be done in a number of ways:

- **HTTP Basic Authentication.** If your services are accessed over HTTP, your server can use HTTP basic authentication to require a user to provide a

username and password to essentially 'log in' to invoke the service. This is a simple but effective mechanism for authentication that is widely used. When combined with a wire-level encryption (i.e., SSL), it is quite secure. This kind of authentication mechanism is roughly equivalent to a normal client login to a database today.

- **SAML.** This is a standard XML-based authentication mechanism modeled on the presentation of a secured token. SAML is considered the future of web service authentication but it is not yet widely used. It is recommended that you use a service infrastructure provider that plans to support SAML within a year. The SAML model is similar to Kerberos, so if you currently use something similar to Kerberos for your enterprise authentication, you will be interested in learning about SAML for use with your service implementations.

- **Custom Login Service.** You can provide a custom service that accepts a user's login credentials and returns an identity token. The identity token is then presented as part of the input to each subsequent service invocation. This mechanism is widely used today, but it does not promote interoperability of services, and it requires all services to accommodate the mechanism. Combined with a wire-level encryption, however, it is quite secure. You can think of this approach as being equivalent to a login box on a web page portal where the web protocol is probably encrypted (HTTPS), but the actual authentication is processed by the application (which is probably running in an application server).

2.5.2 Access Authorization

Once a user is authenticated to the service infrastructure, there are two types of authorization to consider:

- Does the user have permission to invoke the service?
- Does the user have permission to access all of the data returned by the service?

The WS-I Basic Security Profile addresses both of these in detail, so we will not duplicate that effort in this book. However, some general considerations can be offered, including:

- Your service infrastructure should provide general purpose enforcement mechanisms for these. It should not be necessary to build authorization logic into the service implementation itself.

- If a user does not have permission to invoke a service, the simplest way to indicate this is to immediately return a SOAP fault.

- A service may return a rich XML document containing a significant amount of data, but the current user may be authorized to see only portion of the data. In this case, only the sections of the document for which the user

is authorized will be populated. Again, your service infrastructure should provide general purpose enforcement mechanisms for this type of security.

2.5.3 Message Privacy

Services operate using request and response messages, the contents of which are generally XML documents. When transported over an unsecured network, these request and response messages are potentially vulnerable to snooping, which dictates the need for message privacy strategies. There are two main mechanisms used to accomplish this today, SSL for HTTP and WS-Security.

SSL for HTTP

Most services today are accessed via HTTP. SSL can be used to provide a secure (encrypted) point-to-point communication channel between the consumer and the provider (HTTPS). This is the same mechanism used by your web browser when you submit your credit card information during a purchase. The advantage of this mechanism is that it's easy to implement and easy to use. Most secure web service calls are protected by this mechanism today. There are, however, two main disadvantages of this privacy mechanism:

- **Proxy Protection.** If the service call goes through a proxy, the secure communications channel does not extend through the proxy, potentially leaving the communications vulnerable. It is not always clear to the provider or the consumer exactly where proxies exist in the call chain, so care should be taken.

- **In-transit Protection.** The encryption exists only on the point-to-point communication channel, and does not secure the message itself. If the SOA architecture includes mechanisms for service mediation (e.g., store-and-forward), the message is unprotected when not being transported. Similarly, if messages are logged to a disk or database, the message is not secured.

WS-Security

This is a collection of security standards designed to secure web services. Its scope is actually broader than transport privacy (it can also be used to assist with authentication), but it is primarily aligned with message security. The WS-Security standards are not currently in wide use, but it is expected that they will be as SOA implementations proliferate. A comprehensive discussion of WS-Security is beyond the scope of this chapter, but the following is a summary of what it provides:

- **Element-level Message Encryption**. Specific sections of a service message (i.e., the XML document) can be encrypted for privacy. This encryption is within the message, so it persists for the life of the message—regardless of how or where the message travels.

- **Message Integrity**. Allows the consumer of the message to reliably determine whether the message has been modified since being created.

- **Message Authentication**. Reliably identifies and guarantees the sender/creator of the message.

Your service infrastructure vendor should provide support for WS-Security—it should be an important part of your vendor selection criteria.

Security is a broad and deep topic, and we have only scratched the surface in this section. The important point is that you can extend your current enterprise security strategies to embrace services as well. We recommend you formulate your enterprise's service security requirements, and then work with the service infrastructure vendors to put software in place that meets those requirements.

2.6 Conclusion

The collection of services you create will form the foundation for your service oriented architecture efforts. Your foundation's strength and longevity will be enhanced if you follow the suggestions outlined in this chapter.

You can begin creating services today. You do not need to wait until you have a comprehensive set of requirements, and you can get started with limited staff and investment. Select a project with specific well-known needs, and build the services needed to address those requirements.