SOA with Java

Realizing Service-Orientation with Java Technologies

Thomas Erl, Andre Tost,
Satadru Roy, and Philip Thomas

PRENTICE HALL

UPPER SADDLE RIVER, NJ • BOSTON • INDIANAPOLIS • SAN FRANCISCO

NEW YORK • TORONTO • MONTREAL • LONDON • MUNICH • PARIS • MADRID

CAPE TOWN • SYDNEY • TOKYO • SINGAPORE • MEXICO CITY

Contents at a Glance

Chapter 7

Service-Orientation Principles with Java Web-Based Services

Building services for service-oriented solutions requires the application of the service-orientation paradigm whose established design principles drive many Java service contract and implementation decisions. In certain cases, the programming language and runtime environment used for services can also be influenced by these guiding principles. This chapter visits each of the eight service-orientation principles in depth to highlight considerations specific to design and development with Java.

NOTE

The service-orientation principles are formally documented in the series title *SOA Principles of Service Design*. Concise profiles of the principles are also available in Appendix B and at www.serviceorientation.com.

7.1 Service Reusability

The following are common design characteristics associated with reusable services:

- The service is defined by an agnostic functional context.
- The service logic is highly generic.
- The service has a generic and extensible contract.
- The service logic can be accessed concurrently.

Let's take a closer look at each of these characteristics in relation to Java.

Agnostic Functional Contexts

Ensuring that the logic encapsulated by a service is agnostic to any particular functional context allows for the building of service interfaces independent of one particular business process or functional domain. The term "context" refers to the service's functional scope. An agnostic functional context is not specific to any one purpose and is therefore considered multi-purpose. A non-agnostic functional context, on the other hand, is intentionally single-purpose.

A checkpoint that can be part of a regular code review or service interface quality gate is to look at the imports in a Java interface or implementation class for a service. Java interfaces and classes are often structured according to the applicable business domain, and focusing on the list of imported packages can help identify dependencies in the code. Returning to the simplified order management example, the Java service interface for the Credit Check service is seen in Example 7.1.

```
import com.acme.businessobjects.om.Order;
public interface CreditCheck {
  public boolean hasGoodCredit(Order order);
}
```

Example 7.1

The `import` statement indicates that the service logic depends on the functional context of order management. Such a dependency cannot always be avoided if a service is developed specifically for a particular business domain at the cost of its reusability. Utility services are generally agnostic and reusable, as explained in Chapter 8.

Highly Generic Service Logic

Generic Java service logic refers to logic independent of its service contract. In the Java world, this means that a Java service interface is created with no mapping for the data types referenced in the service contract.

The `javax.xml.ws.Provider` interface avoids dependency on the service contract when using the JAX-WS programming model for SOAP-based Web services. An incoming message can be received by the service as a SAAJ `javax.xml.soap.SOAPMessage` with the `Provider` interface, which allows the entire message to be parsed or navigated as a DOM tree, as seen in Example 7.2.

```
@ServiceMode(value=Service.Mode.MESSAGE)
@WebServiceProvider()
public class GenericProviderImpl implements
  Provider<javax.xml.soap.SOAPMessage> {
  public SOAPMessage invoke(SOAPMessage message) {
    // read and process SOAPMessage...
  }
}
```

Example 7.2

For the same SOAP-based Web service, the request message can be alternatively read as a `javax.xml.transform.Source`. As shown in Example 7.3, the message can be treated as a plain XML document with no relationship to SOAP. Only the payload of the request message can be retrieved. Developers can ignore the SOAP envelope or SOAP header to focus on the content in the body of the message.

```
@ServiceMode(value=Service.Mode.PAYLOAD)
@WebServiceProvider()
public class GenericProviderImpl implements
  Provider<javax.xml.transform.Source> {
  public Source invoke(Source source) {
    // read and process SOAPMessage...
  }
}
```

Example 7.3

In both Examples 7.2 and 7.3, the response data returns in the same way the request data was received. If the request is received in an object of type `SOAPMessage`, then a new `SOAPMessage` object must be built for the response. Correspondingly, a new `Source` object must be returned if `Source` is used.

Generic Java types can capture the appropriate MIME media type or resource representation format produced or consumed by a target resource when building a REST service. For text-based request/response entities, Java `String`, `char[]`, and the character-based `java.io.Reader` or `Writer` interfaces can be used in the resource methods. For completely generic entity representations, which can include binary content, a `java.io.InputStream`, `OutputStream`, or a raw stream of bytes can be used as `byte[]`. For XML-based resource representations, the `javax.xml.transform.Source` type can be used to handle XML documents at a higher level than a raw stream.

As seen in Example 7.4, a slightly reworked customer resource example of the REST service from the Chapter 6 uses an `InputStream`. The contents of an entity are extracted in the incoming request to keep the service contract generic.

```
@Post
@Consumes("application/xml")
public void createCustomer(
  InputStream in){
  //extract customer from request
  Customer customer = extractCustomer(in);
  //create customer in system;
}
```

Example 7.4

Similarly, `javax.xml.transform.Source` can extract the customer information from the incoming request. JAX-RS relies on a bottom-up service design philosophy for building REST APIs except when using WADL. Using generic types, such as `java.lang.Object` or `byte[]`, on a JAX-RS resource interface should be sufficient to keep the service contract generic. However, consider what corresponding data types will be used in the WSDL for SOAP-based Web services.

Avoid the use of concrete data types on the Java service interface. The payload of a message is cast into a Java object, such as a `byte` array or `string`. The service contract, such as the WSDL for a Web service, must match the generic type, such as `java.lang.Object` maps to `xsd:anyType`, `byte[]`, which maps to `xsd:hexBinary`, and `java.lang.String` maps to `xsd:string`. The matching generic types require specific code to be developed in both the service consumer and service for the data to be inserted into the request/response messages.

In Example 7.5, the public class employs a `byte` array on its interface to hide the details of the data processed by the service.

```
@WebService
public class OrderProcessing {
  public void processOrder( Order order,
    byte[] additionalData) {
    // process request...
  }
}
```

Example 7.5

Supertypes in the service interface can aid in generalizing a service contract. For example, a service returns detailed account information for a bank's customer. When creating a data model for the different types of information provided by the different types of accounts, take advantage of inheritance in XML Schema. A superclass called `Account` can be created in Java, with a number of subclasses defined for each type of account, such as checking, savings, loans, and mortgages. A return type of `Account` which includes all of the different types of accounts can be specified in the service interface.

The considerations for supertypes are the same for both SOAP and REST services. As in both cases, the XML Java marshaling/unmarshaling is handled by JAXB for XML and MOXy or JSON-P for JSON. MOXy is a framework for marshaling/unmarshaling between JSON and Java objects. JSON-P (Java API for JSON Processing) supports low-level JSON parsing in Java EE 7.

A generic service implementation can serve multiple types of request/response messages, which generally increases reuse opportunities. However, the service logic must be implemented to handle different types of messages. Type-specific code uses language-specific data types. Tooling can generate code that automatically parses messages into the right Java objects, although at the cost of increased coupling. If generic types are used, the processing of incoming and outgoing messages is left to the service implementer. However, generic types offer greater flexibility in terms of type-independence and loose coupling.

Generalizing service logic also applies to the service consumer. For example, JAX-WS defines a generic service invocation API using the `javax.xml.ws Dispatch` interface. Services can be invoked with unknown interfaces when the service consumer code is developed. Similar to how the use of the `Provider` interface supports handling requests from different types of service consumers, the use of the Dispatch API allows a service consumer to interact with different types of services. For REST service clients, if a JAX-RS implementation is used, all the generic Java types the JAX-RS implementation supports can be used to build requests and consume responses. However, generic service logic requires the client code to handle different types of messages and have some knowledge about the message formats expected.

Generic and Extensible Service Contracts

Service logic can be made generic for reuse across a wide range of scenarios and adapted to changes in its environment, such as by changing and evolving services or service consumers. A service contract can be made generic by restricting the dependencies on data types referred to in the service contract and limiting the services composed inside the service logic to a minimum. When translated into Java-specific terms, reduce or eliminate the number of business-domain-specific classes in the service implementation.

Creating a generic service contract means applying generic types like `string`, `hexBinary`, or `anyType` in the schema type definitions for a Web service. Alternatively, message formats can be defined in a service contract with schema inheritance to use common supertypes, and the runtime allowed to determine which concrete subtype is used. Generic types are not only true for the top-level elements in a message but can also be used within a type definition. In Example 7.6, the schema describes a `Customer` type with a number of well-defined fields and a generic part.

```
<xs:complexType name="Customer">
  <xs:sequence>
    <xs:element name="accounts" type="ns:Account"
      nillable="true" maxOccurs="unbounded"
      minOccurs="0"/>
    <xs:element name="address" type="ns:Address"
      minOccurs="0"/>
    <xs:element name="customerId" type="xs:string"
      minOccurs="0"/>
    <xs:element name="name" type="ns:Name" minOccurs="0"/>
    <xs:element name="orderHistory" type="ns:OrderArray"
      nillable="true" maxOccurs="unbounded"
      minOccurs="0"/>
    <xs:any/>
  </xs:sequence>
</xs:complexType>
```

Example 7.6

When used in a service contract, the schema in Example 7.6 allows the service to process messages that have a fixed part at the beginning and a variable part at the end, which is represented by the `<xs:any>` element. SOAP-based Web services can use the Dispatch APIs in the service logic and Provider APIs in the service consumer logic without affecting how the service contract is built.

Service consumer logic can be implemented generically even if a detailed and specific service contract is provided. For REST services, the same considerations hold true for resource representations. XML-based representations can use highly specific types while the JAX-RS resource class can leverage generic Java types. The programmer then becomes responsible for performing the type mapping between XML and Java in the service logic.

Concurrent Access to Service Logic

A particular instance of a shared service will almost always be used by multiple service consumers simultaneously at runtime. How the service is deployed and the characteristics of the service's runtime environment as influences on service reuse are significant design considerations.

For example, each service request starts a new process that executes the service logic for the request, which ensures that the processing of one request does not affect the processing of another request. Each execution is completely independent of other executions and creates a new thread in the process. However, starting a new process is a relatively expensive operation in terms of system resources and execution time in most runtime environments. Sharing services in this way is inefficient.

All service requests executed within the same process share all the resources assigned to that process, which provides a lightweight method of serving multiple concurrent requests to the same service. Starting a new thread is an inexpensive operation on most systems. Most, if not all, Web service engines work this way. Implementing services in a multithreaded environment requires adherence to the basic rules of concurrent Java programming.

CASE STUDY EXAMPLE

As part of an initiative to comply with recently introduced legal obligations, the NovoBank IT team decides to create a Document Manager service that stores documents for auditing purposes. The service supports storing and retrieving XML documents. To maximize the reusability of this service, the NovoBank IT team creates a generic service contract and flexible service implementation to handle different types of documents, which can be extended over time.

Initially, the service will have the operations of `store` and `retrieve`. The service contract does not imply any structure of the documents stored, as shown in Example 7.7.

```
<definitions targetNamespace="http://utility.services.novobank.com/"
  name="DocumentManagerService" xmlns:tns="http://utility.services.
  novobank.com/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xs:schema version="1.0" targetNamespace="http://utility.
      services.novobank.com/" xmlns:xs="http://www.w3.org/2001/
      XMLSchema">
      <xs:element name="retrieve" type="ns1:retrieve"
        xmlns:ns1="http://utility.services.novobank.com/"/>
      <xs:complexType name="retrieve">
        <xs:sequence>
          <xs:element name="arg0" type="xs:string" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
      <xs:element name="retrieveResponse" type="ns2:retrieveResponse"
        xmlns:ns2="http://utility.services.novobank.com/"/>
      <xs:complexType name="retrieveResponse">
        <xs:sequence>
          <xs:any processContents="skip"/>
        </xs:sequence>
      </xs:complexType>
      <xs:element name="store" type="ns3:store"
        xmlns:ns3="http://utility.services.novobank.com/"/>
      <xs:complexType name="store">
        <xs:sequence>
          <xs:any processContents="skip"/>
        </xs:sequence>
      </xs:complexType>
      <xs:element name="storeResponse" type="ns4:storeResponse"
        xmlns:ns4="http://utility.services.novobank.com/"/>
      <xs:complexType name="storeResponse"/>
    </xs:schema>
  </types>
  <message name="store">
    <part name="parameters" element="tns:store"/>
  </message>
  <message name="storeResponse">
    <part name="parameters" element="tns:storeResponse"/>
  </message>
  <message name="retrieve">
    <part name="parameters" element="tns:retrieve"/>
  </message>
  <message name="retrieveResponse">
    <part name="parameters" element="tns:retrieveResponse"/>
  </message>
  <portType name="DocumentManager">
    <operation name="store">
      <input message="tns:store"/>
      <output message="tns:storeResponse"/>
    </operation>
    <operation name="retrieve">
      <input message="tns:retrieve"/>
      <output message="tns:retrieveResponse"/>
    </operation>
  </portType>
  <binding name="DocumentManagerPortBinding"
    type="tns:DocumentManager">
    ...
  </binding>
  <service name="DocumentManagerService">
    ...
  </service>
</definitions>
```

Example 7.7
NovoBank's service contract for the Document Manager service is deliberately generic.

Example 7.7 shows that the content of the stored messages is represented by an `<xs:any/>` element, which means that any well-formed XML can be inserted in the content. The `<xs: any/>` element allows the Document Manager service to be reused across many different types of messages and documents.

The development team decides to create a flexible implementation of the service which is independent of the specific type of document being sent or retrieved. The service must extend without affecting the existing implementation, preparing support for more specific processing of specific document types. A flexible implementation is achieved with a handler and factory. The implementation leverages the `javax.xml.ws.Provider` interface and delegates the processing of each message to a handler. The handler instance is then retrieved via a factory.

Example 7.8 shows the implementation class for the Document Manager service in the detailed source code.

```
package com.novobank.services.utility;
import javax.xml.transform.Source;
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@ServiceMode(value=Service.Mode.PAYLOAD)
@WebServiceProvider
public class DocumentManager implements Provider<Source> {
  public Source invoke(Source source) {
    DocumentHandler handler =
      DocumentHandlerFactory.instance().getHandler(source);
    handler.process(source);
    return null;
  }
}
```

Example 7.8
The implementation of the Document Manager service utilizes a factory to retrieve a handler.

This class implements the `Provider` interface, so that any invocation of the service is directed to the `invoke()` method. The `@ServiceMode` annotation indicates that only the actual content of the SOAP body should be passed into the implementation.

In the implementation of the `invoke()` method, a single instance of a class called `DocumentHandlerFactory` is used to retrieve a `DocumentHandler` for this message. Example 7.9 shows the source code for the `DocumentHandler` interface.

```
package com.novobank.services.utility;
import javax.xml.transform.Source;
public interface DocumentHandler {
  public void process(Source source);
}
```

Example 7.9
The source code for the `DocumentHandler` interface

The `DocumentHandler` interface only defines the `process()` method, which processes the message. Different implementations of the interface that process messages in different ways can exist. The message content is passed into the `process()` method as a stream of type `javax.xml.transform.Source`. When revisiting the source code for the Document Manager service in Example 7.2, the message returned by the factory can be seen to be passed to the handler.

A part of the source code for the factory is presented in Example 7.10.

```
package com.novobank.services.utility;
import javax.xml.transform.Source;
public class DocumentHandlerFactory {
  protected static DocumentHandlerFactory theInstance = new
    DocumentHandlerFactory();
  public static DocumentHandlerFactory instance() {
    return theInstance;
  }
  protected DocumentHandlerFactory() {}
  public DocumentHandler getHandler(Source source) {
    DocumentHandler handler = null;
    // the code where the message is parsed and
    // the appropriate handler is retrieved would be here.
    return handler;
    }
}
```

Example 7.10
The DocumentHandlerFactory source code

The message is parsed to the point where an appropriate handler can be found. Handlers are registered via several mechanisms that can include hardcoding into the factory class, retrieval from a file, or lookup in a registry. The concrete `DocumentHandler` implementation chosen for a particular message is often based on the root element of the passed message. The root element typically provides adequate indication of the nature of the message.

Using the factory mechanism, where the service implementation class calls a factory to retrieve a handler to process the message, allows new handlers to be later added without affecting the existing code. The combination of a generic service definition using the `<xs:any/>` element, a flexible service implementation using the `Provider` interface, and a factory to delegate the processing of the message ensures maximum reusability of the service across a variety of environments and domains.

SUMMARY OF KEY POINTS

- For SOAP-based Web services, the JAX-WS standard offers ways of implementing service logic as generic and therefore capable of handling different types of messages.
- For REST services, Java generic types can be used in JAX-RS-based resource implementations. The request/response entities are treated as a raw sequence of bytes or characters.
- Using generic data types for domain entities or resource representations allows the service to be reused across a greater number of potential service consumers.

7.2 Standardized Service Contract

A foundational criterion in service-orientation is that a service have a well-defined and standardized contract. When building Web services with SOAP and WS-*, portable machine-readable service contracts are mandatory between different platforms as WSDL documents and associated XML schema artifacts describing the service data model. The finite set of widely used HTTP verbs for REST services form an implicit service contract. However, describing the entity representations for capture in a portable and machine-independent format is the same as SOAP and WS-*.

For REST services, capturing and communicating various aspects of resources can be necessary, such as the set of resources, relationships between resources, HTTP verbs allowed on resources, and supported resource representation formats. Standards, such as WADL, can be used to satisfy the mandatory requirements. Having a standards-based service contract exist separate from the service logic, with service data entities described in a platform-neutral and technology-neutral format, constitutes a service by common definition. Even the self-describing contract of HTTP verbs for a REST service establishes a standards-based service contract. Recall the standards used for service contracts, such as WSDL/WADL and XML Schema, from Chapter 5.

Top-Down vs. Bottom-Up

Ensuring that services are business-aligned and not strictly IT-driven is necessary when identifying services for a service portfolio in an SOA. Services are derived from a decomposition of a company's core business processes and a collection of key business entities. For a top-down approach, services are identified and interfaces are designed by creating appropriate schema artifacts to model either the operating data and WSDL-based service contracts, or model REST resources and resource methods. The completed service interface is implemented in code.

However, enterprises can have irreplaceable mission-critical applications in place. Therefore, another aspect of finding services is assessing existing applications and components to be refactored as services for a bottom-up approach. This includes creating standard service contracts, such as WSDL definitions or REST resource models, for the existing components.

Tooling provides support for both approaches in a Java world. For SOAP-based Web services, tools play a more prominent role than in Java-based REST services. JAX-WS defines the wsimport tool, which takes an existing WSDL definition as input to generate Java skeletons. These skeletons can be used as the starting point for implementing the actual service logic. Similarly, the wsgen tool generates WSDL from existing Java code. The mapping between WSDL/XML schema and Java is an important function associated with the wsimport tool.

Machine-readable contracts are also necessary for REST services. JAX-RS, if WADL is not used, starts with a resource model to implement the resources in Java. Consider the contract as a logical collection of the resource model, with the supported resource methods, resource representations, and any hyperlinks embedded in the representations allowing navigability between resources. If WADL is used, tools like wadl2java can generate code artifacts. Initiatives exist to help generate WADL from annotated JAX-RS classes for a bottom-up approach, although these recent developments can have limited usefulness.

Some SOA projects will employ both a bottom-up and a top-down approach to identify and design services and service contracts, which often results in a meet-in-the-middle approach. Service definitions and Java interfaces are tuned and adjusted until a good match is found.

Sometimes an XML schema definition developed as part of the service design cannot map well into Java code. Conversely, existing Java code may not easily map into an XML schema. Java code that does not precisely map to a service interface designed as part of a top-down approach can exist. In this case, the Service Façade pattern can be applied to insert a thin service wrapper to satisfy the service interface and adapt incoming and outgoing data to the format supported by the existing Java code.

Mapping Between Java and WSDL

WSDL is the dominant method of expressing the contract of a Java component. While typically related to Web services, the language can also be utilized for other types of services. Formalization and standardization of the relationship between Java and WSDL has made this possible, such as the work completed on the JAX-RPC standard.

The JAX-RPC standard initially defined basic tasks, such as "a service portType is mapped to a Java interface" and "an operation is mapped to a method." However, formalizing allows the definitions described in the JAX-RPC standard to define how an existing Java component (a class or interface) can generate a WSDL definition, and vice versa. Consequently, most contemporary Java IDEs support generating one from the other without requiring any manual work.

JAX-WS, the successor standard for JAX-RPC, builds on top of its predecessor's definitions and delegates all the issues of mapping between Java and XML to the JAXB specification (as discussed in Chapter 6). These sections serve to highlight some of the issues raised when creating standard service contracts from Java or creating Java skeletons from existing service contracts. The majority of the details explained in the next section apply specifically to Web services.

Wrapped Document/Literal Contracts

The WSDL standard identifies a variety of styles for transmitting information between a service consumer and service. Most of the styles are specific for the chosen message and network protocol, and specified in a section of the WSDL definition called the binding. A common binding found in a WSDL definition uses SOAP as the message protocol and HTTP as the network transport. Assume that SOAP/HTTP is the protocol used for the services presented as examples.

The portType is a binding-neutral part of a service definition in WSDL that describes the messages that travel in and out of a service. Reusable across multiple protocols, the portType is not bound to the use of a Web service. Any service, even if invoked locally, can be described by a WSDL portType, which allows service interfaces to be defined in a language-neutral fashion regardless of whether the service logic will be implemented in Java or another language.

As discussed in Chapter 5, the WSDL binding information defines the message format and protocol details for Web services. For SOAP-based bindings, two key attributes known as the encoding style and the invocation style determine how messages are encoded and how services are invoked.

The wrapped document/literal style supported by default in all Java environments for services dictates that an exchange should be literal. Literal means that no encoding happens in the message, so the payload of the message is a literal instantiation of the schema descriptions in the `<types>` element of the WSDL. The invocation style is document. Document means that the runtime environment should generate a direct copy of the input and output messages as defined in the portType and not just an arbitrary part of the message. Wrapped means that the payload of the message includes a wrapper element with the same name as the operation invoked.

In order to understand how the WSDL standard relates to Java, let's review Example 7.11 to expose the following class as a service and create a standardized contract.

```
package pack;
import javax.jws.*;
@WebService
public class Echo {
  public String echo(String msg) {
    return msg;
  }
}
```

Example 7.11

Using the wrapped document/literal style implements a wrapper element called `"echo"` after the `echo()` method in the public class. Echo is included in the XML schema associated with this service. An excerpt in the resulting schema is provided in Example 7.12.

```
...
  <xs:element name="echo">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="arg0" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  </xs:element>
...
```

Example 7.12

Wrapping a message in one additional element named after the operation is prevalent and the default in any commonly used tool. Note that naming the global element after the operation is common practice and not required by the specification.

Implicit and Explicit Headers

Transferring information as part of the `<Header>` portion of the SOAP message, to be added to the WSDL definition, is another important part of the binding information for SOAP. This section discusses how to bind the information with explicit, implicit, or no headers.

Explicit Headers

Header data is part of the messages referenced in the portType of the service, which is often called an explicit header. The header definition in the SOAP binding refers to a message part either included in the input message or the output message of an operation.

In Example 7.13, assume an Echo service takes a `string` as input and returns that `string` as the response. A `timestamp` must also be added into the header of the SOAP request message, indicating the time at which the request was sent.

```
<definitions targetNamespace="http://pack/" name="EchoService"
  xmlns:tns="http://pack/" xmlns:xs="http://www.w3.org/2001/
  XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xs:schema targetNamespace="http://pack/">
      <xs:element name="echo">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="arg0" type="xs:string" minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="echoResponse">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="return" type="xs:string" minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="timestamp" type="xs:dateTime"/>
    </xs:schema>
  </types>
  <message name="echo">
    <part name="parameters" element="tns:echo"/>
    <part name="timestamp" element="tns:timestamp"/>
  </message>
  <message name="echoResponse">
    <part name="parameters" element="tns:echoResponse"/>
  </message>
  <portType name="Echo">
    <operation name="echo">
      <input message="tns:echo"/>
      <output message="tns:echoResponse"/>
    </operation>
  </portType>
  <binding name="EchoPortBinding" type="tns:Echo">
    <soap:binding transport="http://schemas.xmlsoap.org/ soap/http"
      style="document"/>
    <operation name="echo">
      <soap:operation soapAction=""/>
      <input>
        <soap:body parts="parameters" use="literal"/>
        <soap:header message="tns:echo" part="timestamp"
          use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
...
</definitions>
```

Example 7.13

The Echo service WSDL definition with an explicit header contains a timestamp.

Example 7.13 contains an extract of the respective WSDL definition for an Echo service that shows:

- an additional element in the schema, called `"timestamp"` of type `xs:dateTime`
- an additional part in the input message definition, which refers to the `timestamp` element
- an additional definition for the header in the SOAP binding, which indicates that the `timestamp` element should be carried in the SOAP header of the request message

The header binding shown in Example 7.14 refers to a part also included in the portType of the service, the input message, and the Java service interface generated by the JAX-WS wsimport tool. Note that the import statements are left out.

```
@WebService(name = "Echo", targetNamespace = "http://pack/")
@SOAPBinding(parameterStyle = ParameterStyle.BARE)
public interface Echo {
  @WebMethod
  @WebResult(name = "echoResponse", targetNamespace = "http://pack/",
    partName = "parameters")
  public EchoResponse echo(
    @WebParam(name = "echo", targetNamespace = "http://pack/",
      partName = "parameters")
    Echo_Type parameters,
      @WebParam(name = "timestamp", targetNamespace = "http://pack/",
      header = true, partName = "timestamp")
    XMLGregorianCalendar timestamp);
}
```

Example 7.14

The service interface includes a parameter for the explicit header and indicates two parameters: one that contains the string wrapped into the `Echo_Type` class and another that carries the `timestamp` element. Note that nothing in the interface indicates that the `timestamp` will go into the SOAP header, as this information is only contained in the WSDL definition.

Implicit Headers

Assume that the header data is not part of the portType but instead uses a message part unused in any input or output message, known as an implicit header. The header information is not included in the portType of the service or in the Java interface. Example 7.15 shows that the WSDL for the Echo service has been changed to include an explicit header.

```
<definitions targetNamespace="http://pack/" name="EchoService"
  xmlns:tns="http://pack/" xmlns:xs="http://www.w3.org/2001/
  XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  ... (this part is as before) ...
  <message name="echo">
    <part name="parameters" element="tns:echo"/>
  </message>
  <message name="echoResponse">
    <part name="parameters" element="tns:echoResponse"/>
  </message>
  <message name="header">
    <part name="timestamp" element="tns:timestamp"/>
  </message>
  <portType name="Echo">
    <operation name="echo">
      <input message="tns:echo"/>
      <output message="tns:echoResponse"/>
    </operation>
  </portType>
  <binding name="EchoPortBinding" type="tns:Echo">
    <soap:binding transport="http://schemas.xmlsoap.org/ soap/http"
      style="document"/>
    <operation name="echo">
      <soap:operation soapAction=""/>
      <input>
        <soap:body parts="parameters" use="literal"/>
        <soap:header message="tns:header" part="timestamp"
          use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  ...
</definitions>
```

Example 7.15
A WSDL definition contains an implicit header for an Echo service.

The WSDL definition presented in Example 7.15 is not that much different from Example 7.13, with a separate message defined for the header. The separate message has a significant impact on the Java interface seen in Example 7.16, where the import statements have been omitted again.

```
@WebService(name = "Echo", targetNamespace = "http://pack/")
public interface Echo {
  @WebMethod
  @WebResult(targetNamespace = "")
  @RequestWrapper(localName = "echo",
    targetNamespace = "http://pack/", className = "pack.Echo_Type")
  @ResponseWrapper(localName = "echoResponse",
    targetNamespace = "http://pack/", className = "pack.EchoResponse")
    public String echo(
      @WebParam(name = "arg0", targetNamespace = "")
      String arg0);
}
```

Example 7.16
The service interface does not include the implicit header.

The interface in Example 7.16 takes a simple `string` parameter, and does not refer to the `timestamp` element or use the `Echo_Type` class to wrap the input message. The implicit header definition requires extra work on the service implementer by the service client developer to ensure the appropriate header information is inserted into the message. The implicit header definition cannot simply be passed to the service proxy as a parameter. In both cases, JAX-WS handlers can be leveraged to process the SOAP header, or intermediaries inserted between service consumer and service can manage all header information, such as part of an ESB.

The header portion of a service message should only contain contextual information, omitting any business connotation. The implementations of the service consumer and the service should only deal with business logic and not with infrastructure-level information. The use of implicit headers is common, although extra code to generate and process the headers must be written.

No Headers

A final option is to put no header information in the WSDL definition, which can appear to leave the contract incomplete but is actually preferable. Headers typically contain information independent from the business payload being exchanged between services. Recall a timestamp that had been inserted into the SOAP message presented in Examples 7.13 and 7.15. Inserting a timestamp might be a defined company policy across all services, and a common method for doing so can be established. Adding this detail to each WSDL definition is not required and creates an unnecessary dependency between the business-relevant service contract and technical cross-service policy.

Data Mapping with REST

XML schemas can be used to represent service data elements, with JAXB and JAX-WS generating the mapped Java classes and Web service artifacts for SOAP-style Web service implementations. For REST services, the JAX-RS service implementations are similar. When the convenience of code generation is needed, JAXB annotated POJOs can be used as the service entities in JAX-RS resource classes. Behind the scenes, the JAX-RS runtime will marshal the Java objects to the appropriate MIME-type representations for the entities, such as `application/xml`. A `customer` object annotated with JAXB annotations is shown in Example 7.17.

```
@XmlRootElement(name="customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
  private int id;
  private String name;

  public Customer() {}
  //...other attributes omitted
  //...getters and setters for id and name
  //...
}
```

Example 7.17

The resource methods in the JAX-RS implementation that produce or consume customer information in the form of XML can be seen in Example 7.18.

```
//...retrieve customer and return xml representation
@Get
@Path("id")
@Produces("application/xml")
public Customer getCustomer(
  @PathParam("id") Long id){
  Customer cust = findCustomer(id);
  return cust;
}
//...create customer with an xml input
@POST
@Consumes("application/xml")
public void createCustomer(
  Customer cust){
  //...create customer in the system
  Customer customer = createCustomer(cust);
  //...
}
```

Example 7.18

The JAX-RS implementation automatically handles the marshaling and unmarshaling of the XML-based resource representation to and from JAXB objects. Generic types, such as a `javax.xml.transform.Source`, can be handled in the resource class to keep the resource class independent of any specific types defined in the domain, such as a `customer` type. However, extra work is required to handle the extraction of the customer information from the `Source` object seen in Example 7.19.

```
@PUT
@Path("id")
@Consumes("application/xml")
public void updateCustomer(@PathParam("id") Long id,
  javax.xml.transform.Source cust) {
  // do all the hard work to
  // extract customer info in
  // extractCustomer()
  updateCustomer(id, extractCustomer(cust));
}
```

Example 7.19

JAX-RS supports alternate MIME types, such as JSON. Just as JAXB handles the binding of XML to and from Java objects, numerous frameworks exist that handle mapping JSON representations to Java objects and vice versa. Some commonly used frameworks for mapping between JSON and Java are MOXy, Jackson, and Jettison.

In the Jersey implementation of JAX-RS 2.0, the default mechanism for binding JSON data to Java objects leverages the MOXy framework. When the Jersey runtime is configured to use MOXy, the runtime will automatically perform binding between Java objects (POJOs or JAXB-based) and a JSON representation. Jackson or Jettison can also perform similar binding with appropriate runtime configuration. A low-level JSON parsing approach can be achieved with the newly introduced JSON-P API (Java API for JSON Processing) in Java EE 7. JSON-P should not be confused with JSONP (JSON with Padding), which is a JavaScript communication technique used to avoid certain types of browser restrictions.

Conversion Between JSON and POJOs

Given the same customer representation as illustrated in Example 7.20, no special code is required to handle an incoming JSON document or return a JSON-based representation.

```
//...
@GET
@Path("id")
@Produces("application/json")
public Customer getCustomer(
@PathParam("id") Long id){
  return findCustomer(id);
}
```

Example 7.20

The returned customer representation would be a JSON string, such as `{"name":"John Doe","id":"1234" ... }`.

The same JAXB objects can be used for handling JSON media types that would normally be used for XML representation. The addition of another MIME type in the `@Produces` can be seen in Example 7.21.

```
@GET
@Path("id")
@Produces("application/json", "application/xml")
```

```
public Customer getCustomer(
//...
```

Example 7.21

The JAX-RS runtime returns an appropriate representation (XML or JSON) that is determined by the client's preference. In spite of the convenience offered by JSON binding frameworks like MOXy or Jackson, greater control over the processing of the JSON input and output can be a requirement, as opposed to letting the JAX-RS runtime perform an automatic binding between JSON and Java objects.

For example, REST service operations must consume or produce only selective parts of large JSON documents, as converting the whole JSON document to a complete Java object graph can cause significant resource overheads. In such cases, a JSON parsing mechanism based on a streaming approach can be more suitable. JSON-P APIs allow a developer complete control over how JSON documents are processed. JSON-P supports two programming models, the JSON-P object model and the JSON-P streaming model.

The JSON-P object model creates a tree of Java objects representing a JSON document. The `JsonObject` class offers methods to add objects, arrays, and other primitive attributes to build a JSON document, while a `JsonValue` class allows attributes to be extracted from the Java object representing the JSON document. Despite the advantage of convenience, processing large documents with the object model can create substantial memory overheads, as maintaining a large tree of Java objects imposes significant demands on the Java heap memory. This API is similar to the Java DOM API for XML parsing (`javax.xml.parsers.DocumentBuilder`).

In comparison, the JSON-P streaming model uses an event parser that reads or writes JSON data one element at a time. The `JsonParser` can read a JSON document as a stream containing a sequence of events, offer hooks for intercepting events, and perform appropriate actions. The streaming model helps avoid reading the entire document into memory and offers substantial performance benefits. The API is similar to the StAX Iterator APIs for processing XML documents in a streaming fashion (`javax.xml.stream.XMLEventReader`). The `JsonGenerator` class is used to write JSON documents in a streaming fashion similar to `javax.xml.stream.XMLEventWriter` in StAX API.

JSON-P does not offer binding between JSON and Java. Frameworks, such as MOXy or Jackson, are similar to JAXB in how they will need to be leveraged to perform conversion between Java objects and JSON.

CASE STUDY EXAMPLE

SmartCredit launches an aggressive expansion campaign with the intention of offering premium credit cards with cashback offers to high-value customer prospects across a retail chain's locations. After signing an agreement with the retail chain, SmartCredit obtains prospect data from all the retail stores containing prospect names, e-mail addresses, and net transaction values at the end of every month. An internal Prospect Analyzer application will process the data to target prospects with high monthly transaction values and send out e-mails with new premium credit card offers.

The retail chain's IT department sends customer data to SmartCredit in large JSON documents containing prospect information. However, the SmartCredit Prospect Analyzer service is only interested in prospects that spend in excess of 2,000 dollars during the month. A fragment of a typical monthly JSON extract from the retail stores is provided in Example 7.22.

```
"txnsummary":{
"date":"2014-01-31T23:30:00-0800",
"store":"Fashion Trends #132",
"txn": [
  {
    "type":"cash",
    "amount":235.50,
    "e-mail":null
 },
 {
    "type":"credit",
    "amount":3565.00,
    "e-mail":"jane@doe.com"
  }
]}
```

Example 7.22

SmartCredit IT accepts the prospect and transaction data through an HTTP POST from the retail stores at the end of every month. A REST API that consumes this JSON data and extracts the prospects for marketing campaigns is built. After reviewing the size of the monthly feed, a SmartCredit architect quickly realizes that memory limitations will prevent a typical JSON-Java binding approach from working for such large payloads. In addition, SmartCredit is only interested in processing selective parts of the payload, such as credit card transactions with amounts greater than 2,000 dollars.

Converting the entire JSON data into Java objects is a waste of time, memory, and resources. The JSON-P streaming API is a suitable option for allowing selective processing of only the data sections meeting the given criteria. A simplified version of the final resource class is illustrated in Example 7.23.

```
import javax.ws.rs.Consumes;
import javax.ws.rs.Path;
import javax.ws.rs.POST;
import javax.ws.rs.core.MediaType;
import java.io.Reader;
import javax.json.Json;
import javax.json.streaming.JsonParser;
import javax.json.streaming.JsonParser.Event
@Path("upload")
@Consumes(MediaType.APPLICATION_JSON)
public class ProspectFilteringResource {
  private final double THRESHOLD = 2000.00;
  @POST
  public void filter(final Reader transactions) {
    JsonParser parser = Json.createParser(transactions);
    Event event = null;
    while(parser.hasNext()) {
      event = parser.next();
      if(event == Event.KEY_NAME&&"type".equals(parser.getString()))
      {
        event = parser.next(); //advance to Event.VALUE_STRING for
          the actual value of "type"
        if("credit".equals(parser.getString()) {
          parser.next(); //Event.KEY_NAME for "amount"
          event = parser.next(); //Event.VALUE_NUMBER for amount
          value
          if(parser.getBigDecimal().doubleValue() > THRESHOLD) {
            parser.next(); //advance to Event.KEY_NAME for "e-mail"
            parser.next(); //advance to Event.VALUE_STRING for
            e-mail info
            String e-mail = parser.getString();
            addToCampaignList(e-mail);
          }
        }
      }
    }
  }
  private void addToCampaignList(String e-mail) {
    // actual logic of adding e-mail to campaign list
  }
}
```

Example 7.23

The JSON-P streaming API can parse a large JSON document selectively.

The code uses the streaming API to advance the parser to consume only specific events and avoids reading the entire JSON data structure into memory, which would have been the case using the standard JSON-Java binding approach. One drawback to the JSON-P approach is the cumbersome maneuvering of the event stream in the application code, although such trade-offs are often necessary in real-world usage.

Binary Data in Web Services

Candidates looking to utilize a service-oriented solution often require the transfer of large amounts of data kept in some binary format, such as a JPEG image file. The Java language offers support for handling binary data in its JavaBeans Activation Framework (JAF) as well as classes and interfaces in other packages, which depend on the format. For example, the `java.awt.Image` class hierarchy supports image formats, such as JPEG. The JAXB specification defines how to map certain Java types to XML schema, such as the mapping of a `java.awt.Image` type to `base64Binary`.

Binary formats without a special Java type associated can use the `javax.activation.DataHandler` class defined in JAF. However, `byte[]` is the most generic way of representing binary data in Java. JAXB defines that a `byte[]` is mapped to `base64Binary` and `hexBinary`.

When generating a WSDL contract from a Java interface, binary data types are mapped using JAXB mapping rules. Generating the reverse is not as straightforward. An XML schema element of type `base64Binary` can map into multiple different Java types. By default, `byte[]` is used. Indicating that an element declared as `base64Binary` in the schema should be mapped to `java.awt.Image` in the Java service implementation can be performed in a number of ways. Binary data can be transferred in a SOAP message or inline, which means binary data is encoded into text and sent like any other data in the message.

Transferring the data inline maintains interoperability between different vendor environments, but is ineffective when dealing with large pieces of binary data, such as a CAD drawing of an airplane. The text encoding increases the size of the message by an average of 25%. The MTOM is an alternative to text encoding, which describes how parts of a message can be transferred in separate parts of a MIME-encoded multipart message. The binary content is removed from the SOAP payload, given a MIME type, and transferred separately.

JAXB provides support for MTOM, which must be explicitly enabled for the JAX-WS runtime. JAX-WS plugs into this support when building and parsing messages with attachments. An element definition can be annotated in an XML schema document with two specific attributes indicating which MIME type to give the element. When using MTOM, the `contentType` and `expectedContentTypes` attributes demonstrate how to MIME-encode the element and determine which Java type the element is mapped to in the Java service interface.

Nothing in the schema or in any of the Java code indicates whether the binary data is transferred as an attachment using MTOM or inserted directly into the SOAP envelope. In JAX-WS, distinguishing the difference is defined either by a setting on the service configuration file or programmatically. The following case study example illustrates a SOAP-based Web service managing attachments via MTOM.

CASE STUDY EXAMPLE

NovoBank offers a remote method of opening an account from its Web site for customers to download a form, fill it out, and mail or fax it to a branch office. Alternatively, customers can fill out the form in branch and provide the completed form to a bank employee. The forms are scanned in at the branch office for further processing by NovoBank’s back-end system before being archived.

To reduce processing time, the bank wants to offer customers and employees a Web application that accepts an uploaded binary image of the form to send to the bank. Internally, the Open Account service uses a Web service that takes the binary image as a parameter. To simplify the implementation of the associated service logic in Java, the service contract uses the `expectedContentType` attribute in its schema to indicate that the scanned image is formatted as a JPEG document. The resulting WSDL definition is shown in Example 7.24 with parts of the WSDL document omitted for brevity.

```
<definitions targetNamespace= "http://personalbanking.services.
  novobank.com/" name="AccountService"
  xmlns:tns="http://personal banking.services.novobank.com/"
  xmlns:xsd="http://www.w3.org/ 2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns=http://schemas.xmlsoap.org/wsdl/
  xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
  <types>
    <xsd:schema>
      <xs:element name="openAccount" type="ns1:openAccount"
        xmlns:ns1="http://personalbanking.services.novobank.com/"/>
      <xsd:complexType name="openAccount">
        <xsd:sequence>
          <xsd:element name="arg0" type="xsd:base64Binary"
            xmime:expectedContetTypes="image/jpeg" minOccurs="0"/>
        </xsd:sequence>
      </xsd:complexType>
      ...
    </xsd:schema>
  </types>
  <message name="openAccount">
    <part name="parameters" element="tns:openAccount"/>
  </message>
  <message name="openAccountResponse">
    <part name="parameters" element="tns:openAccountResponse"/>
  </message>
  <portType name="Account">
    <operation name="openAccount">
      <input message="tns:openAccount"/>
      <output message="tns:openAccountResponse"/>
    </operation>
  </portType>
</definitions>
```

Example 7.24

The WSDL for NovoBank's Open Account service with an MTOM content type definition will now accept the JPEG format.

An element of type `base64Binary` will be mapped to a `byte[]` in the Java interface. However, the additional annotation of the `parameter` element, using the `expectedContentTypes` attribute, leads to the following Java interface presented in Example 7.25.

```
package com.novobank.services.personalbanking;
import java.awt.Image;
// other imports omitted
@WebService(name = "Account", targetNamespace = "http://
  personalbanking.services.novobank.com/")
public interface Account {

  @WebMethod
  @WebResult(targetNamespace = "")
  @RequestWrapper(localName = "openAccount",
    targetNamespace = "http://personalbanking.services.novobank.
    com/", className = "com.novobank.services.personalbanking.
    OpenAccount")
  @ResponseWrapper(localName = "openAccountResponse",
    targetNamespace = "http://personalbanking.services.novobank.
    com/", className = "com.novobank.services.personal banking.
    OpenAccountResponse")
  public String openAccount(
  @WebParam(name = "arg0", targetNamespace = "")
    Image arg0);
}
```

Example 7.25
The content type is mapped to the appropriate Java type in the service interface.

Note how the `parameter` passed to the service implementation is mapped to the `java.awt.Image` type. Defining whether MTOM is used to transfer the form image as an attachment can be performed programmatically using JAX-WS, or in the client or the endpoint configuration for the service endpoint. A sample client for the new Open Account service is shown in Example 7.26.

```
package com.novobank.services.personalbanking.client;
import java.awt.Image;
import java.awt.Toolkit;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.soap.SOAPBinding;

public class AccountServiceClient {
  public static void main(String[] args) {
    Image image = Toolkit.getDefaultToolkit().getImage("c:\\temp\\
      java.jpg");
    Account serviceProxy = new AccountService(). getAccountPort();
    SOAPBinding binding = (SOAPBinding)((BindingProvider)
      serviceProxy).getBinding();
    binding.setMTOMEnabled(true);
    String accountNumber = serviceProxy.openAccount(image);
    System.out.println("returned account number is
      "+accountNumber);
  }
}
```

Example 7.26
A Java client enables MTOM transport.

After deploying the service and running the test client listed in Example 7.26, the SOAP message sent can be reviewed by executing the client in Example 7.27.

```
POST /attachment/account HTTP/1.1
Content-Length: 13897
SOAPAction: ""
Content-Type: Multipart/Related; type="application/xop+xml";
boundary="----=_Part_0_14949315.1177991007796"; start-info="text/
xml"
Accept: text/xml, application/xop+xml, text/html, image/gif, image/
jpeg, *; q=.2, */*; q=.2
User-Agent: Java/1.5.0_11
Host: 127.0.0.1:8081
Connection: keep-alive

------=_Part_0_14949315.1177991007796
Content-Type: application/xop+xml; type="text/xml"; charset=utf-8

<?xml version="1.0" ?><soapenv:Envelope xmlns:soapenv=http://
schemas.xmlsoap.org/soap/envelope/ xmlns:xsd=http://www.w3.org/2001/
XMLSchema xmlns:ns1="http://personalbanking.services.novobank.
com/">/"><soapenv:Body><ns1:openAccount><arg0><xop:Include
xmlns:xop="http://www.w3.org/2004/08/xop/include" href="cid:f17b4f2b-
db2c-4bc5-96d1-2d4857aaa5b8@example.jaxws.sun.com"></xop:Include></
arg0></ns1:openAccount></soapenv:Body></soapenv:Envelope>

------=_Part_0_14949315.1177991007796
Content-Type: application/octet-stream
Content-ID: <f17b4f2b-db2c-4bc5-96d1-2d4857aaa5b8@example.jaxws.sun.
com>
Content-transfer-encoding: binary_KýÝ_ÔF-¡úÞr*îÉu1_)Š1áñ}K²£•Ñ\
êKQZ_Ý4Ï¡É)*G'~ªohBïýfÙÈï¦zùì£_Ö(Ö &_Èõmâoå¨\Ó\( ò¹åq€°Ê _W¶èÁh";_
ÐÚ´z0"ç5WÃ"_üv|DÜî7IÚù_é6³ÈÚ1•lëÉäl›BîöWÈ"ý|›i"ì™Åã]ÓÉÝ9ƒ_"7Üý¶j9{
ßáÉ?w(_"86‹ü£Ã_´?_:Ž§òÔŠvM¦éÈë4_ŸÊ"=ƒÑ]™EýÕ×Ès˜Hýå÷©?7‹â""¨r{‹â³¯
Rè<...R×©Ô]z_íµÙ4_ÿBîùš_?ÿPé3ê
... the rest of the data omitted ...
------=_Part_0_14949315.1177991007796--
```

Example 7.27
A SOAP message with an MTOM-compliant binary part

The request message is divided into the SOAP message, including reference to the binary data, and the binary data sent as an attachment. The message can be handled more efficiently and does not impact the processing of the remaining XML information. Again, whether the data is sent as an attachment or not is not defined in the service contract (the WSDL definition).

Binary Data in REST Services

JAX-RS supports the conversion of resource representations into Java types, such as `byte[]` or `java.io.InputStream`, in resource methods. To produce or consume binary data from resource methods, JAX-RS runtime handles the conversion through a number of built-in content handlers that map MIME-type representations to `byte[]`, `java.io.InputStream`, and `java.io.File`. If binary content must be handled as a raw stream of bytes in a resource method, the corresponding resource method is seen in Example 7.28.

```
@Get
@Produces("image/jpg")
public byte[] getPhoto() {
  java.io.Image img = getImage();
  return new java.io.File(img);
}

@Post
@Consumes("application/octet-stream")
public void processFile(byte[] bytes) {
  //...process raw bytes
}
```

Example 7.28

NOTE

Indicate the MIME type through an appropriate `@Produces` annotation, such as a JPG image, so that the runtime can set the right content-type for the returned resource representation.

The `java.io.File` type can help process large binary content in a resource method, such as an attachment containing medical images, as seen in Example 7.29.

```
@POST
@Consumes("image/*")
public void processImage(File file) {
  //...process image file
 }
```

Example 7.29

The JAX-RS runtime streams the contents to a temporary file on the disk to avoid storing the entire content in memory. For complete control over content handling, JAX-RS offers several low-level utilities which can be useful for custom marshaling/unmarshaling of various content types. The `javax.ws.rs.ext.MessageBodyReader` and `javax.ws.rs.ext.MessageBodyWriter` interfaces can be implemented by developers to convert streams to Java types and vice versa.

Back-and-forth conversion is useful when mapping custom MIME types to the domain-specific Java types. The classes that handle the mapping are annotated with the `@Provider` annotation and generically referred to as JAX-RS entity providers. Entity providers are used by the JAX-RS runtime to perform custom mapping.

A special case of `javax.ws.rs.ext.MessageBodyWriter` is the `javax.ws.rs.core.StreamingOutput` callback interface, which is a wrapper around a `java.io.OutputStream`. JAX-RS does not allow direct writes to an `OutputStream`. The callback interface exposes a `write` method allowing developers to customize the streaming of the response entity. Example 7.30 demonstrates the `gzip` format used to compress a response entity.

```
import javax.ws.rs.core.StreamingOutput;
...
@GET
@Produces("application/gzip-compressed")
public StreamingOutput getCompressedEntity() {
```

```
return new StreamingOutput() {
  public void write(OutputStream out)
    throws IOException, WebApplicationException {
      try {
        ...
        GZipOutputStream gz =
          new GZipOutputStream(out);
          //...get array of bytes
          //   to write to zipped stream
          byte[] buf = getBytes();
          gz.write(buf, 0, buf.length);
          gz.finish();
          gz.close();
          ...
          } catch(Exception e) { ... }
        }
    };
}
```

Example 7.30

For mixed content containing both text and binary payload, entity providers can be used to perform custom marshaling, while multipart MIME representations are suitable for dealing with mixed payloads. JAX-RS standards do not mandate support for handling mixed MIME multipart content types apart from multipart FORM data (multipart/form-data), which is useful for HTML FORM posts but has limited use in a system-to-system interaction context. Various JAX-RS implementations, such as Jersey and RESTEasy, provide support for mixed multipart data. Handling mixed content with binary data is common, as seen in the following case study example for SmartCredit's Submit Credit Application service.

CASE STUDY EXAMPLE

SmartCredit is building a REST service known as the Submit Credit Application service that is intended for service consumers to submit credit card applications. Apart from basic information such as customer details, the supporting information in the form of various collaterals, such as mortgage papers and loan approvals, must be scanned as images and attached to the application.

The SmartCredit application development team considered using Base64 encoding, but moved onto other alternatives after realizing a substantial size bloat would result. The development team decides on the mixed multipart representation for the application data. The multipart application data will have the customer information in an XML format as the first part, and a series of images in the subsequent parts.

A sample multipart application request over HTTP is shown in Example 7.31.

```
POST /creditapps/ HTTP/1.1
Host: smartcredit.com
Content-Type: multipart/mixed; boundary=xyzw

--xyzw
Content-Id: <abcdefgh-1>
Content-Type: application/xml
<customer>
  <name>John Doe</name>
  <Address>...</Address>
  ...

--xyzw
Content-Id: <abcdefgh-2>
Content-Type: image/jpg
...
&_Èõmâoå¨\Ó\( ò¹åq °Ê _W¶èÁh";_ÐÚ´z0"ç5WÃ"_üv|DÜî7IÚù_
é6³ÈÚ1•lëÉäl›BîöWÈ"ý|›i"ì™Åã]ÓÉÝ9ƒ_"7Üý¶j9{ßáÉ?w(_"86‹ü£Ã_´?_:Ž§òÔŠv
M¦éÈë4_ŸÊ"=ƒÑ]™EýÕ×Ès˜Hý...rest of the binary data goes here
--xyzw—
```

Example 7.31

The SmartCredit service development team considered using a custom JAX-RS Entity Provider to handle the mixed multipart data, but realized the reference JAX-RS implementation Jersey already provides support for mixed multipart data through an add-on called `jersey-media-multipart`. The key classes leveraged in this implementation include:

- The `org.glassfish.jersey.media.multipart.BodyPartEntity` represents the entity of a part when a MIME Multipart entity is received and parsed.
- The `org.glassfish.jersey.media.multipart.BodyPart` is a mutable model representing a body part nested inside a MIME Multipart entity.

The resource class method that handles the submitted application can be seen in Example 7.32.

```
import org.glassfish.jersey.media.multipart.MultiPart;
import org.glassfish.jersey.media.multipart.BodyPart;
import org.glassfish.jersey.media.multipart.BodyPartEntity;
import javax.ws.rs.core.Response;
...

import com.smartcredit.domain.Customer;
...

@Path("/creditapps")
public class CreditAppResource {

    @POST
    @Consumes("multipart/mixed")
    public Response post(MultiPart multiPart) {
      // First part contains a Customer object
      Customer customer =
        multiPart.getBodyParts().get(0).
          getEntityAs(Customer.class);

      // process customer information
      processCustomer(customer);

      // get the second part which is a scanned image
      BodyPartEntity bpe =
        (BodyPartEntity) multiPart.getBodyParts().
        get(1).getEntity();
      try {
        InputStream source = bpe.getInputStream();
        //process scanned image
      }

      // Similarly, process other images in the multipart
      // content, if any...

      //If everything was fine, return Response 200
      return Response.status(Response.Status.OK).build();

      //else if there were errors...
      return Response.status(Response.Status.BAD_REQUEST).
      build();
}
```

Example 7.32

In the code fragment, the `@Consumes` annotation indicates that a resource representation of multipart/mixed is expected. In this case, the payload contains customer data in XML and one or more scanned images. The following Jersey utilities perform different steps in managing the mixed multipart data:

- `com.sun.jersey.multipart.MultiPart.getBodyParts()` returns a list of `com.sun.jersey.multipart.BodyParts`.
- `BodyPart.getEntityAs(Class<T> cls)` returns the entity converted to the passed-in class type.
- `com.smartcredit.domain.Customer` is a regular JAXB-annotated Java class. Since the first entity in the multipart message is known to be the customer entity, the `BodyPart.getEntityAs(Customer.class)` method is used to unmarshal the XML entity body into a JAXB `customer` object.
- `BodyPart.getEntity()` returns the entity object to be unmarshaled from a request. The entity object is known to be a `BodyPartEntity` and is cast accordingly.
- `BodyPartEntity.getInputStream()` returns the raw contents, which in this case are the contents of the scanned image.

Note that by using the MIME multipart utilities in Jersey, the development team is able to avoid writing much of the plumbing code that would otherwise be necessary to deal with multipart MIME messages.

Use of Industry Standards

The use of industry standards in developing service contracts builds on the IT-specific standards and seldom offers challenges when using Java as the language and runtime environment. Many industries have established data formats that ensure interoperability between business partners, suppliers, and between systems within an enterprise, such as the ACORD standard for insurance, HL7 for the healthcare industry, or SWIFT for banking.

Used primarily to exchange information between companies or independent units within an enterprise, industry standards are prime candidates for use as part of the service contract. Industry standards are generally expressed as XML schema definitions, which can be directly referenced in a service contract or serve as the basis for a specific schema.

Before the advent of JAXB 2.0 which supports the full set of XML schema constructs, a common issue was the inability to map all of the elements used in an industry schema into Java because such mapping was not defined. JAXB 2.x nearly resolves this issue, but cases still occur where a large and complex industry standard schema cannot be handled by a data binding tool like JAXB. Using an industry standard unchanged in a service contract can be tedious and lead to the generation of hundreds of Java classes mapping all of the complex types defined in the schema.

SUMMARY OF KEY POINTS

- SOAP-based Web services can be developed top-down, bottom-up, or meet-in-the-middle.
- For REST services, a resource implementation artifact is used to model a Web resource that responds to HTTP operations. Apart from XML, support for resource representations can encompass a wide range of other media types. REST services can also use MIME features, such as multipart messages, to deal with binary content. JAX-RS implementations provide content handlers which can be customized for dealing with different representations.
- Standards-based service contracts can map the relevant XML schema constructs to and from Java. For both SOAP and REST services, the JAXB 2.0 standard offers support for the entire set of XML schema features. Industry standards are widely used in service contracts and do not introduce particular challenges when using Java.
- For SOAP-based Web services, special considerations apply when leveraging specific WSDL features, such as header fields, attachments with binary data, or the use of wrapper elements.

7.3 Service Loose Coupling

In a service-oriented environment, components can be coupled at a number of different levels. Coupling can occur at the level of the service contract, the service implementation logic, or the underlying technology platform the service is running on if a service consumer is coupled with a particular service. In general, SOA promotes the notion of decoupled systems by which the parts of a service-oriented solution are as decoupled as possible. Reducing the dependency of one part on another allows one area, such as a service or an aggregation, to be changed without requiring changes to other areas.

The following design characteristics are helpful in the creation of decoupled systems:

- separation of contract and implementation
- functional service context with no dependency on outside logic
- minimal requirements for the service consumer

When service logic is implemented in Java and executed in a JRE, a coupling is created between the logic and the underlying technology platform, Java. Additional dependencies arise when Java EE is used as the hosting platform. For an in-depth explanation of the variations of positive and negative coupling, see Chapter 7 in *SOA Principles of Service Design*.

Separation of Contract and Implementation

The separation of a service contract from its implementation is a key characteristic of service-oriented design that was first established as part of the definition of remote procedure calls and then in object-oriented programming. The interface of a component is exposed to a point that allows other logic to invoke this component, but no details about the implementation are added. Changes to the implementation can then be made without affecting the client. SOA took this concept further by establishing the notion of a standards-based contract. For example, the service interface can be expressed in a programming language-neutral way to allow for the integration of logic written in different languages and running on different platforms.

The choice of programming language can be exposed in the service contract. In the context of a SOAP Web service from the *Top-Down vs. Bottom-Up* section, a common approach is to generate the service contract from existing Java logic. In JAX-WS, the wsgen tool can be used to generate a complete WSDL document, including the XML schema definition from an existing JavaBean or Java interface. However, the existing Java code may not be easily mapped into XML schema.

For example, assume that the `public java.util.Hashtable<String, String> getTable(int i);` method is part of an interface to be turned into a service. Using the wsgen tool to generate both the WSDL and XML schema definition from this interface creates the type definitions presented in Example 7.33.

```
<xs:element name="getTableResponse" type="ns2:getTableResponse"
  xmlns:ns2="http://the.package/"/>
<xs:complexType name="getTableResponse">
  <xs:sequence>
    <xs:element name="return" type="ns3:hashtable" minOccurs="0"
      xmlns:ns3="http://the.package/"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="hashtable">
  <xs:complexContent>
    <xs:extension base="ns4:dictionary"
      xmlns:ns4= "http://the.package/">
      <xs:sequence/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="dictionary" abstract="true"/>
```

Example 7.33

The result is a valid XML schema definition, although a Java `Hashtable` is exposed in the service interface which renders the schema not useful or usable outside of Java. The same is true for many other classes and interfaces in Java, which, while useful as part of the implementation, do not map well into XML schema and non-Java environments. Many of the more technically-oriented classes, including the members of the `java.io` package, fall into this category.

The use of internal, platform-specific information carried in a generic type is another aspect of decoupling the contract and implementation. For example, the name of a file usually stored as a string can be mapped into XML schema and added to a service contract. However, exposing file names on a service contract is considered poor practice, and revealing details of the service implementation to the service consumer should be avoided. The same holds true for names of databases, database tables, machine names, and addresses.

Using the reverse approach is another way of creating this coupling, particularly when generating a service logic skeleton directly from a service contract. In JAX-WS, the wsimport tool is used to create Java code skeletons representing a given WSDL definition. Despite being the recommended approach, be aware that the generated Java code is now tightly coupled to its contract, which prevents the code from being easily reused to serve other types of service contracts or updated versions of the current contract. In most cases, the tight coupling is acceptable because the logic is created for a particular contract and nothing else.

However, instances occur where service logic must be created for reuse despite changes to the underlying contract changes or the need to concurrently serve multiple versions of the same contract. The JAX-WS `Provider` API is equipped for such instances. In this model, the service logic parses the incoming message at runtime for dynamic processing with no direct dependency on the types and elements defined in the WSDL contract. Use of the JAX-WS `Provider` API is detailed in Chapter 8.

For a REST service, coupling the code to the service contract is inconsequential because generating implementation artifacts from a machine-readable contract, unless WADL is being used, is uncommon. Implementing a Web resource for a platform, such as JAX-RS, out of the box supports only a finite subset of Java classes that can be automatically mapped to appropriate content-types. For custom types not mapped automatically by the built-in content handlers, developers must provide implementations of `javax.ws.rs.core.MessageBodyReader/MessageBodyWriter` interfaces to map such types to a known set of resource representations or media types.

Independent Functional Contexts

Besides the direct compile-time of coupling services, consider the coupling of a service to its outside functional context. Service invocations happen as part of a business transaction or process to establish a context that effectively binds the services, which are invoked downstream, into an aggregated set. Having a set of services as part of the same process establishes a type of coupling between the services, a coupling that should always be top-down. Particularly when leveraging other services to fulfill functionality, a service implementation can be coupled with or have dependency on those services. However, the service should not have a dependency on any services at a higher level of the process hierarchy, or be coupled with the invoking service or with peer services.

For example, assume that a service offers Credit Check functionality. The service is implemented as a business process that invokes a number of other finer-grained services, such as Credit Lookup, Credit Eligibility, Update Profile, and Notification. All four downstream services are peers within the Credit Check business process. Service peers should have no dependency on each other, or be aware of or coupled with their upstream Credit Check service.

Services representing business processes, such as Credit Check and Maintain Customer Information, are decomposed into a set of fine-grained services to create a downstream dependency. A service should not introduce a dependency on another service that is higher level. For example, the implementation of the Update Profile entity service should not introduce any dependency on the Maintain Customer Information task service.

In Java, the same principles of downstream dependency hold true. Unwanted dependencies can be detected by examining the classes that are used by a piece of Java logic. Organizing classes into packages directly identifying the service and/or affiliated business process and ensuring that logically decoupled functions are not packaged together is recommended. A package name should be selected with consideration for possible reuse opportunities. For Web services, the same requirements for namespaces are used in the service contract.

Service Consumer Coupling

For SOAP-based Web services, a service consumer will often be tightly coupled with the service contract and not the implementation of the service being invoked. However, a looser coupling lessens the impact when the service contract changes.

Using the JAX-WS Dispatch API, service consumer logic can dynamically assemble a request message at runtime and manage any response messages. Additional effort is required to build service consumer logic that can build request messages that the intended service can process.

CASE STUDY EXAMPLE

After using the Account service in production for some time, a new operation is added to enhance account services. NovoBank wants to allow all required information about new accounts to be sent to the service as XML on top of the binary image that the initial version supported. Different branch offices use different systems to capture the data required to open a new account, such as traditional “fat client” or browser-based solutions. Additionally, the details of the information stored with new accounts change consistently.

The development team will design the new operation to process different types of input XML formats, and deliver a generic piece of service consumer code that shows how to invoke the new operation from within a JAX-WS supported client. Example 7.34 illustrates the updated WSDL definition for the enhanced Account service to accept the input of XML formats.

```
<definitions targetNamespace="http://personalbanking.services.
  novobank.com/" ...>
  <types>
    <xsd:schema>
      <xsd:element name="openAccount" type="ns1:openAccount"
        xmlns:ns1="http://personalbanking.services.novobank.com/"/>
      <xsd:complexType name="openAccount">
        <xsd:sequence>
          <xsd:element name="arg0" type="xs:base64Binary"
            xmime:expectedContentTypes="image/jpeg" minOccurs="0"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:element name="openAccountXML" type="ns1:openAccountXML"
        xmlns:ns1="http://personalbanking.services.novobank.com/"/>
      <xsd:complexType name="openAccountXML">
        <xsd:sequence>
          <xsd:any/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:element name="openAccountResponse"
        type="ns2:openAccountResponse"
        xmlns:ns2="http://personalbanking. services.novobank.com/"/>
      <xsd:complexType name="openAccountResponse">
        <xsd:sequence>
          <xsd:element name="return" type="xsd:string" inOccurs="0"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </types>
  <message name="openAccount">
    <part name="parameters" element="tns:openAccount"/>
  </message>
  <message name="openAccountResponse">
    <part name="parameters" element="tns:openAccountResponse"/>
  </message>
  <message name="openAccountXML">
    <part name="parameters" element="tns:openAccountXML"/>
  </message>
  <portType name="Account">
    <operation name="openAccount">
      <input message="tns:openAccount"/>
      <output message="tns:openAccountResponse"/>
    </operation>
    <operation name="openAccountXML">
      <input message="tns:openAccountXML"/>
    </operation>
  </portType>
...
</definitions>
```

Example 7.34

The updated WSDL definition for the enhanced Account service accommodates the new operation, `openAccountXML`.

Note how the element named `openAccountXML`, which acts as the wrapper element for the `openAccountXML` operation, contains only one `<xsd:any/>` element. The contract indicates that any kind of XML content can be sent to the service without providing any further details, which allows for decoupling of the service logic from the contract.

Use of the `<xsd:any/>` element minimizes the requirements for service consumers of this service. Any XML document can be passed to the service, allowing the development of generic service consumer logic. As an example of completely decoupling the service consumer from the service, the NovoBank development team delivers the following piece of client code to the users of the Account service in Example 7.35.

```
package com.novobank.services.personalbanking.client;

import java.io.StringReader;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;

public class Account2Client {
  public static String testMessage =
    "<ns1:openAccountXML xmlns:ns1=\"http://personalbanking.
    services.novobank.com/\"><someDocument><personalData>Here goes
    the information</personalData></someDocument> </
    ns1:openAccountXML>";

  public static void main(String[] args) throws Exception {
    QName serviceName =
      new QName("http://personalbanking.services.novobank.com/",
        "AccountService");
  QName portName = new QName("http://personalbanking.services.
    novobank.com/","AccountPort");

  Service service = Service.create(
    new URL("http://localhost:8080/account2/account2?wsdl"),
    serviceName);
  Dispatch<Source> dispatch = service.createDispatch(
    portName, Source.class, Service.Mode.PAYLOAD);
    dispatch.invoke(new StreamSource(new StringReader(testMessage)));
  }
}
```

Example 7.35

Java client code decouples the service consumer from the service for the Account service.

First, an instance of the `Service` class providing the location of the appropriate WSDL file and the name of the targeted service is created. A `Dispatch` object is then created from the `Service` class, and the `Service.Mode.PAYLOAD` is defined to indicate that only the content within the SOAP `<body>` element and not the entire message will be passed. Finally, the service can be invoked dynamically via the `Dispatch` object. The service consumer has no compile-time dependency on the service or its contract.

The payload XML document has one root element called `openAccountXML`. Given that the service definition used the wrapped document/literal style described in the *Mapping Between Java and WSDL* section, this `openAccountXML` element indicates which operation of the service is being invoked. Similarly, the service code can be developed in a flexible way with regard to any kind of input message being sent, of which a detailed case study can be found in Chapter 8. Even though any XML content can be sent to the service, the implementation logic will always have constraints on what can be processed. The message must contain data meaningful in the context of the invoked operation.

Another way of further decoupling a service consumer from the service is to insert an intermediary between the two. The intermediary, generally deployed as part of an ESB, can mediate the differences in message format and network protocol to further decouple the service consumer and service. Chapter 12 explores this further as part of its coverage of ESBs.

SUMMARY OF KEY POINTS

- For SOAP-based Web services, coupling between service contract and implementation can also occur. Generating service contracts directly from existing Java logic often exposes language-specific details and creates an unwanted tight coupling between the logic and the contract. However, service consumers can be developed in a generic way independent of a particular service contract using the JAX-WS `Dispatch` API.
- With REST services, the service contract is tightly constrained by a known set of media types and a handful of HTTP operations.
- Namespaces and Java package names can help structure code to control and minimize the dependencies among service implementation pieces.

7.4 Service Abstraction

The appropriate level of abstraction at which services are described achieves additional agility and alignment between business and IT areas of an enterprise. Abstracting information means taking technical details out of a problem to be solved on a higher level. Since the beginning of computer technology, information has been abstracted into higher levels in a number of ways, for example:

- Assembler constructs, which are instructions for a processor, are translated into a series of `1` and `0`.
- Operating systems offer access to system resources and APIs that encapsulate lower-level constructs.
- Programming languages, such as Java, introduce a more abstract way of describing logic, which is then compiled into a format that the operating system can understand.

Service-orientation continues the evolution of higher-level abstraction use to make creating and changing solutions easier. For example, a business process defined with WS-BPEL describes a sequence of service invocations and the data flows between them by expressing this sequence in XML form without writing any actual code. A side effect of this increased abstraction is the ability to utilize visual programming tools that support the creation of process definitions via drag-and-drop-style interfaces. The Service Abstraction principle advocates that the technical details of the technology platform underlying a service contract are hidden from the service consumer. It also promotes hiding non-essential details about the service itself.

Abstracting Technology Details

The service contract represents an abstraction of the functionality implemented in the service logic. Included in this notion is the abstraction of technical resources utilized to fulfill a service's functionality.

Filenames, database tables, machine names, and network addresses should be omitted from the service contract and completely hidden from service consumers. Given that a service contract should always be designed with a particular business purpose in mind, this should never be a problem.

Concerns arise when services are generated straight out of existing code, because technology details which should have otherwise been abstracted away will often be exposed. For example, whether or not a service is implemented in Java or running in a Java environment such as Java EE should be completely irrelevant to the service consumer.

Hiding Service Details

Maximum flexibility is achieved when the technology used to implement a service and additional details about that service are hidden, which can be divided into information about the input or output message format and contextual information.

Hiding the information about the input or output message format may seem counterintuitive. If a service's input and output messages are hidden, what is left to put into the service contract? Message formats can be abstracted to a generic level without surrendering the message definition altogether.

For example, assume a Credit Check service receives customer information as input. The customer information can be defined and represented by a Customer complex type in the XML schema definition to a detailed level, adding constraint information to each attribute and element of that schema. The length of the `lastName` character field is limited to 35 characters in Example 7.36.

```
<complexType name="Customer">
  <sequence>
    <element name="firstName" type="string"/>
    <element name="lastName">
      <simpleType>
        <restriction base="string">
          <length value="35"/>
        </restriction>
      </simpleType>
    </element>
    <element name="customerNumber" type="string"/>
  </sequence>
</complexType>
```

Example 7.36
XML schema definition with added constraint inheritance can limit the `lastName` field to a set number of characters.

The XML schema definition in Example 7.36 maps to a Java class, `Customer.java`, as seen in Example 7.37. (The generated Javadoc comments are omitted.)

```
public class Customer {
  @XmlElement(required = true)
  protected String firstName;
  @XmlElement(required = true)
  protected String lastName;
  @XmlElement(required = true)
  protected String customerNumber;

  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String value) {
    this.firstName = value;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String value) {
    this.lastName = value;
  }
  public String getCustomerNumber() {
    return customerNumber;
  }
  public void setCustomerNumber(String value) {
    this.customerNumber = value;
  }
}
```

Example 7.37
The generated Java type does not include the schema type restriction.

NOTE

The limit on the length of the `lastName` element was not carried over into Java because Java has no concept of a fixed-length string.

On the opposite end of the abstraction spectrum would be a message definition stating that the incoming message is an XML document with a root `Customer` element. No information is given about individual attributes or elements contained in the document, as seen in Example 7.38.

```
<complexType name="Customer">
  <sequence>
    <any/>
  </sequence>
</complexType>
```

Example 7.38
An XML schema definition using the `<any/>` element

The generic type definition leads to a generic Java class, as shown in Example 7.39.

```
public class Customer {
  @XmlAnyElement(lax = true)
  protected Object any;
  public Object getAny() {
    return any;
  }
  public void setAny(Object value) {
    this.any = value;
  }
}
```

Example 7.39
The `<any/>` element is mapped to `java.lang.Object`.

Hiding service details and increasing what is abstracted about a service may always appear to be a prudent step. However, the service consumer is sometimes not provided with all of the necessary information on what exactly the service expects and what will be returned in response. Details of an interaction on both sides are left to be resolved at design-time or runtime (outside of the service contract).

In Example 7.38, the generic version states that a `Customer` object contains a `java.lang.Object`, which must be defined at runtime to allow for processing. Generally, more Java code must be written for errors that can occur at runtime. For XML payloads, this consideration is equally valid for SOAP and REST services, as the mechanics perform the same role for both in mapping XML to Java via a binding tool such as JAXB. An abstract service contract can be appropriate in some instances, such as utility services.

Contextual data can also be hidden, as this type of data is commonly about an interaction which does not contain any business-relevant payload information. When using SOAP-based Web services, SOAP header fields store contextual data, such as unique message identifiers, timestamps, and service consumer identity. For greater abstraction, detailed information about contextual header fields can be left out of the service contract altogether. This contextual information can be added or removed depending on the environment in which a service runs, is not relevant for its business purpose, and can often be left out of the contract.

For REST services, in the absence of any kind of a payload envelope, contextual information must be part of a resource representation. Such resource metadata can still be packaged inside specially designated header elements. The technical details of a service that are not part of the service interface, such as a WSDL or service-level information about response times and availability, can be abstracted. The technical details can be important to know, but often change and depend on a particular runtime environment and deployment of a service. For REST services, such service-level agreement characteristics can be described in a separate document.

Document Constraints

Non-technical information about a service cannot be articulated in a standard format. A generic example of this is a service-level agreement, but may also include other constraints about the usage of a service, valid ranges of input data beyond what can be expressed in WSDL and XML schema, and any additional applicable documentation.

A service can be implemented and deployed in different ways throughout an enterprise. As such, this documentation should not be directly linked with a service. Java, for example, is well suited for deployment on multiple platforms and operating systems. A Unix-based environment has different performance, scalability, and availability characteristics than a Windows-based system. Additionally, a Java EE application server can be leveraged to host the Java logic. A service instance can run on just one server instance on a small machine. As reuse of the service increases, the service instance can be moved to a clustered environment with greater computing power.

As per the Dual Protocols pattern, a Web service offered over HTTP can be later exposed via JMS for additional reliability requirements by particular service consumers. Abstract service contracts provide the freedom to make changes throughout the lifetime of the service, without breaking or violating previous versions. REST service implementations are synonymous with HTTP, making transport mechanism abstraction a non-issue. As the information about a service grows in abstraction, the service implementation and service consumer logic must become more flexible to anticipate future changes.

SUMMARY OF KEY POINTS

- Details about a specific technology used to implement and/or host a service should be abstracted out of a service contract, which is achieved by REST services over HTTP by default.
- Services can be built in a more abstract fashion by using abstract and generic message specifications and leaving contextual information out of the service contract altogether.
- Non-technical information about a service often assumes a separate life-cycle from the service contract and its implementation.

7.5 Service Composability

The composability of a service is an implicit byproduct of the extent to which the other service-orientation design principles have been successfully applied. The ability to compose services into new, higher-level services is an inherent, logical consequence if the other design principles outlined in this chapter are followed. For example, a service contract that is standardized allows interaction and composition between services implemented in different languages using different runtime environments. Decoupling a service implementation from its contract allows the same logic to be reused in other compositions.

With regards to the implications that service composability has for the service contract, this section highlights some of the issues and requirements for the runtime environment in which the services run. See Chapter 11 for further exploration of service composition with Java.

Runtime Environment Efficiency

A highly efficient and robust runtime environment is required for service compositions. The ability for services to participate in multiple compositions places severe challenges on runtime environments and must be taken into account when designing a service inventory.

If a service is used by multiple compositions, applying different non-functional characteristics can be necessary. One composition rarely invokes a service with no particular requirement for fast response times, whereas another composition using the same service can require support for high transaction loads with short response times. Similarly, one composition can require a reliable connection between the composition controller and its members, whereas another can be tolerant of lost messages. Applying different QoS definitions to the same service, depending on the composition, must be possible without requiring code changes in the service logic itself.

Java, as a programming language, has no built-in features that help or hinder the runtime environment. In most cases, Java-based services are hosted in a Java EE-compliant application server or an alleged Web server environment, which is Java SE-compliant. In either case, the runtime environments provide advantages over other traditional server platforms in terms of composability.

NOTE

Depending on how a given service or service consumer participates in a service composition at runtime, it may assume one or more roles during the service composition's lifespan. For example, when a service is being composed, it acts as a composition member. When a service composes other services, it acts as a composition controller. To learn more about these roles, visit www.serviceorientation.com or read Chapter 13 of *SOA Principles of Service Design*.

Java EE defines the management of solution logic according to the roles people perform, such as component provider, deployer, and assembler. As a result, much of the information on the execution of solution logic at runtime is not hardcoded into the logic itself but is instead stored in declarative configuration files, called deployment descriptors. The same piece of Java code can be used differently with different deployment descriptors and changed at runtime without requiring recompilation of the code. Despite being undefined by the Java EE standard, most Java EE application servers support defining individual server instances with specific runtime definitions. One instance usually runs in one process with its own configuration for elements like thread pools or memory heap sizes. In many cases, instances can be clustered to provide one logical application server across multiple physical processes or machines.

Separating runtime information about a component from its actual logic is possible because the Java EE application server runs as a container. This means that all incoming and outgoing data is intercepted and processed by the application server based on the current configuration. For example, if a certain piece of Java logic can only run within the scope of a transaction, the container can be configured to ensure a transaction is present whenever this piece of logic is invoked.

The same approaches to the runtime environment apply to Web services on two levels:

1. The runtime environment that processes an incoming message, such as a SOAP or REST request message, can perform several tasks before forwarding the request to the actual service. These tasks include the ability to decrypt data sent in an encrypted form, establish handshaking between service consumer and service by returning acknowledgement that the message has been received, interpret information in the request message header indicating that the invocation of the service must be part of a distributed transaction, and convert incoming XML data into Java objects.

2. For SOAP-based Web services, JAX-RPC and JAX-WS provide a mechanism and an API that allow the insertion of custom logic to parse, interpret, or change incoming and outgoing messages. The custom logic runs inside a handler. For JAX-RS-based implementations of REST services, such interception logic can be implemented by developers in special entity handlers known as entity providers. The JAX-RS 2.0 release provides added filters and interceptors otherwise not included in previous versions.

Both approaches are similar regardless of the Web services used. One is controlled by the user of the system, whereas the other is implicitly included with the runtime. Reading and manipulating incoming and outgoing messages separate from the service logic is crucial to supporting the Service Composability principle. Composing hosted services, including both composition members and composition controllers, is supported by the concept of containers and deployment descriptors and enhanced by SOAP-based Web services, such as handlers.

Ultimately, implementing a highly efficient runtime allows the developer to focus on the business logic of the actual service implementation, leaving everything else to the underlying Java platform. More advanced technologies, such as the SCA, expand on this concept by separating core business logic implementation further away from aspects of a service component, such as the protocol bindings used to interact with service consumers and other services used to fulfill functionality.

Service Contract Flexibility

Service contracts can be designed to increase the ability of a service for multiple compositions. Generally, multiple compositions are only applicable to the composition members for increasing the reusability of a service in different business contexts or business tasks. A service contract can be rewritten to enable reuse of a service without changing the core functionality of the service.

To write a flexible service contract that is reusable, recall the following approaches:

- Use generic data types or supertypes instead of concrete subtypes. If an enterprise deals with both commercial customers (`CommercialCustomer`) and personal customers (`PersonalCustomer`), evaluate whether a common supertype can be established for both (`Customer`) to be used in the service contract. JAXB supports polymorphism to ensure that the appropriate Java object is created when a message containing a subtype is received.
- Decouple the service contract from its underlying runtime platform by hiding details about QoS characteristics, which can change over time and will vary depending on service consumer requirements and how a service is deployed.
- Decouple the service implementation from its contract by utilizing generic APIs, such as the JAX-WS Provider API for SOAP-based Web services. For REST services, deal with generic Java types in resource methods, such as `String`, `byte[]`, and `InputStream`. Note that generated generic service consumer or service logic results in additional code that must be developed and tested.

Standards-Based Runtime

Composition members and controllers benefit from a runtime environment that supports a wide range of accepted standards. Composing services means the services interact and interoperate. Interoperability of services is supported by a runtime environment upheld by relevant standards, such as the WS-I. Java's APIs for SOAP-based Web services, JAX-RPC and JAX-WS, require support for the WS-I Basic Profile, which allow services to be designed with a high degree of interoperability. REST services achieve full interoperability inherently through HTTP.

Advanced standards relevant in a composition of services include the WS-Security standards for which a WS-I profile also exists, the WS-Transaction and related standards, WS-Addressing, and WS-ReliableMessaging, which supports the reliable exchange of messages between services. These advanced standards are combined in another WS-I profile known as the Reliable Secure profile.

SUMMARY OF KEY POINTS

- Composability is supported in utilizing a runtime environment that allows hosting both composition members and composition controllers efficiently and flexibly, such as Java EE-compliant application servers.
- Creating flexible service contracts can facilitate the use of services as composition members.
- Platforms that support accepted and established standards can be utilized to improve service interoperability and composability.

7.6 Service Autonomy

Service-orientation revolves around building flexible systems. Flexibility is, to a large degree, achieved through making services decoupled and autonomous to enable them to be composed, aggregated, and changed without affecting other parts of the system. For a service to be autonomous, the service must be as independent as possible from other services with which it interacts, both functionally and from a runtime environment perspective. Java and Java EE provide a highly efficient runtime environment that supports service composition. For example, a Java EE application server supports concurrent access to its hosted components, making each access to such a component autonomous from the others.

Well-Defined Functional Boundary

Occasionally, the functional boundary is defined by a certain business domain that a service lives within, as is the case if a service implements a particular business process or task within that domain. Alternatively, the functional boundary of a service can be described by the type of data the service operates on, such as entity services.

Translating this requirement into Java and XML schema utilizes namespaces and Java packages as structuring elements. Checking the list of imported classes and packages for a particular service implementation will help identify dependencies throughout the system and provide an indication of whether the functional boundary of the service is maintained in its implementation.

For example, an entity service called Customer delivers customer data retrieved from a variety of data sources for reuse across many business processes. The service is defined in the `http://entity.services.acme.com/Customer` namespace and uses `com.acme.services.entity` as the Java package for its implementation. The Customer service imports a package called `com.acme.services.accounting`, which immediately identifies that the Java service implementation contains a potentially undesired dependency on a business domain-specific piece of code. This warrants further investigation of the underlying logic and removal of the dependency.

The Customer service has a well-defined functional boundary in delivering relevant customer data to its service consumer. However, a dependency on logic specific to the Accounting service business domain naturally reduces the autonomy of the Customer service.

Runtime Environment Control

The underlying runtime influences the degree of autonomy a service can achieve. For each service to have control over its runtime environment, the environment must be partitioned to allocate dedicated resources accordingly. The JVM offers all internal code a degree of autonomy by isolating the executed code from the operating system and providing controlled access to physical resources, such as files or communication ports. Java EE application servers leverage the concept of a container in which components run.

For SOAP-based Web services, runtime control in Java can be achieved (while maintaining a high degree of autonomy) by exposing plain JavaBeans as Web services or utilizing Stateless Session EJBs. The service implementation and all other relevant artifacts, such as WSDL files, are packaged in a module, such as a JAR or WAR file, which can then be installed on an application server independent of other code running on that server.

The same is true for non-Web services or services implemented as regular EJBs. The components related to the service have individual private deployment descriptors that can be configured on the application server as independent entities.

Java EE allows for the packaging of multiple modules and multiple services in one Enterprise ARchive file. This EAR file is deployed and installed on the application server as an independent enterprise application. To increase a service's autonomy, use of only one service packaged per EAR file is recommended. This allows each service to be treated as a completely independent unit to configure, start, stop, or replace without affecting any other services running on the same system.

To decrease the number of moving parts in the environment, however, multiple services can be packaged into one enterprise application. Co-locating services is suitable when the services interact frequently with each other in a performance-critical manner.

In JAX-RS, POJOs are more commonly used to model Web resources, although stateless and single session beans can be designated as root resource classes. The JAX-RS runtime packages the resource artifacts into a WAR or EAR file which can be deployed as a standalone module in an application server. Some JAX-RS implementations support embeddable containers, in which a JAX-RS runtime is bootstrapped from inside a driver program for testing purposes.

High Concurrency

Developing services for reuse across multiple service consumers is a benefit of implementing service-oriented design. The considerations presented in the *Agnostic Functional Contexts* and *Concurrent Access to Service Logic* sections help illustrate the benefits of providing each service consumer exclusive access to a certain instance of the service implementation, which is true regardless of whether the service implementation is located in a JavaBean or is a stateless session EJB.

For REST services, the default lifecycle of root resource classes is per-request. A new instance of a root resource class is created every time the request URL path matches the `@Path` annotation of the root resource. With this model, resource class fields can be utilized without concern for multiple concurrent requests to the same resource. However, a resource class can also be annotated with the `javax.inject.Singleton` annotation, creating only one instance per Web application. Using the default lifecycle model for resources is recommended, unless compelling reasons arise to do otherwise.

In general, each Java component that implements a service is accessible concurrently by definition via the application server. However, installing the same service separately on different machines is also acceptable. The ultimate autonomy of a service is achieved if one instance of a service, running in its own process and controlling all of its underlying resources, serves only one specific service consumer. While inefficient from a resource utilization and maintenance perspective, requirements can dictate a service as part of a mission-critical business process which requires high performance and high transaction loads that force a dedicated instance of the service to be deployed for that specific purpose.

NOTE

Establishing environments that support high levels of concurrent access introduces scalability considerations that some IT enterprises may not be equipped to handle, relational directly to the amount of service compositions a given service participates in and the amount of access within each composition the service is subjected to.

Cloud computing platforms provide infrastructure with IT resources that can dramatically improve the extent to which the Service Autonomy principle can be applied to a service implementation, by reducing or eliminating the need for the service implementation to share or compete for resources within the enterprise boundary. For more information about scalability and elasticity as part of cloud computing environments, see the series title *Cloud Computing: Concepts, Technology & Architecture*.

SUMMARY OF KEY POINTS

- A well-defined functional boundary of a service, reflected in its contract, ensures a high degree of autonomy. However, this boundary must also be maintained in the service implementation by avoiding any dependencies on other services outside of that functional boundary.
- A service can increase autonomy by having complete control over its runtime. Java and Java EE support control over the runtime with the concept of containers, which depict virtual runtime boundaries that can be controlled individually and independently.
- Despite being accessed concurrently by multiple service consumers, services running in a Java application server can run autonomously on behalf of each service consumer. The JAX-WS and JAX-RS programming models automatically ensure that a new thread is started for each new service request.
- Highly critical services can have total autonomy by being deployed on a server for exclusive access by one particular service consumer.

7.7 Service Statelessness

Each invocation of a service operation is completely independent from any other invocation, whether by the same service consumer or any other service consumer. The Service Statelessness principle offers various benefits centered around improved scalability by which additional stateless service instances can be easily provisioned on available environments. With the advent of cloud computing, on-demand scaling out of services is considered as a natural evolutionary step for stateless services.

Many real-life business scenarios can be expressed as business processes that include automated steps, which can require manual intervention. Designing and implementing such a process requires some state to be maintained for the duration of the process. Executing an instance of the process definition forms the notion of a session or transaction across multiple service invocations. Therefore, a service implementing the execution of a business process cannot be stateless and may need to even maintain context information over extended periods.

Orchestration Infrastructure

An orchestration infrastructure, such as a WS-BPEL execution environment, will support state maintenance either by storing state in the local stack of the thread executing the process or in a permanent relational data store. A permanent relational data store ensures that a process instance can be continued after a system crash or other interruption. This style of statefulness is built into WS-BPEL and its execution environment, so there is little to be aware of when designing such a service. Designing a service that aggregates other service invocations in its implementation without utilizing WS-BPEL will require the developer to decide whether to store temporary data in a relational data store for later recovery in case of failure.

Session State

Another aspect of achieving service statelessness occurs when a service must establish some form of session with a service consumer, such as when the state is not kept in the calling logic but in the called service itself. Compare this to the shopping cart scenario in which a service consumer uses a service repeatedly to collect a set of data, which is then committed all in one final step. Java Servlets, which at the core also offer a stateless programming model, leverage the `HTTPSession` information and the concept of cookies to enable data to be stored on behalf of a certain client.

For REST services, using cookies as handles to store client state violates the stateless constraint of a REST-style architecture. Any maintenance of server-side session state is not recommended, and REST limits application state to be held on the client and not on the server.

Using cookies breaks the statelessness model as the server is expected to maintain a reference to a session and use the information in the incoming cookie for discovery. Each invocation is expected to carry the complete contextual information without any references to any previous interaction, promoting scalability where any available service instance can process this request. For idempotent requests, the failed request can be redirected to another functioning stateless service. Note that the convenience of server-side session maintenance and the associated personalization benefits are sacrificed for superior scalability and optimum resource usage.

REST services can read or write state information for various domain entities from or to databases or other persistent stores, but storing client state violates the statelessness constraint with implications for scalability and resilience. SOAP-based Web services have no such constraints and the servlet model is leveraged by JAX-WS to handle stateful interactions.

Storing State

Two methods are available for clients to store state as part of a repeated interaction with a servlet. The first allows the data to be sent back and forth with each request, and the amount of data grows with each invocation. The second method enables the servlet to maintain the state, and the servlet sends back a key to this data (the cookie). The client sends the same key along for a subsequent request, which allows the servlet to retrieve the appropriate state for this particular client. The interface used to store the data is called `javax.servlet.http.HTTPSession`.

The behavior described in maintaining a key is a well-defined part of the standard servlet programming model, and APIs are available to enable its use. The `javax.servlet.http.HTTPSession` interface offers methods for storing simple key-value pairs. JAX-WS describes a mechanism to pass a reference of the `HTTPSession` instance into the service implementation class.

A Web service implementation can be made stateful by leveraging HTTP sessions and the JAX-WS support for the sessions. This requires that the service consumer receives and reuses the cookie sent back with the response to the original HTTP POST request.

CASE STUDY EXAMPLE

Several of NovoBank's business processes require that certain forms and brochures be mailed to customers. The need for mailing each form is established in a separate part of the process, but in each case, the same back-end system is called via a Web service interface offering an `addOrderForm` operation. To reduce mailing costs, the design team makes the service establish a session with each process instance so that all orders can be bundled together. This requires the service to keep track of all forms requested. All orders are confirmed at once with the invocation of the `confirm` operation.

Example 7.40 shows an excerpt of the WSDL definition for the Order Form service.

```
<definitions targetNamespace="http://utility.services.novobank.com/"
  name="OrderFormService"
  xmlns:tns= "http://utility.services.novobank.com/"
  xmlns:xsd= "http://www.w3.org/2001/XMLSchema"
  xmlns:soap= "http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns= "http://schemas.xmlsoap.org/wsdl/">
  ...
  <xs:element name="addOrderForm" type="ns1:addOrderForm"
    xmlns:ns1="http://utility.services.novobank.com/"/>
  <xs:complexType name="addOrderForm">
    <xs:sequence>
      <xs:element name="arg0" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="addOrderFormResponse" type="ns2:
    addOrderFormResponse" xmlns:ns2="http://utility.services.
    novobank.com/"/>

  <xs:complexType name="addOrderFormResponse"/>
  <xs:element name="confirm" type="ns3:confirm" xmlns:ns3= "http://
    utility.services.novobank.com/"/>

  <xs:complexType name="confirm">
    <xs:sequence>
      <xs:element name="arg0" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  ...
  <xs:element name="confirmResponse" type="ns4:confirmResponse"
    xmlns:ns4="http://utility.services.novobank.com/"/>
  <xs:complexType name="confirmResponse"/>
  <portType name="OrderForm">
    <operation name="addOrderForm">
      <input message="tns:addOrderForm"/>
      <output message="tns:addOrderFormResponse"/>
    </operation>
    <operation name="confirm">
      <input message="tns:confirm"/>
      <output message="tns:confirmResponse"/>
    </operation>
  </portType>
</definitions>
```

Example 7.40
The WSDL definition for the Order Form service

Nothing in the contract indicates that the `addOrderForm()` operation keeps state from previous invocations by the same client in its `HTTPSession`, meaning this information must be documented elsewhere.

Example 7.41 identifies the implementation class for the Order Form service with the `addOrderForm()` operation in Java.

```
package com.novobank.services.utility;

import java.util.Vector;
import java.util.Iterator;
import javax.annotation.Resource;
import javax.jws.WebService;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;

@WebService
public class OrderForm {

  @Resource
  private WebServiceContext webServiceContext;

  public void addOrderForm(String formNumber) {
    System.out.println("Form with number "+formNumber+" was
      ordered.");
    HttpSession session = retrieveSession();
    Vector<String> formList = (Vector<String>)session.
      getAttribute("formList");
    if (formList==null) {
      formList = new Vector<String>();
   }
    formList.add(formNumber);
    session.setAttribute("formList", formList);
  }

  private HttpSession retrieveSession() {
    MessageContext messageContext = webServiceContext.
      getMessageContext();
    HttpServletRequest servletRequest =
      (HttpServletRequest)messageContext.get(MessageContext.
        SERVLET_REQUEST);
    return servletRequest.getSession();
  }
...
}
```

Example 7.41
The implementation class for the Order Form service

Each invocation of the `addOrderForm()` method retrieves the `HTTPSession` instance by using the `WebServiceContext` attribute injected via the `@Resource` annotation. This attribute provides access to the `MessageContext` for the request, which in turn stores a pointer to the servlet request object. Finally, the servlet request allows the session to be retrieved. These steps are all encapsulated in the private `retrieveSession()` method.

The list of ordered form numbers is stored in a `Vector`, which is kept in the `HTTPSession` under the name `formList`. Each time the `addOrderForm()` method is called, the new form number is added to that `Vector`.

What is missing from Example 7.41 is the implementation for the `confirm()` operation. This is where the accumulated information from previous invocations is used, as seen in Example 7.42.

```
public void confirm(String customerNumber) {
  HttpSession session = retrieveSession();
  List<String> formList =     (Vector<String>)session.
getAttribute("formList");
  if (formList==null) {
    System.out.println("No orders found.");
  } else {
    System.out.println("Confirming "+formList.size()+" orders.");
    for (String s : formList) {
     System.out.println("Order for Form " + s + ".");
    }
    session.removeAttribute("formList");
  }
}
```

Example 7.42

Access to the `HTTPSession` is provided using the same private method called `retrieveSession()` which provides the complete list of ordered forms. Note how the list is reset by setting the `formList` attribute in the `Vector` to `null` after processing.

The service consumer of this Web service must store the cookie returned from the first invocation to send along with any subsequent invocations, which JAX-WS is equipped to manage. The NovoBank team decides to provide a sample client along with their service for service consumer reuse, as shown in Example 7.43.

```
package com.novobank.services.utility.client;

import java.util.Map;
import javax.xml.ws.BindingProvider;

public class OrderFormClient {

  public static void main(String[] args) {
    OrderForm orderForm =
      new OrderFormService().getOrderFormPort();

    Map<String, Object> requestContext =
      ((BindingProvider)orderForm).getRequestContext();

    requestContext.put(BindingProvider.SESSION_MAINTAIN_PROPERTY,
      true);

    orderForm.addOrderForm("123");
    orderForm.addOrderForm("456");
    orderForm.confirm("any customer");
  }
}
```

Example 7.43
Sample client code for the Order Form service

Note that the local service proxy is cast to the `javax.xml.ws.BindingProvider` interface so that the request context can be retrieved and the `SESSION_MAINTAIN_PROPERTY` value can be set to `true`.

HTTP sessions and JAX-WS are only applicable for services with SOAP/HTTP bindings, although similar behavior in a service not accessed over HTTP can be created using the same principles. At invocation by a specific service consumer, the service can return a unique identifier to the service consumer, such as in the SOAP response header, and expect that identifier to be returned with every subsequent request. The service uses the identifier as a key into a table where the data is stored.

How the data is physically stored depends on the developer's requirements. The data stored on behalf of specific service consumers must be recoverable across a server restart persistently in a relational database, which can be accessed using JDBC or a similar mechanism.

Note that state as discussed in this section is usually transient and only exists for the duration of the interaction with the relevant service consumer. Business-relevant data, which needs to be persisted permanently, would not be stored using the mechanisms described. Business data should be stored using data access APIs, such as JDBC, or persistence frameworks, such as JPA or Hibernate.

SUMMARY OF KEY POINTS

- Many scenarios require data to be stored beyond a single invocation of a service in the calling service, such as from either within a WS-BPEL process or in the called service. Data is often stored for long durations.
- JAX-WS provides a mechanism to utilize the `HTTPSession` object to store temporary state on behalf of a specific service consumer.
- Business-relevant data that must be stored permanently should be handled through entity services that explicitly wrap the handling of such data.

7.8 Service Discoverability

The two primary aspects of the Service Discoverability principle are discovery at design-time (which promotes service reuse in a newly developed solution), and discovery at runtime (which involves resolving the appropriate endpoint address for a given service or retrieving other metadata). Even though the information can be physically stored in one place, the way in which the information is accessed in each scenario varies.

Design-Time Discoverability

At design-time, it is important for project teams to be able to effective identify the existence of services that contain logic relevant to the solution they plan to build. This way they can either discover services that they can reuse or confirm that new service logic they plan to build does not already exist. For example, a service is designed to address an Order Management business process for which customer information is required. The service designer must investigate whether a `Customer` data type already exists, and if so, determine whether it meets the requirements for the Order Management Process service. If the data sent into the newly designed service is missing information, the designer can also check whether an entity service encapsulating all data access to this type of data exists. Additional customer information required can be built or retrieved via a Customer entity service.

During the design of a new service, several types of artifacts must be evaluated for reuse directly on the new service's interface and within the service via some kind of aggregation. This includes non-functional aspects, which may not be expressed in machine-readable form. Meta information, such as performance and reliability of existing services, can influence whether a service is reusable in a new context.

Code can exist for existing data type definitions. JAXB, for example, defines a mapping between XML schema and Java. Java code can be directly generated from a schema definition and reused wherever that particular data type is used.

All of the relevant information must be available to the designer during design-time, ideally via the development environment directly. This relevant information includes the artifacts themselves, such as the WSDL, XML schema, and Java source code as well as relationships and dependencies between them. These dependencies must be documented thoroughly, as part of the service profile so that a complete metamodel of the service exists once the design is complete.

Since the design of a service is manually performed by a service designer, this information must be accessible and searchable by humans. How this feature is enabled depends on the registry type used and the access mechanisms offered, without depending on the programming language used to implement services or on the runtime platform that the service will run on. The underlying mechanism should offer a way to control access to the information and support versioning of artifacts.

Runtime Discoverability

Runtime service discovery refers to the ability of software programs to programmatically search for services using APIs exposed by the service registry. Doing so allows for the retrieval of the physical location or address of services on the network. Because services may need to be moved from one machine to another or perhaps redundantly deployed on multiple machines, it may be advisable for service addresses not to be hardcoded into the service consumer logic.

For SOAP services using JAX-WS, the location of the target service is included in the `<port>` element of the WSDL definition by default. In turn, the location of the WSDL file is automatically added to the generated `...Service` class and passed to the tooling, or wsimport, that generates the appropriate service consumer artifacts. Pointing to the local WSDL file in the `filesystem` will result in a service consumer that cannot be moved to a different environment, because functionality depends on that particular location for the WSDL file and the endpoint address contained by the WSDL file.

Using the URL and appending `"?wsdl"` to the service is a more flexible approach. For example, a service can be located at `myhost.acme.com:8080/account/account`, in which case the WSDL definition can be retrieved via the URL `http://myhost.acme.com:8080/account/account?wsdl`. During the installation and deployment of a service, the endpoint information in this WSDL file is updated to reflect the real address of the hosting system.

The address points to a fixed location for the WSDL file despite now residing on the network, meaning the address of any target services should always be resolved separately from the generated service consumer code. JAX-WS provides a way to set a target endpoint address on a proxy instance at runtime that utilizes the `javax.xml.ws.BindingProvider` interface. Each proxy implements this interface and allows properties to be dynamically set on an exchange between a service consumer and service, with the endpoint address being one of the pre-defined properties. The code in Example 7.44 illustrates how a service consumer sets a new endpoint address on a service proxy before invocation.

```
String endpointAddress = ...; //retrieve address somehow
OrderForm orderForm =
  new OrderFormService().getOrderFormPort();
  java.util.Map<String, Object> requestContext = (javax.xml.
    ws.BindingProvider)orderForm).getRequestContext();
  requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
    endpointAddress);
orderForm.addOrderForm("123");
```

Example 7.44

The WSDL location can alternatively be set for a service at runtime instead of the endpoint address, to maintain the address of the service together with the rest of the service contract in its WSDL file and not in multiple locations, as shown in Example 7.45.

```
URL wsdlLocation = ...; //retrieve WSDL location
OrderFormService orderFormService =
  new OrderFormService(wsdlLocation, new QName("http://utility.
  services.novobank.com/", "OrderFormService"));
OrderForm orderForm = orderFormService.getOrderFormPort();
orderForm.addOrderForm("123");
```

Example 7.45

In addition to the lookup of a service endpoint address, other artifacts can be used by service consumers at runtime to identify an appropriate service and build the applicable request message. The JAX-WS `Dispatch` API allows a service request to be completely built at runtime. Theoretically, a service consumer could look up a WSDL definition at runtime, read its portType and XML schema definitions, and build the right messages from scratch. However, building a service request completely at runtime is impractical in a real-life scenario, as the service will likely perform poorly and require plenty of tedious coding.

Non-functional characteristics of a service are other examples of information that a service consumer can discover at runtime. For example, assume that two physical instances of a service exist on both a slower machine and a faster machine. The service consumer can retrieve this information and select the appropriate service to be used, depending on the business context of the call. For instance, a silver customer is routed to the slower machine, whereas a gold customer is routed to the faster machine. The service is still implemented in both invocations, as the differentiation in routing is a non-functional characteristic.

API endpoints are unnecessary in a REST-based system because no RPC-style invocation is involved. Recall that URI-based addressability, like statelessness, is a formal REST constraint. Resources that offer services are the fundamental entities and must be discoverable through URIs before clients can invoke operations on them. The hypermedia constraint, if followed accurately, requires only the URI of the root resource to be provided to the service consumer.

Various operations on the root resource will publish URIs of related resources, which can then be used by the service consumers to reduce the amount of foreknowledge required. However, service consumers would still require knowledge of the semantics associated with the URIs to be able to make meaningful use of them, which makes this approach impractical in real-life application.

Service Registries

Information can be retrieved by a service consumer at runtime with the service registry. As the following methods are inapplicable to REST services, the remainder of this discussion will only apply to SOAP-based Web services. Storing information about WSDL locations and endpoint addresses in a file accessible through a proprietary API at runtime is a retrieval mechanism appropriate for small environments. However, the format in which the information is stored, such as XML or a character-delimited format, must be defined, and the location of this file must always be maintained throughout the enterprise for easy accessibility.

Alternatively, storing the information in a relational database allows for remote access and query using a standard language, such as SQL, although a proprietary relational model for this information must still be invented. Other options include leveraging the JNDI or LDAP to serve the same purpose.

In the early days of Web services, a platform-independent standard describing how to uniformly store, find, and retrieve business and technical information about services was developed as the UDDI registry, which offers a query and publish interface. As a Web service, the UDDI registry allows the API to be described by the WSDL so that any Web service-capable service consumer can access a UDDI registry by generating a proxy from the standard WSDL. For accessing registries at runtime, many IDEs such as Eclipse's Web Tools Platform have built-in support for existing UDDI registries.

NOTE

Discoverability processing can be delegated into the ESB, where tasks such as service lookup can be handled centrally and uniformly. The service lookup logic does not then clutter the service consumer, which can instead focus on the business purpose.

SUMMARY OF KEY POINTS

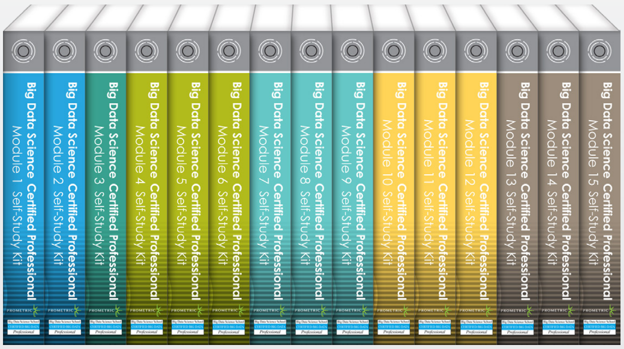
- For Web services, JAX-WS provides runtime APIs to set the endpoint address of a target service. UDDI defines a standardized way for storing service meta information, which includes access via a Web service-based API.
- REST services use embedding-related resource links in resource representations to leverage the hypermedia constraint and lead the client through a discovery of networked resources.

Arcitura
Big Data School

Vendor-Neutral Big Data Training & Certification

15 Course Modules ● 15 Exams ● 5 Certifications

The Big Data Science Certified Professional (BDSCP) program from the Arcitura Big Data Science School is dedicated to excellence in the fields of Big Data science, analysis, analytics, business intelligence, technology architecture, design and development, as well as governance. A collection of courses establishes a set of vendor-neutral industry certifications with different areas of specialization. Founded by best-selling author, Thomas Erl, this curriculum enables IT professionals to develop real-world Big Data science proficiency. Because of the vendor-neutral focus of the course materials, the skills acquired by attaining certifications are applicable to any vendor or open-source platform.



Certified Big Data Science Professional
Certified Big Data Scientist
Certified Big Data Engineer
Certified Big Data Architect
Certified Big Data Governance Specialist