11
Designing the Service Contract

Service contracts provide the glue that enables us to assemble disparate pieces of software or services into complete, composite applications. If we are to build a sustainable solution, that is, one that will achieve our goals of improved agility, reuse, and interoperability, then a careful design of the service contract is crucial.

The contract of a web service is made up of the following technical components:

- **WSDL Definition**: Defines the various operations which constitute a service, their input and output parameters, and the protocols (bindings) it supports.
- **XML Schema Definition** (**XSD**): Either embedded within the WSDL definition or referenced as a standalone component, this defines the XML elements and types which constitute the input and output parameters.
- **WS-Policy Definition**: An optional component that describes the service's security constraints, quality of service, and so on.

Additionally, the service contract may be supported by a number of non-technical documents, which define areas such as service-level agreements, support, and so on.

From a contract design perspective, we are primarily interested in defining the XML Schema and the WSDL definition of the service. This chapter gives guidance on best practices in the design of these components as well as providing strategies for managing change, when it occurs. We leave the discussion on defining security and management policies to *Chapter 21*, *Defining Security and Management Policies*.

Using XML Schema to define business objects

Each business service acts upon one or more business objects. Within SOA, the data that makes up these objects is represented as XML, with the structure of the XML being defined by one or more XML Schemas. Thus the definition of these schemas forms a key part in defining the overall service contract.

To better facilitate the exchange of data between services, as well as achieve better interoperability and reusability, it is a good practice to define a common data model, often referred to as the "canonical data model", which is used by all services.

As well as defining the structure of data exchanged between components, XML is also used in all components of the SOA Suite. For example, it defines the structure of variables in BPEL and provides the vocabulary for writing business rules and transforming data via XSLT. So it is important that our XML model is well thought out and designed.

Modeling data in XML

When designing your XML data model, a typical starting point is to base it on an existing data model, such as a database schema, UML model, or EDI document.

While this is a perfectly legitimate way to identify your data elements, the crucial point in this approach is how you make the transition from your existing model to an XML model. Very often, the structure of the existing model is directly replicated in the XML model, resulting in poor design.

Data decomposition

In order to produce an effective model for XML, it's worth taking a step back to examine the core components that make up an XML document. Consider the following XML fragment:

```
<order>
  <orderNo>123456</orderNo>
   <shipTo>
      <name>
         <title>Mr</title>
         <firstName>James</firstName>
         <lastName>Elliot</lastName>
      </name>
      <address>
```

```
            <addressLine1>7 Pine Drive</addressLine1>
            <addressLine2></addressLine2>
            <city>Eltham<city>
            <state>VIC</state>
            <zip>3088</zip>
            <country>Australia</country>
        </address>
    </shipTo>
</order>
```

If we pull out the raw data from this, we would end up with:

```
123456 Mr, James, Elliot, 7 Pine Drive, , Eltham, VIC, 3088.
Australia.
```

By doing this, we have greatly reduced our understanding of the data. XML, through the use of tags, gives meaning to the data, with each tag describing the content of the data it contains. Now this may seem an obvious point, but too often, by failing to sufficiently decompose our data model, we are doing exactly this, albeit within an XML wrapper. For example, another way of modeling the previous piece of XML is as follows:

```
<order>
    <orderNo>123456</orderNo>
    <shipTo>
        <name>Mr James Elliot</name>
        <address>7 Pine Drive, Eltham, VIC, 3088, Australia</address>
    </shipTo>
</order>
```

With this approach, we have again reduced our understanding of the data. So you should always look to decompose your data to the appropriate level of granularity. If in doubt go for the more granular model, as it's a lot simpler to convert data held in a granular model to a less granular model than the other way round.

Data hierarchy

Another key component of our data model is the relationship between the different elements of data, which is defined by the composition or hierarchy of the data. If from our example fragment we take the element `<city>Eltham<city>`, on its own it does not signify much, as we have provided insufficient context to the data.

However, in the context of `<order><shipTo><address><city>`, we have a far clearer understanding of the data.

A common mistake is to use a flat structure, and then name the elements to compensate for this, for example, changing the name to `<order_shipTo_address_city>`. While this provides more context than just `<city>`, it introduces a whole set of other problems, including:

- It makes your XML far harder to read, as well as more bloated.
- The relationships are no longer visible to your XML parser. This makes XPath manipulation, XSLT mappings, and so on a lot more complex and onerous.
- It reduces the opportunity for reuse; for example, each address element will have to be redefined for every single context in which it is used. This is likely to lead to inconsistent definitions as well as make changes harder to manage.

If you see elements named in this fashion, for example, `<a_b_c>`, `<a_b_d>`, it's a pretty good clue that the schema has been poorly designed.

Data semantics

Another key to our data model is the semantics of our data. Looking at the preceding example, it is obvious what the `<state>` element contains, but the exact format of that data is not as obvious; that is, it could be Victoria, VIC, Vic, 0, and so on.

While different target systems will have different requirements, it is important that a set of semantics are agreed upon for the canonical model, so that these differences can be isolated in the Virtual Services layer.

While semantics can be enforced within our XML Schema through the use of facets such as `enumeration`, `length`, `pattern`, and so on, this is not always the best approach. This is an area we examine in more detail in *Chapter 13, Building Validation into Services.*

Using attributes for metadata

A common debate is when to model XML data using elements and when to use attributes, or whether attributes should be used at all.

Elements are more flexible than attributes, particularly when it comes to writing extensible schemas, as you can always add additional elements (or attributes) to an element. However, once an attribute has been defined, it can't be extended any further.

One approach is to use attributes for metadata and elements for data. For example, on some of our query-based operations (for example. `getSellerItems`, `getBuyerItems`) we have defined two attributes, `startRow` and `endRow`, which are used to control which portion of the result set is returned by the query.

Schema guidelines

It's important to have a clear set of guidelines for schema design. This not only warrants that the best practice is followed, it also ensures that schemas are created consistently, making them more reusable, easier to combine, simpler to understand, and easier to maintain.

Element naming

Consistent naming of elements within an XML Schema will ensure that schemas are easier to understand, and reduce the likelihood of error due to names being spelt differently within a different context.

Name length

While there is no theoretical limit on the length of an element or attribute name, you should try and limit them to a manageable length. Overly long names can reduce the readability of a document as well as make them overly bloated, which, in extreme cases, could have performance implications.

We tend to try and limit names to a maximum of 15 characters; now this may not always be possible, but there are a number of simple strategies.

Compound names

For element and attribute names that are made up of multiple words, we follow the practice of capitalizing the first letter of each word (often called camel case). For example, we would write the name 'shipping address' as `shippingAddress`.

Another approach is to use a separator, that is, a hyphen or an underscore between words. Personally, we don't find this as readable as well as resulting in marginally longer names. Whichever approach you use, you should ensure that you do so consistently.

For types (for example, `complexType` and `simpleType`), we follow the same convention but prefix the name with a `'t'` in order to easily distinguish it from elements and attributes.

An alternative convention is for type names to have the word `Type'` at the end. However, we are not as keen on this convention as it can lead to type names finishing in `TypeType`. For example we have the element `auctionType`; using this convention; its type name would be `auctionTypeType`.

Naming standards

We also recommend the use of a defined dictionary for commonly used words, abbreviations, and so on to ensure that they are used in a consistent fashion. Areas that should be considered include:

- **Synonyms**: For names that have multiple potential options, for example, Order and Purchase Order, the dictionary should define the standard term to be used.
- **Abbreviations**: For words that are used commonly, it makes sense to define a standard abbreviation. For example, we may use `address` on its own, but when combined with another word (for example, shipping), we use its abbreviation to get `shippingAddr`.
- **Context Based Names**: When a name is used in context, don't repeat the context in the name itself. For example, rather than using `addressLine1` inside an `address` element, we would use `line1`.

In some cases, this is not always pragmatic, particularly if it reduces clarity in the meaning of the element. For example, if within the context of name, you have the element `family`, then this is not as clear as using `familyName`. So a degree of common sense should be used.

- **Generic Names**: As far as possible, use generic names. For example, avoid using specific company or product names, as this will result in more reusable models and also reduce the likelihood of change.

A sample of the oBay dictionary is shown in the following table:

Standard term	Abbreviation	Synonyms
address	addr	
amount	amt	cost, price, fee
description	desc	
end	end	finish, stop
id	id	number, identifier
item	item	product
max	max	ceiling, maximum, top
min	min	least, lowest, minimum
order	ord	purchase order
start	start	effective, begin
status	status	state
user	usr	client, customer

Namespace considerations

Namespaces are one of the biggest areas of confusion with XML Schemas, yet in reality, they are very straightforward. The purpose of a namespace is just to provide a unique name for an element, type, or attribute, thus allowing us to define components with the same name.

For example, the element `Glass`, will have a different definition to a company that sells windows as opposed to one that runs a bar. The namespace allows us to uniquely identify each definition, so that we can use both definitions within the same instance of an XML document, as well as understand the context in which each element is being used.

If you're familiar with Java, then a namespace is a bit like a package name, that is, you can have multiple classes with the same name, but each one would be defined in a separate package.

One feature of namespaces is that they have a degree of flexibility in how you apply them, which then impacts how you construct and interpret an instance of an XML document. This is often a cause of confusion, especially when they are used inconsistently across multiple schemas.

So it's critical that you define a standard approach to namespaces before defining your canonical model.

Always specify a target namespace

Unless you are defining a chameleon schema (seen later in the chapter) always specify a target namespace.

Default namespace

When defining a schema, you have the option of defining a default namespace. If you do, we would recommend setting the default namespace to be equal to the target namespace. The advantage of this approach is that you only prefix elements, types, and attributes that are defined externally to the schema (that is, anything that is not prefixed is defined locally).

An alternative approach is not to use a default namespace, so that all elements require a prefix. This can often be clearer when combining many schemas from multiple namespaces, especially if you have similarly named elements.

Qualified or unqualified element names

When constructing an XML instance based on an XML Schema, we have the option as the schema designer to decide whether each element should be qualified, that is, have a prefix that identifies the namespace of where the element is defined, or have no prefix, that is, it is unqualified and the origin of the namespace is hidden within the instance.

The approach you take is often a question of style. However, each has its own advantages and disadvantages, particularly when you create XML instances that are defined across multiple namespaces. Take the schema definition for the element `<address>`, as shown in the following code snippet:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns="http://rubiconred.com/obay/xsd/common"
            targetNamespace="http://rubiconred.com/obay/xsd/common"
            elementFormDefault="qualified or unqualified">

    <xsd:element name="address" type="tAddress"/>
    <xsd:complexType name="tAddress">
        <xsd:sequence>
            <xsd:element name="addressLine1" type="xsd:string"/>
            <xsd:element name="addressLine2" type="xsd:string"/>
            <xsd:element name="city" type="xsd:string"/>
            <xsd:element name="state" type="xsd:string"/>
            <xsd:element name="zip" type="xsd:string"/>
            <xsd:element name="country" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

If we chose **unqualified** elements, then an instance of this schema would look as shown in the following code snippet:

```
<cmn:address xmlns:cmn="http://rubiconred.com/obay/xsd/common">
   <addressLine1>7 Pine Drive</addressLine1>
   <addressLine2></addressLine2>
   <city>Eltham<city>
   <state>VIC</state>
   <zip>3088</zip>
   <country>Australia</country>
</cmn:address>
```

However, if we chose to use **qualified** elements, our XML instance would now appear as shown in the following code snippet:

```
<cmn:address xmlns:cmn="http://rubiconred.com/obay/xsd/common">
   <cmn:addressLine1>7 Pine Drive</cmn:addressLine1>
   <cmn:addressLine2></cmn:addressLine2>
   <cmn:city>Eltham<cmn:city>
   <cmn:state>VIC</cmn:state>
   <cmn:zip>3088</cmn:zip>
   <cmn:country>Australia</cmn:country>
</cmn:address>
```

With unqualified namespaces, the XML instance loses most of its namespace declarations and prefixes, resulting in a slimmed down and simpler XML instance that hides the complexities of how the overall schema is assembled.

The advantage of using qualified namespaces is that you can quickly see what namespace an element belongs to. As well as removing any ambiguity, it provides the context in which an element is defined, giving a clearer understanding of its meaning.

Whichever approach you use, it's important to be consistent, as mixing qualified and unqualified schemas will produce instance documents where some elements have a namespace prefix and others don't. This makes it a lot harder to manually create or validate an instance document, as the author needs to understand all the subtleties of the schemas involved, making this approach more error-prone.

Another consideration over which approach to use is whether you are using local or global element declarations, as unqualified namespaces only apply to local elements. Having a mixture of global elements and local unqualified elements in your schema definition will again produce instance documents where some elements have a namespace prefix and others don't, with the same issues mentioned earlier.

A final consideration is whether you are using default namespaces. If you are then you should use qualified names, as unqualified names and default namespaces don't mix.

As we recommend, using both global elements (see later for the reason why) and default namespaces, we would also recommend using qualified namespaces.

Qualified or unqualified attributes

Like elements, XML Schema allows us to choose whether an attribute is qualified or not. Unless an attribute is global (that is, declared a child of `schema` and thus can be used in multiple elements), there is no point in qualifying it.

The simplest way to achieve this is to not specify the `form` and `attributeFormDefault` attributes. This will result in globally declared attributes being prefixed with a namespace and locally declared attributes will have unqualified names.

Namespace naming conventions

We also recommend defining a namespace naming convention, as this will provide greater clarity as well as simplify the overall governance of assets. In the case of oBay, our namespaces use the following convention:

```
http://<domain>/<sub-domain>/<namespace-type>/<subject_area>
```

Here, `obay` is a sub-domain within `rubiconred.com`. The `<namespace-type>` defines the type of component (for example, schema, service, and so on) to which the namespace applies.

So within our canonical model, we have defined several namespaces, including:

```
http://rubiconred.com/obay/xsd/account
http://rubiconred.com/obay/xsd/auction
http://rubiconred.com/obay/xsd/common
```

As part of your naming standards, you should also define standard namespace prefixes for each namespace in your canonical model.

Global versus local

A component (element, simple type, or complex type) is considered global if it's defined as a child of the schema element. If defined within another component, it's considered local. Consider the following fragment of XML:

```
<shipTo>
    <name>
        <title>Mr</title>
        <firstName>James</firstName>
        <lastName>Elliot</lastName>
    </name>
    <address>
        <addressLine1>7 Pine Drive</addressLine1>
        <addressLine2></addressLine2>
        <city>Eltham<city>
        <state>VIC</state>
        <zip>3088</zip>
        <country>Australia</country>
    </address>
</shipTo>
```

One way of implementing its corresponding schema would be to design it to mirror the XML, for example:

```
<xsd:element name="shipTo">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="name">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="title"     type="xsd:string"/>
                  <xsd:element name="firstName" type="xsd:string"/>
                  <xsd:element name="lastName"  type="xsd:string"/>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="address">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="line1"     type="xsd:string"/>
                  <xsd:element name="line2"     type="xsd:string"/>
                  <xsd:element name="city"      type="xsd:string"/>
                  <xsd:element name="state"     type="xsd:string"/>
                  <xsd:element name="zip"       type="xsd:string"/>
                  <xsd:element name="country"   type="xsd:string"/>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

Using this approach, only the `shipTo` element is declared globally and thus is reusable; no other elements or types either within this schema or another schema can make use of the elements or types declared inside the `shipTo` element.

Another way of defining the schema would be as shown in the following code snippet:

```
<xsd:element name="shipTo">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="name"/>
         <xsd:element ref="address/>
      </xsd:sequence>
   </xsd:complexType>
```

```
</xsd:element>

<xsd:element name="name">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="title"     type="xsd:string"/>
         <xsd:element name="firstName" type="xsd:string"/>
         <xsd:element name="lastName"  type="xsd:string"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>

<xsd:element name="address">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="line1"     type="xsd:string"/>
         <xsd:element name="line2"     type="xsd:string"/>
         <xsd:element name="city"      type="xsd:string"/>
         <xsd:element name="state"     type="xsd:string"/>
         <xsd:element name="zip"       type="xsd:string"/>
         <xsd:element name="country"   type="xsd:string"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

With this approach `shipTo`, `name`, and `address` are globally declared. Therefore, the elements `name` and `address` are also reusable now.

You could always go a step further and separately define all the simple types such as title, first name, and so on as global elements.

The temptation may be to define elements you wish to reuse within your schema as global and have the rest as local definitions. However, you should consider the following point:

- Any element you may wish to use as a parameter for a web service operation must be globally defined
- BPEL variables can only be declared for global elements, not local elements

As at the point of schema definition, it's not always easy to determine where an element may need to be reused. We would recommend always declaring your components as global.

Elements versus types

A common dilemma is whether to use elements or types to define global components. Types tend to be more flexible, in that once you've defined the type, it can be reused to define multiple elements of the same type.

Also, once you have defined a type, you can easily use it to define an element. In the following example, we have remodeled the above schema to separately define the types and then use them to define the elements. As a result, we have slightly fewer lines of XML as well as a more flexible model:

```
<xsd:element name="shipTo" type="tShipTo">
<xsd:complexType name="tShipTo">
   <xsd:sequence>
      <xsd:element ref="name"/>
      <xsd:element ref="address/>
   </xsd:sequence>
</xsd:complexType>

<xsd:element name="name" type="tName">

<xsd:complexType name="tName">
   <xsd:sequence>
      <xsd:element name="title"     type="xsd:string"/>
      <xsd:element name="firstName" type="xsd:string"/>
      <xsd:element name="lastName"  type="xsd:string"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:element name="address" type="tAddress">

<xsd:complexType name="tAddress">
   <xsd:sequence>
      <xsd:element name="line1"     type="xsd:string"/>
      <xsd:element name="line2"     type="xsd:string"/>
      <xsd:element name="city"      type="xsd:string"/>
      <xsd:element name="state"     type="xsd:string"/>
      <xsd:element name="zip"       type="xsd:string"/>
      <xsd:element name="country"   type="xsd:string"/>
   </xsd:sequence>
</xsd:complexType>
```

With BPEL 1.1, you can only create variables based on global elements, NOT global types.

When reusing components from other namespaces, refer to the **element** that is defined against the type (as highlighted previously), rather than using the type directly. Otherwise, the namespace of the top element will take on the namespace of the schema that is reusing the type.

Finally, we recommend that you use different names for elements and complex types. Although the XML Schema specification allows for an element and a type to have the same name, this can cause issues for some tools. So for our purposes, we prefix all types with a lower case `"t"` to indicate that it's a type.

Partitioning the canonical model

When building your first SOA application, it's very easy to fall into the trap of defining a single schema that meets the specific needs of your current set of services. However, as each project develops its own schema, it will often redesign its own version of common elements. This not only reduces the opportunity for reuse, but makes interoperability between applications more complicated as well as increases the maintenance burden.

The other common pitfall is to design a single, all encompassing schema that defines all your business objects within an organization. There are two issues with this approach. First, you could end up "boiling the ocean", that is, you set out to define every single business object with the organization and the project never starts because it's waiting for the model to be completed.

Even if you take a more iterative approach, only defining what's required upfront and extending this schema as new applications come on line, you very quickly end up with the situation where every application will become dependent on this single schema. Change often becomes very protracted, as a simple change could potentially impact many applications. The end result is a strict change control being required, often resulting in protracted time frames for changes to be implemented, which is not exactly an agile solution.

The approach, of course, lies somewhere in the middle, and that is to partition your data model into a set of reusable modules, where each module is based on a logical domain. For example, in oBay, we have defined the following schemas:

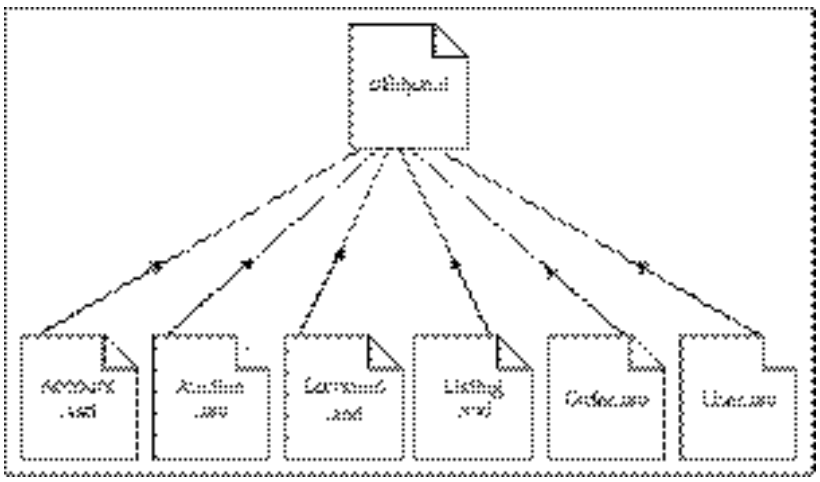
- `Account.xsd`: Defines every business object specific to a financial account, that is, a record of all debit and credit activities related to a specific user
- `Auction.xsd`: Defines all business objects specific to an auction
- `Order.xsd`: Defines all business objects specific to order fulfillment, that is, from the placement of an order through to its final shipment and delivery
- `User.xsd`: Defines all business objects specific to a user
- `Common.xsd`: This schema is used to define common objects, such as name, address, credit card, which are used by multiple domains, but have no obvious owner

Once we have partitioned our data model, we need to decide on our strategy for namespaces. There are a number of potential approaches, which we cover below.

Single namespace

With this approach, we have a single target namespace which is used by all schema definitions. We typically have a single master document which uses the `xsd:include` element to combine the various schema documents into a single schema.

This approach is illustrated below, where we have a master "oBay" schema that includes all of our other schemas:



The advantage of this approach is that it keeps it simple, as we only have a single namespace and corresponding prefix to worry about.

The disadvantage is that we have taken a single schema and just broken it up into multiple manageable files. But apart from this, we still have all the other disadvantages that we outlined previously when creating a single master model.

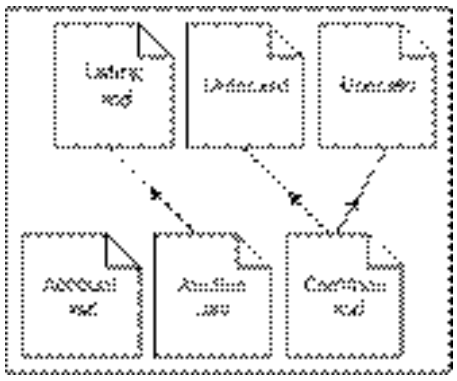
The major disadvantage is that if we change any single document, we change the entire model. This would result in a new version of the entire model, and thus for us, potentially having to update every single service implemented to date.

Multiple namespaces

The other approach, of course, is to define a namespace for each individual domain; each schema can then reference a definition in other schema's by making use of the `xsd:import` element. This is the approach we have taken for oBay and is illustrated as follows:

With this approach, we have no master schema, thus services only need to import the parts of the canonical model which are relevant to them. Whilst with the single namespace approach, you will typically end up being required to import the entire schema.

Another advantage of this approach is that it allows different groups to be responsible for each namespace, and for each namespace to evolve to a certain extent, independent of others.



The drawback to this approach is that instance documents become more complex, as they will need to reference multiple namespaces. To prevent this from becoming a problem, it's important to partition your data model into sensible domains and also resist the urge to over partition it and end up with too many namespaces.

Separate common objects into their own namespaces

Common objects, which are used across multiple namespaces, should be created in their own namespace. For example, the address element is used across all domains. If we were to create it in the order namespace, we would be forcing all our other schemas to import the order schema, which would unnecessarily complicate our XML instances. The issue would become more acute if common object definitions were sprinkled across multiple namespaces.

Using WSDL to define business services

A service, as defined by a WSDL document, is made up of two parts. Firstly, there is the abstract part, which defines the individual operations that make up a service, the messages that define the parameters for the operations, and the types which define our XML data types used by our messages.

The second part of the WSDL document defines the binding, that is, how to physically encode the messages on the wire (for example, SOAP), the transport protocol on the wire (for example, HTTP), and also the physical location or endpoint of the service (for example, its URL).

Ideally, we should only be concerned with designing the abstract part of the WSDL document, as the runtime binding should be more of a deployment detail. However, the reality is that the style of binding has implications for how we design the abstract components if we want to ensure interoperability between service providers and consumers.

By far, the most common binding for a web service is SOAP over HTTP. However, this comes in a number of different varieties, as we can specify whether the invocation method adopts a Remote Procedure Call (RPC) style or a document style binding (that is, a more message-oriented approach). We also have a choice as to whether the SOAP message is encoded or literal. This gives us four basic combinations, that is, RPC/encoded, RPC/literal, Document/encoded, and Document/literal.

It is generally accepted that for the purposes of interoperability, Document/literal is the best practice. However, the Document/literal style has some drawbacks.

Firstly, not all Document/literal services are WS-I compliant, as WS-I recommends that the SOAP body contains only a single child element within the SOAP body. However, Document/literal allows you to define WSDL messages containing multiple parts, where each part is manifested as a child element within the SOAP body.

Another minor issue with document/literal is that it doesn't contain the operation name in the SOAP message, which can make dispatching of messages difficult in some scenarios and can also make debugging complicated when monitoring SOAP traffic, particularly when multiple operations contain the same parameters.

Use Document (literal) wrapped

Document Wrapped is a particular style or use of Document/literal that addresses these issues. With this approach, the request and response parameters for an operation are 'wrapped' within a single request and response element.

The request wrapper must have the same name as the corresponding operation, while the name of the response wrapper is derived by appending 'Response' to the operation name.

This ensures that the SOAP body only contains a single nested element whose name matches that of the operation.

These wrapping elements must be defined as elements, not complex types. While WSDL allows either, the use of `complexTypes` is not WS-I compliant.

This approach ensures that you have WS-I compliant messages.

A document wrapped web service looks very similar to a RPC/literal style, as they both produce a SOAP message containing a single nested element matching the name of the operation within the `soap:Body`.

Building your abstract WSDL document

Once we have standardized on the document wrapped pattern, we can define the abstract part of our WSDL contract at this stage, without having to worry about the actual binding.

WSDL namespace

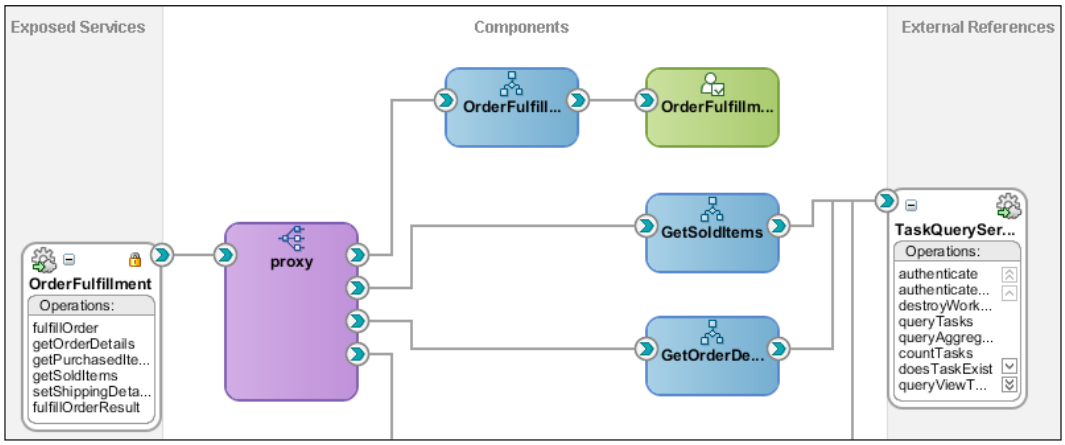
As with our schema definitions, we need to define a namespace for our business service. Here we would recommend defining a different namespace for each service. This should also be different from the namespaces used within your canonical model.

Defining the 'wrapper' elements

In the 10gR3 release of SOA Suite, we used to advocate defining the wrapper elements within the WSDL document of the web service, as separating those out into a standalone schema provided little value.

With 11gR1, we recommend a **different** approach—to define the wrapper elements in a separate schema in their own namespace, which we then import into our WSDL document. This allows us to reuse these wrapper elements within our composite.

Recall the basic composite design pattern that we introduced in the previous chapter, where we use a Mediator as the entry point for a composite, as shown in the following screenshot:



With this pattern, the Mediator implements the abstract WSDL for our service, and acts as a proxy responsible for routing requests from the service consumer to the appropriate components within the composite.

Each of these components, in turn, implements one or more of the operations defined in our abstract WSDL. By separating out the wrapper elements into a separate schema, we can reuse these when defining the WSDL for each of these components.

This ensures that within our composite, we have a single consistent definition of our message types. This makes it simpler to manage change as well as ensuring our proxy is not required to perform any transformations.

Whichever approach you follow, resist the temptation to define your wrapper elements within your canonical model, as doing so will pollute your model as well as make changes harder to manage.

Defining a schema for our wrapper elements

When defining an XML Schema for our wrapper elements, we need to import the relevant parts of our canonical model, and for each imported schema, define its namespace and corresponding prefix.

For example, we have defined our wrapper elements for the `OrderFulfillment` service in the schema `OrderFulfillmentEBM.xsd`. This imports the `Order.xsd` schema, defined in the namespace '`http://rubiconred.com/obay/xsd/order`', as highlighted in the following code:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://rubiconred.com/obay/ebm/OrderFulfillment"
    xmlns="http://rubiconred.com/obay/ebm/OrderFulfillment"
    xmlns:ord="http://rubiconred.com/obay/xsd/order"
    elementFormDefault="qualified">

   <xsd:import namespace="http://rubiconred.com/obay/xsd/order"
               schemaLocation="Order_v1_0.xsd" />

     <!-- Wrapper Elements Defined Here -->

</xsd:schema >
```

Next, we can define our wrapper elements, so for the `setShippingInstruction` operation within the `OrderFulfillment` service, we have defined the following:

```
<xsd:element name="setShippingInstruction"
             type="tSetShippingInstruction"/>

<xsd:element name="setShippingInstructionResponse"
            type="tSetShippingInstructionResponse"/>

<xsd:complexType name="tSetShippingInstruction">
   <xsd:sequence>
       <xsd:element ref="ord:orderNo"/>
       <xsd:element ref="ord:shippingDetail" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="tSetShippingInstructionResponse">
  <xsd:sequence>
       <xsd:element ref="ord:order"/>
  </xsd:sequence>
</xsd:complexType>
```

Importing our wrapper elements

The next step is to import the schema containing the wrapper elements into our WSDL; we do this by using an `import` statement within the `types` section of our WSDL, as shown in the following code snippet:

```
<types>
  <xsd:schema elementFormDefault="qualified">
    <xsd:import schemaLocation="OrderFulfilmentEBM_v1_0.xsd"
        namespace="http://rubiconred.com/obay/ebm/OrderFulfilment"/>
    …
  </xsd:schema>
<types>
```

Before we can refer to the wrapper elements contained within this schema, we must also declare its namespace and corresponding prefix within the `definitions` element of the WSDL, as highlighted in the following code snippet:

```
<definitions name="OrderFulfillment"
    targetNamespace="http://rubiconred.com/obay/svc/OrderFulfillment"
    xmlns:tns="http://rubiconred.com/obay/svc/OrderFulfillment"
    xmlns:ebm="http://rubiconred.com/obay/ebm/OrderFulfillment"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Defining the 'message' elements

Once we have defined and imported our wrapper elements, it's pretty straightforward to define our message elements. We should have one message element per wrapper element. From a naming perspective, we use the same name for the message element as we did for our wrapper elements. So for our `setShippingInstruction` operation, we have defined the following message elements:

```
<message name="setShippingInstruction">
   <part name="payload" element="ebm:setShippingInstruction"/>
</message>

<message name="setShippingInstructionResponse">
  <part name="payload" element="ebm:setShippingInstructionResponse"/>
</message>
```

Defining the 'PortType' Element

The final component of an abstract WSDL document is to define the `portType` and its corresponding operations. For our `OrderFulfillment` service, we have defined the following:

```
<portType name="orderFulfilment">
    <operation name="setShippingInstruction">
        <input message="tns:setShippingInstruction"/>
        <output message="tns:setShippingInstructionResponse"/>
    </operation>
    <operation name="submitInvoice">
        <input message="tns:submitInvoice"/>
        <output message="tns:submitInvoiceResponse"/>
    </operation>
    …
</portType>
```

Note that for the sake of brevity, we have only listed two operations; for the full set, please refer to `OrderFulfilment.wsdl` contained within the sample application.

Using XML Schema and the WSDL within SOA Suite

Once we have defined the abstract WSDL and corresponding XML Schemas, we are ready to implement the services they define within the SOA Suite. These services will typically be implemented as composites or proxy services within the Service Bus.

As we've seen in earlier chapters, the simplest way to use a predefined schema within a composite is to import the schema from the filesystem into our SOA project.

When we do this, JDeveloper does two things. First, it will add a copy of the schema file to our SOA project. Second, it will add an import statement into the WSDL of the service component (for example, BPEL, Mediator) that is referring to it, for example:

```
<types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema">
        <import  namespace="http://rubiconred.com/obay/xsd/order"
                 schemaLocation="Order_v1_0.xsd" />
    </schema>
</types>
```

Here, `schemaLocation` is a relative URL that refers to the imported file. In many scenarios, this is fine. However, if you have several processes, each referring to their own local copy of the same XML Schema, which is likely to be the case with our canonical model. Then when you need to change the schema, you will be required to update every copy.

One way is to just have a master copy of your XML Schema and use build scripts to update each of the copies every time you create a build. However, this isn't ideal. A better approach is to have a single copy of the schema that is referenced by all composites.

Sharing XML Schemas across composites

The SOA infrastructure incorporates a **Metadata Service** (**MDS**), which allows you to share common artifacts such as XML Schemas across SOA composites. MDS supports two types of repository:

- **File-based repository**: This is quicker and easier to set up, so it is typically used as the design-time MDS by JDeveloper.
- **Database repository**: This is installed as part of the SOA infrastructure. This is used at runtime, but can also be used by JDeveloper as the MDS at design-time.

By default, a file-based repository is installed with JDeveloper and sits under the directory structure:

```
<JDeveloper Home>/jdeveloper/integration/seed
```

This already contains the subdirectory `soa`, which is reserved for and contains artifacts used by the SOA infrastructure. For artifacts that we wish to share across our applications in JDeveloper, we should create the subdirectory `apps` (under the `seed` directory). This is critical as when we deploy the artifacts to the SOA infrastructure, they will be placed in the `apps` namespace.

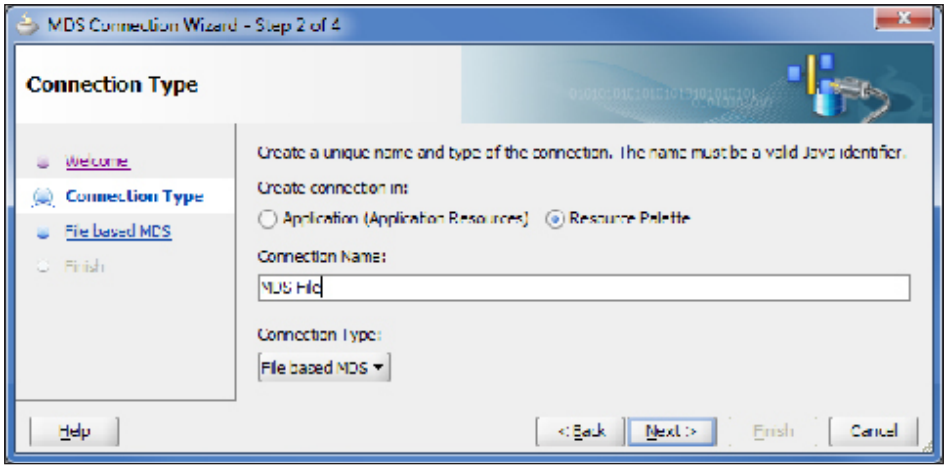
For oBay, we have created the following file structure under `apps`:

```
com/rubiconred/obay/xsd/Account_v1_0.xsd
com/rubiconred/obay/xsd/Auction_v1_0.xsd
com/rubiconred/obay/xsd/Base_v1_0.xsd
com/rubiconred/obay/xsd/Common_v1_0.xsd
com/rubiconred/obay/xsd/Item_v1_0.xsd
com/rubiconred/obay/xsd/Listing_v1_0.xsd
com/rubiconred/obay/xsd/Order_v1_0.xsd
com/rubiconred/obay/xsd/User_v1_0.xsd
```

We have defined the path based on the namespace for each of the XML Schemas as this makes it simple to locate the schema within the resource browser.

Defining an MDS connection

Before we can reference these from within JDeveloper, we need to define a connection to the file-based MDS. Within JDeveloper, from the **File** menu, select **New** to launch the **Gallery**. Under **Categories**, select **General | Connections**, and then select **SOA_MDS Connection** from the **Items** list. This will launch the **MDS Connection Wizard**, as shown in the following screenshot:



Create the connection in the **Resource Palette**, give it an appropriate name, specify a **Connection Type** of **MDS-File**, and then click on **Next**.

We then need to specify the MDS root folder on our local filesystem. This will be the directory that contains the `apps` directory, namely:

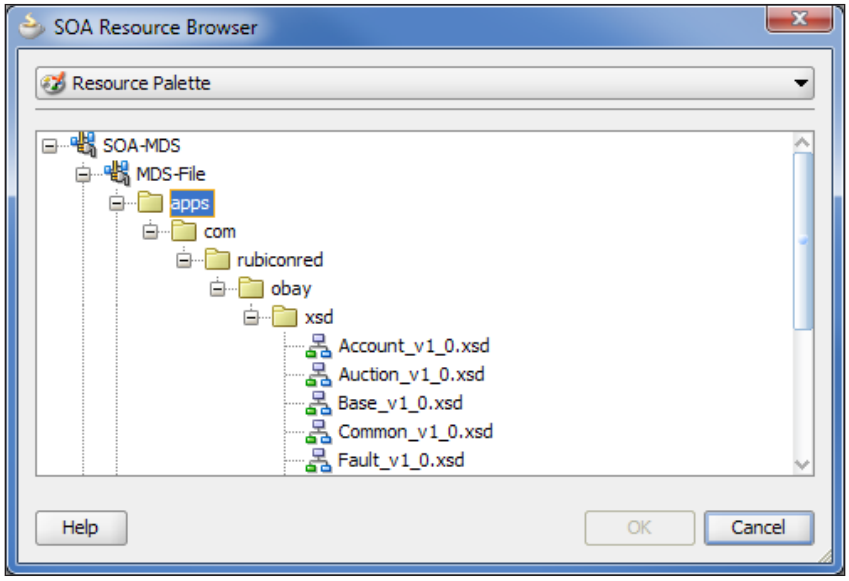
```
<JDeveloper Home>\jdeveloper\integration\seed
```

Once specified, click **Next**, and then click **Finish**.

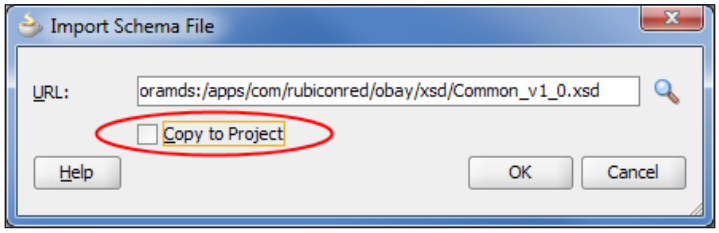
Alternatively, we could create a **DB based MDS** connection against the MDS database repository installed as part of the SOA infrastructure. In this case, we would need to specify the database connectivity information for the SOA MDS schema (which we defined when we installed the SOA Suite).

Importing schemas from MDS

We import schemas from MDS in a similar fashion to how we import them from the local filesystem. The key here is that when we launch the **SOA Resource Browser**, we select **Resource Palette** from the drop-down menu (as shown in the following screenshot).



Select the schema that you wish to import, and click **OK**. This will take you back to the **Import Schema File** window. Make sure you deselect **Copy to Project** (as circled in the following screenshot):



When we import a schema in this fashion, JDeveloper will import it as an **Inline Schema**, meaning it doesn't actually make a copy of the schema; rather it just adds an import statement into the WSDL for the service component where the `schemaLocation` attribute is set to the specified URL.

For schemas referenced by the MDS, the `schemaLocation` attribute takes the following format:

```
oramds:/apps/<schema name>
```

While `oramds` indicates that it is located in the MDS, `apps` indicates that it is in the application namespace and `<schema name>` is the full path name of the schema in the deployed archive. So in the preceding example, it would be:

```
schemaLocation="oramds:/apps/com/rubiconred/obay/xsd/Common_v1_0.xsd"
```

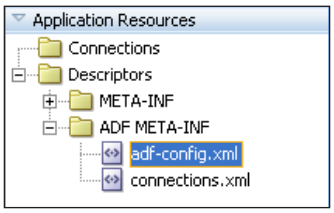
The schema location doesn't specify the physical location of the schema; rather it is relative to the MDS (which is specific to the environment in which the composite is deployed). This makes the WSDL more portable, as we don't have to modify the schema location for each environment that it's deployed to (as we did with SOA Suite 10.1.3).

Manually importing schemas

Instead of using the **SOA Resource Browser** to import the schema, it may seem simpler (particularly if we have to import multiple schemas) to manually edit the WSDL file to contain the appropriate import statements.

However, there is one nuance that we need to be aware of, that is, we need to define in the application properties how it should resolve the location of metadata in the `apps` namespace. The reason we didn't have to worry about this earlier is that when we imported the schema via the resource browser, JDeveloper automatically updated the application properties for us.

Within the **Application Resources** view for our SOA application, you will notice that it contains the file **adf:config.xml**, as shown in the following screenshot:



Double-click on this to open it, switch to the source view, and scroll down to the `metadata-namespaces` section, as shown in the following code snippet:

```
<persistence-config>
  <metadata-namespaces>
    <namespace metadata-store-usage="mstore-usage_1"
               path="/soa/shared"/>
    <namespace metadata-store-usage="mstore-usage_2"
               path="/apps/com"/>
  </metadata-namespaces>
  <metadata-store-usages>
    <metadata-store-usage id="mstore-usage_1">
      <metadata-store class-name
        ="oracle.mds.persistence.stores.file.FileMetadataStore">

        <property value="${oracle.home}/integration"
                  name="metadata-path"/>
        <property value="seed" name="partition-name"/>
      </metadata-store>
    </metadata-store-usage>
    <metadata-store-usage id="mstore-usage_2">
      <metadata-store class-name
        ="oracle.mds.persistence.stores.file.FileMetadataStore">

        <property value="${oracle.home}/integration"
                  name="metadata-path"/>
        <property value="seed" name="partition-name"/>
      </metadata-store>
    </metadata-store-usage>
  </metadata-store-usages>
</persistence-config>
```

By default, this contains a single `namespace` entry and a corresponding `metadata-store-usage` entry. However, for those applications where we have imported a resource from MDS, it will contain a second entry (highlighted earlier), which defines the MDS repository to the application. If we manually edit the WSDL files to import a schema (something we will do in a moment), then we will need to manually edit this file.

If you examine the definition of `mstore-usage_2`, you will see that it's the same as the default definition `mstore-usage_1` and so is redundant. This is because we choose to use the file-based repository installed with JDeveloper.

Thus the only change we need to make to the default `adf:config.xml` file is to add a second namespace entry that references the default metadata store and defines the `apps` path, as shown in the following code snippet:

```
<persistence-config>
  <metadata-namespaces>
    <namespace metadata-store-usage="mstore-usage_1"
               path="/soa/shared"/>
    <namespace metadata-store-usage="mstore-usage_1"
               path="/apps/com"/>
  </metadata-namespaces>
  <metadata-store-usages>
    <metadata-store-usage id="mstore-usage_1">
      <metadata-store class-name
        ="oracle.mds.persistence.stores.file.FileMetadataStore">

        <property value="${oracle.home}/integration"
                  name="metadata-path"/>
        <property value="seed" name="partition-name"/>
      </metadata-store>
    </metadata-store-usage>
  </metadata-store-usages>
</persistence-config>
```

Note that for those scenarios where we are using the DB-based MDS of the SOA infrastructure, we would need to keep the `mstore-usage_2 entry` and configure it appropriately, for example:

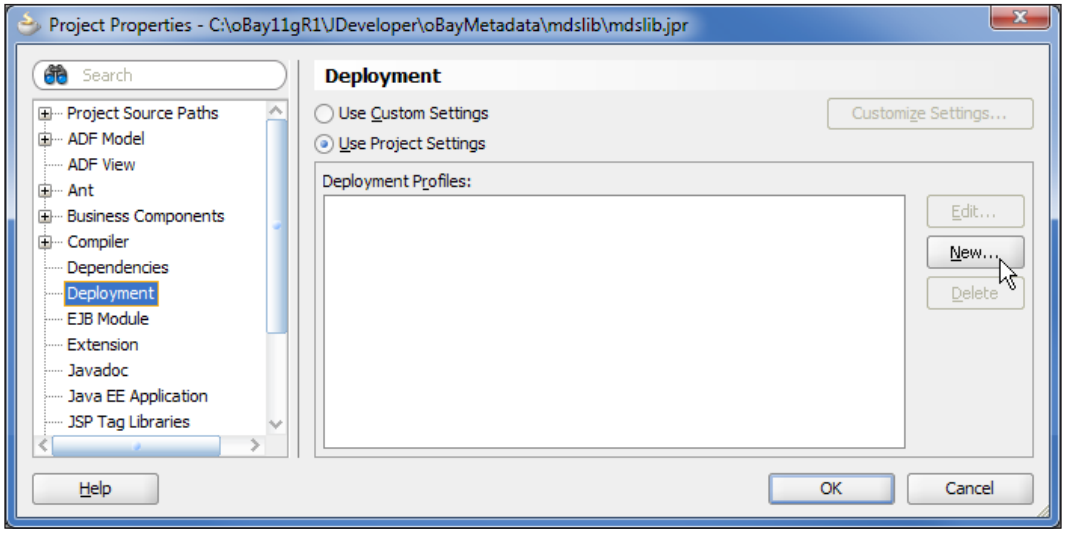
```
<metadata-store-usage id="mstore-usage_2">
  <metadata-store class-name=
    "oracle.mds.persistence.stores.db.DBMetadataStore">

    <property value="DEV_MDS" name="jdbc-userid"/>
    <property value="welcome1" name="jdbc-password"/>
    <property value="jdbc:oracle:thin:@localhost:1521:XE"
                  name="jdbc-url"/>
    <property value="soa-infra" name="partition-name"/>
  </metadata-store>
</metadata-store-usage>
```

Deploying schemas to the SOA infrastructure

Before we can deploy a composite that references artifacts held in MDS, we must deploy those artifacts to the MDS on the SOA infrastructure. To do this, we need to create a `JAR` file containing the shared artifacts and then deploy it as part of an SOA Bundle.

Creating a JAR file within JDeveloper

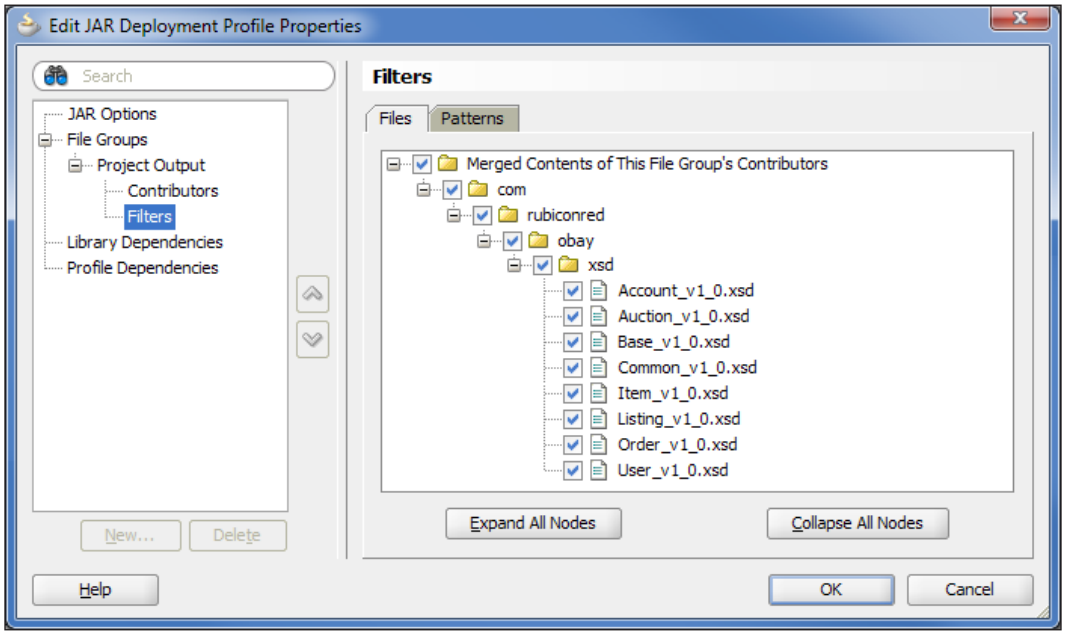
To create this JAR file within JDeveloper, create a **Generic Application** (for example, **obayMetadata**) and when prompted to, create a project and give it an appropriate name (for example, `mdslib`). In the application navigator, right-click the **mdslib** project and select **Project Properties**. This will launch the **Project Properties** window, select **Deployment** from the navigational tree, as shown in the following screenshot:



Click **New…**, this will launch the **Create Deployment Profile** dialog. Specify an archive type of **JAR File**, specify an appropriate name (for example, **mdslib**), and click **OK**. This will launch the **Edit JAR Deployment Profile Properties** window, where we can specify what goes in the JAR file.

We only want to include the actual XML Schemas in the JAR file, so deselect **Include Manifest File**, then select **File Groups** | **Project Output** | **Contributors** from the navigational tree, and deselect **Project Output Directory** and **Project Dependencies**.

Now we can specify the actual schemas we wish to add to the JAR file. Click on **Add**. This will launch the **Add Contributor** window. Click on the magnifying glass and browse to the **apps** directory that we previously created and click **OK**. Next select **File Groups** | **Project Output** | **Filters**, and check that only the files we want are included within the JAR file.

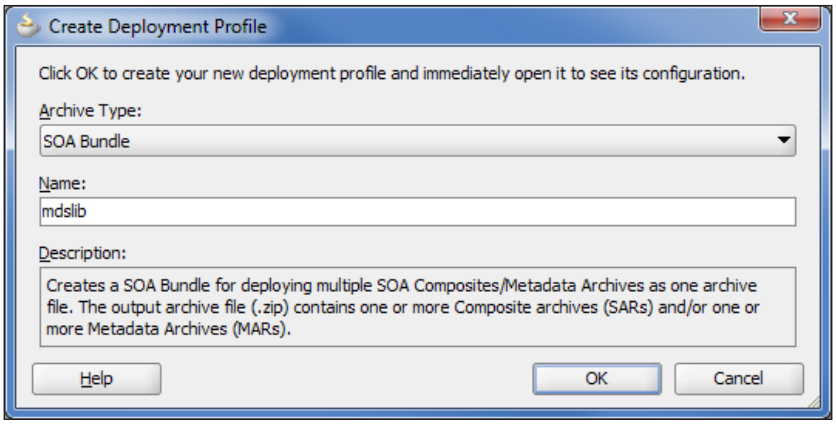


Click **OK** to confirm the content of the JAR file, and then click **OK** one more time to complete the deployment profile, finally, from the main menu select **Save All**.

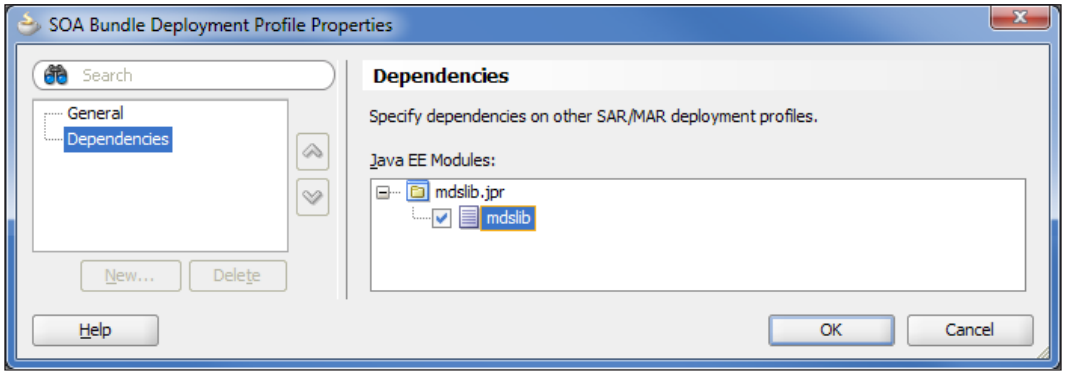
Creating an SOA Bundle for a JAR file

In order to deploy our JAR file to the metadata repository, we need to place it within an SOA Bundle (see *Chapter 19*, *Packaging and Deployment* for more details on what an SOA Bundle is) and deploy that to our SOA infrastructure.

To create an **SOA Bundle**, from the **Application Menu** select **Application Properties**, which will launch the corresponding window. From the navigational tree, select **Deployment**, and then click **New**. This will launch the **Create Deployment Profile** window, as shown in the following screenshot:



Specify an archive type of **SOA Bundle** and an appropriate name and then click **OK**. This will launch the **SOA Bundle Deployment Profile Properties** window; select **Dependencies** from the navigational tree, and ensure that **mdslib** is selected.



Click **OK** twice and then select **Save All** from the toolbar. We are now ready to deploy our XML Schemas to the metadata repository. In order to do this, from the **Application Menu**, select **Deploy | SOA Bundle Name**. This will launch the **Deployment Action** dialog. Select **Deploy to Application Server** and follow the standard steps to deploy it to your target SOA infrastructure server(s).

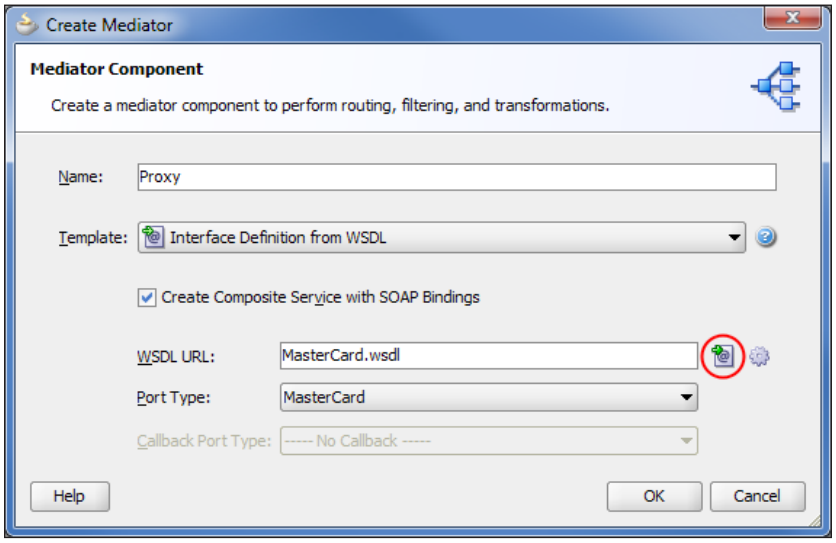
The schemas will then be made available to SOA Composites deployed on the same SOA infrastructure.

Importing the WSDL document into a composite

As discussed in the previous chapter, when creating a composite, it is a good practice to use a Mediator as a proxy for the composite, which implements the abstract WSDL contract that we have designed.

Essentially, there are two approaches to this. The first is to create a Mediator in the normal way, using the appropriate template—synchronous, asynchronous, or one-way. When you do this, JDeveloper will create a basic WSDL file for the process. You can then modify this WSDL (using the WSDL editor in JDeveloper) to conform to the abstract WSDL that you have already defined.

The alternative is to import the abstract WSDL document into the Mediator itself. With this approach, you create your Mediator using the template for an **Interface Definition from WSDL**, as shown in the following screenshot:



Ensure that **Create Composite Service with SOAP Bindings** is selected. Then click on **Find existing WSDLs** (circled in the preceding screenshot) and select the WSDL we want to import from the local filesystem. Ensure that the appropriate Port Type is selected and click **OK**.

This will add an empty Mediator component to our composite and expose it as an external service.

Note that if the WSDL references any external resources defined in the MDS, we must first define the MDS repository to the applications; otherwise we will get an exception when we try to create the Mediator.

The simplest way to achieve this, when we first create a new SOA application, is to create the project using the template for an **Empty Composite**. We can then update the `adf:config.xml` file, as described earlier. Finally, we can add a Mediator to the composite and create it based on our predefined WSDL.

We can also create a BPEL process based on our abstract WSDL by following the same approach.

Sharing XML Schemas in the Service Bus

As with composites, it is possible within the Service Bus to create multiple projects each with their own local copy of the schema. However, as before, it's considered best practice to only have a single copy of each schema.

This is easily achieved by having a single project that defines your schemas, which is then shared across other projects. In order to be consistent with the SOA infrastructure, we have defined the project `mds`, and under this, created an identical folder structure into which we have imported our schemas.

For example, to mirror how we have deployed the order schema to MDS, we have created the folder structure `com/rubiconred/obay/xsd` within the `mds` project, into which we have imported the `Order_v1_0.xsd` schema.

Importing the WSDL document into the Service Bus

Before we create a proxy service that implements our abstract WSDL, we need to define the bindings for the service, which in our case will be Document/literal. We can either modify the WSDL file to include the bindings before we import it, or add in the bindings after we have imported the WSDL into the Service Bus.

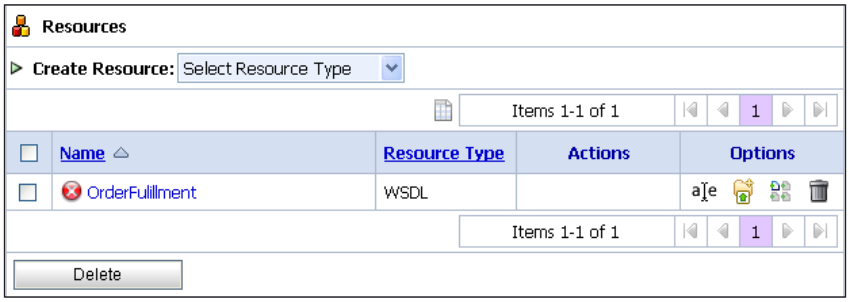
Defining the SOAP bindings for our service and each of its corresponding operations is pretty straightforward, as we have already settled on Document/literal for this.

For example, the bindings for our `orderFulfillment` service are as follows:

```
<binding xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
         name="orderFulfillmentBinding"
         type="tns:orderFulfillment">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name=" setShippingInstruction">
        <soap:operation style="document"
                        soapAction="setShippingInstruction"/>
        <input>
           <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
    </operation>
    <operation name="submitInvoice">
        <soap:operation style="document" soapAction="submitInvoice"/>
        <input>
           <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
    </operation>
    …
</binding>
```

When we import our WSDL, if it imports any schemas, then the Service Bus will present us with a warning message, similar to the one shown in the following screenshot, indicating that there were validation errors with the WSDL:

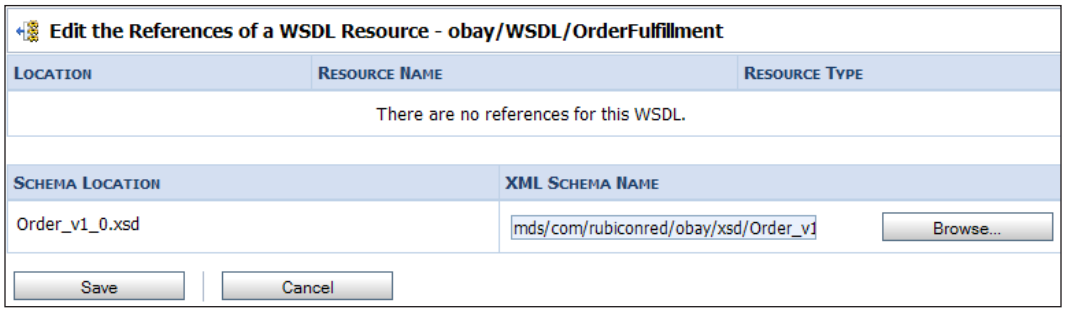
If you look at the list of resources in the browser, it will also have an **X** next to the WSDL we just imported.



If you click on the WSDL name to edit it, the Service Bus will display the WSDL details with the error **One of the WSDL dependencies is invalid**.

This is because if a WSDL references any external resources (that is, the order schema in this case), we must first import that resource into the Service Bus and then update the WSDL reference to point to the imported resource.

To do this, click on the button **Edit References**. The Service Bus will display a window listing all the references included in the WSDL, with a section listing all the schema references, as shown in the following screenshot:



Clicking on the **Browse...** button will launch a window from where you can select the corresponding XML Schema that the WSDL is referring to, as shown in the following screenshot:

By default, the window will list all the schemas defined to the Service Bus, though you can restrict the list by defining the search criteria. In the case of our `orderFulfilment` service, just select the schema `Order_v1_0.xsd`, and click on **Submit**.

This will take you back to the **Edit References** screen and click **Save**. This will take you back to the **View/Edit WSDL** screen, which should display the confirmation **The References for the WSDL "orderFulfillment" were successfully updated**.

Your WSDL can now be used to define a proxy service in the normal way.

If you import a schema into the Service Bus, which references other schemas, then you will need to go through a similar process to define all its dependencies.

Strategies for managing change

One of the key design principles behind SOA is that systems should be designed and built to accommodate future changes in response to ever-changing business requirements.

So far, we have looked at how to design and build the initial system, so that when change does occur, it can be isolated through the use of service contracts to particular parts of the overall system.

While allowing us to restrict the impact of change, it doesn't completely mitigate all the complexities, especially when you consider that the consumer and provider of a service may be in completely separate organizations.

Major and minor versions

When we upgrade the version of a service, for example, from version 1 to version 2, then from the consumer's perspective, there are two possible outcomes. Either the version 1 consumer can continue to successfully use version 2 of the service, in which case the service is said to be backward compatible or the change will break the existing contract.

To be explicit, a service is said to be backward compatible if **ALL** messages that would have been successfully processed by version 1 of the service, will be successfully processed by version 2 of the service.

It is a good practice to assign a version number to each service, which indicates the level of backward compatibility. A typical approach is to assign a major and minor version of the format `<major>`, `<minor>` (for example, 1.0, 1.1, 2.0, and so on), where:

- A minor change signifies a change that is backward compatible with previous versions of the service that share the same major number. These types of changes typically contain small new features, bug fixes, and so on.
- A major change signifies a change that is incompatible with a previous deployment of the service. Major changes typically indicate significant new features or major revisions to the existing services.

You also have the concept of forward compatibility, whereby the consumer is upgraded to use a future version of the service, before the actual provider of the service is upgraded.

If we examine the anatomy of a web service, it is essentially made up of three components, namely, its WSDL contract, referenced data types from our canonical model, and the actual implementation of the service.

From a versioning standpoint, we need to consider how a change to any of these components is reflected in the version of the overall service.

Service implementation versioning

This may seem a strange topic to cover. After all, surely one of the key concepts of SOA is to isolate change. For example, if I change the implementation of a service, but the consumer sees no change to the contract, then has it really changed at all?

Initially, the answer may seem obvious. However, if we revisit our earlier definition of backward compatible, we can see some issues:

> *A service is said to be backwards compatible if* **ALL** *messages that would have been successfully processed by version 1 of the service, will be successfully processed by version 2 of the service.*

Under this definition, if we add some extra validation to version 2 of the service (for example, we check that a date is in the future). This would mean some messages valid under the original version are no longer valid. The same sort of scenario could again occur if we were to fix a bug (or even introduce one).

Another more surreptitious change is one whereby we change the processing of the data, so that ALL messages are still processed successfully, but the result is different. For example, if we had a service which returned the price of an item, but instead of returning the price in dollars and cents, it now returned the price in cents.

With each of these scenarios, there is no hard and fast rule. However, when you implement these types of changes, you need to consider whether it requires the release of a new version of a service and whether that should be a minor or major version.

Another way of handling this type of change is not to modify the version of the service, but rather provide a means of notifying the consumer that there has been a change to the service. One mechanism for managing this is through the Oracle Enterprise Repository.

Schema versioning

When we modify a schema, we follow the same core principles for major and minor versions that we outlined earlier: a minor change to a schema indicates that an instance document, which was created against a previous version of the schema, is still valid against the new version of the schema, as long as the schemas share the same major version number. Minor changes include:

- The definition of new elements, attributes, and types
- Adding optional attributes and elements to existing elements and types
- Making existing mandatory attributes and elements optional
- Convert an element into a choice group
- Making simple types less restrictive

Changing schema location

Encode the schema version in the filename of the schema; for example, we have named our auction schema `Auction_v1_0.xsd`. Whenever we import a schema, either in another schema or within a WSDL document, the `schemaLocation` attribute will contain the version of the schema being used.

- This has two advantages, we can immediately see what version of a schema a web service is based on, simply by looking at what files we are importing within the WSDL. Additionally, it allows us to have multiple versions of a schema deployed side-by-side allowing each service to upgrade to a newer version of a schema as it suits them.
- When we upgrade a service to use the new version of a schema, then of course we will have a corresponding new version of the service.

Updating schema version attribute

Use the schema version attribute to document the version of the schema. Note that this is purely for documentation, as there is no processing of this attribute by the parser. This ensures that if the schema is renamed so as to remove the encoding of the schema version from the filename, we still know the version of that schema.

Resisting changing the schema namespace

One common practice is to embed the version of the schema within its namespace, and update the namespace for new versions of the schema. However, this has the potential to cause major change to both consumers and providers of a service, so I would strongly recommend that you use this approach with care, if at all.

Firstly, when you change the namespace of a schema, it is no longer backward compatible with previous versions of the schema. So by definition, changing the namespace is a major change in its own right. Therefore, never change the namespace for a minor change.

For major changes, changing the namespace would seem a very valid approach, as apart from being a clear indication to the client that we have introduced a service that is not backward compatible, it will prevent you from successfully calling any of the operations provided by the new version of the service.

However, it is important to understand the magnitude of this change, as it will typically break a number of components on both the client and service provider. For example, all your XPath assignments and XSLT transformations will have to be updated to use the new namespace. Therefore, implementing the change will be more time-consuming.

In some ways, you may want to consider how significant a major change is. For example, the change might impact one operation out of ten. Do you really want your clients to have to reimplement every call to every single operation because one operation (which they might not be using) has changed?

WSDL versioning

When we modify our WSDL contract, we again follow the same core principles for major and minor versions that we outlined previously. From a service perspective, a minor change includes:

- Addition of an operation: This merely extends the WSDL and thus on its own is backwards compatible
- Addition of a new optional parameter within the element wrapper used by an input message
- Making existing mandatory parameters within the element wrapper optional for input messages

While major changes would include:

- Deletion or renaming of an operation
- Changes to the input and output parameters of an operation that don't fall under the category of a minor change, that is, adding a new parameter whether optional or mandatory to the response wrapper

Incorporating changes to the canonical model

If we upgrade our service to use a new minor version of the canonical model, then our initial reaction might be that this only results in a minor change to the service, as our new version will still be able to process all requests from consumers using a service definition, based on an earlier version of the schema.

While this is true, the response generated by our service may no longer be compatible with an earlier version of the schema. So, the consumer may not be able to successfully process the response. In this scenario, you need to create a new major version of the service.

Changes to the physical contract

From a versioning perspective, we don't generally consider changes to either the `<binding>` or `<service>` element. With regards to the `<binding>` element, we find it helpful to consider it as part of the service implementation and thus follow the same guidelines discussed earlier to decide whether it warrants a new version of a service.

While changes to the service end point presented by a composite merely indicates the relocation of a service, as is typically required when moving a composite from development into test and finally into production. As such, this is more of a deployment consideration and is covered in detail in *Chapter 19*, *Packaging and Deployment*.

Updating the service endpoint

A simple way to record the version of a service is to encode it within its endpoint. The SOA Composite infrastructure does this for you already; for example, whenever you deploy a composite, you specify its version.

For example, when deploying version `1.0` of the `Auction` composite to the SOA infrastructure, its end point will be:

```
http://host:port/soa-infra/services/default/Auction!1.0/proxy_ep
```

With the WSDL for the service being available at the following URL:

```
http://host:port/soa-infra/services/default/Auction!1.0/proxy_
ep?WSDL
```

With the Service Bus, you have even more flexibility over the endpoint of a proxy service, as you can specify this as part of the transport configuration. We recommend following a similar naming strategy to that used by SOA composites in order to maintain consistency.

This has two advantages; first we can immediately see from the URI what version of a service we are looking at. Additionally, it provides a simple mechanism for us to have multiple versions of a service deployed side-by-side, which is important when we consider the lifecycle of a service.

Including version identifiers in the WSDL definition

While the WSDL definition element doesn't provide an explicit version attribute, we can still make use of the WSDL documentation element to hold the version number. For example, to add a version number to the Order Fulfilment WSDL, we can add the `<documentation>` element, as highlighted in the following code snippet:

```
<definitions name="OrderFulfillment"
    targetNamespace="http://rubiconred.com/obay/svc/OrderFulfillment"
    xmlns:tns="http://rubiconred.com/obay/svc/OrderFulfillment
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ord="http://rubiconred.com/obay/xsd/order"
    xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
```

```
        <documentation>Version 1.0</documentation>
    …
    </definitions>
```

Managing the service lifecycle

When we release a new version of a service, we need to consider how we wish to manage previous releases of that service.

A typical first step is to set the status of the previous version to be deprecated. This indicates to existing users that the service has been updated with a newer version and therefore will be retired at some point in the future.

This tells the existing users that they need to start the process of migrating to the newer version of the service as well as indicating to new users that there is a newer version of the service they should use. The final step is to retire the service. At this point, the service is removed from production, so that it is no longer available for use.

When we make a minor release of the service, as it is backward compatible with the previous version it should be straightforward to migrate to the newer version, as the only change that the consumer will be required to make is to call the service at a new endpoint (and even this may not have changed). In this case, the previous version of the service can be retired relatively quickly.

However, with a major release, changes will have to be made to the consumer before they can move to the new version; in this case, the deprecated service will need to be maintained for a longer period of time and may require even minor releases of its own to fix bugs and so on.

With both of these scenarios, a lot will depend on the number of consumers, and how easy or difficult they are to identify and coordinate changes across, as well as the nature of the change.

One way to handle change is to create a façade that would map the old interface to the new service interface. This maintains support for existing consumers (without modification), but means that there is only a single instance of the implementation of the service.

A key to simplify this is to also keep service consumers informed of planned future versions of services, as well as those under development, as this will allow them to plan for future releases and thus shorten the required life span of deprecated services.

Summary

Design of the service contract and the underpinning canonical model are fundamental steps in the overall implementation of an SOA-based solution. The keyword here is design, as it's all too easy with the tools we have at our disposal to knock out a model in order that the "real work" of implementation can begin.

In this chapter, we have given you an overview of how to go about structuring your XML canonical model, both in terms of modeling your data in a tree-like structure as well as how to partition it across multiple namespaces.

We've also given some guidance on best practice for the implementation of those schemas, whether you follow ours or define your own. The key is to put in place some standard guidelines in order to ensure consistency, as this will result in schemas which interoperate better and are easier to reuse and maintain.

The canonical model provides the foundation for our service contracts, and with this in place, we have defined the best practice regarding how we define our service contract, paying particular attention to using the Document/literal wrapped pattern in order to conform to WS-Interoperability guidelines.

As stated earlier, a core tenet of SOA is that systems should be designed to accommodate change. With this in mind, we have also examined how we can manage change, both in our schemas and in our actual service contract, and have outlined a versioning approach to support this.

Lastly, we looked at how the SOA Suite supports the running of deprecated services alongside the most recent release in order to enable consumers to upgrade to the newer version of a service in their own time.