

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Chapter 5. Established SOA Standards

In most cases, you use the following open-standard technologies when developing a Web service.

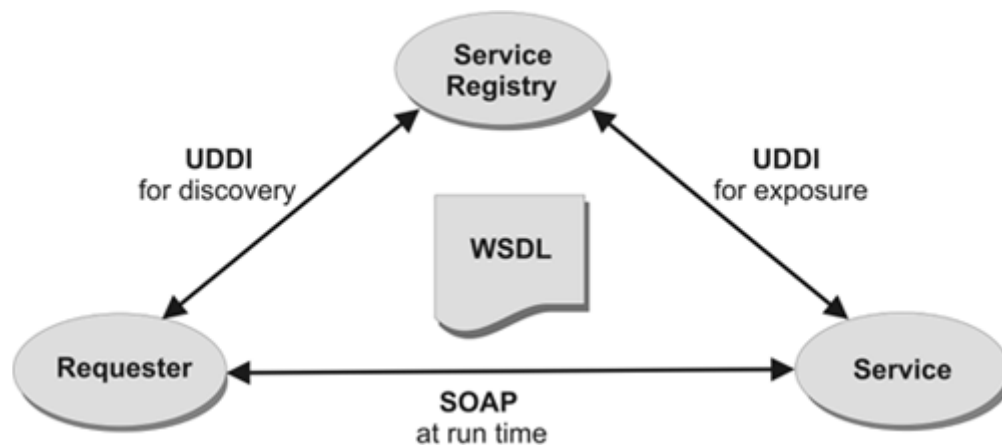
- *Web Services Description Language (WSDL)* is an XML format for describing how to access a service.
- *SOAP* is an XML format for transmitting data to and from a Web service. The runtime message includes business data (as defined in the WSDL definition) and may contain Quality-of-Service (QoS) details too, including (for example) the security details needed to gain access to the service. Although QoS details may be defined in the WSDL definition, those details are increasingly likely to be defined only when a system administrator uses an SOA runtime product to configure the message.

SOAP was once called Simple Object Access Protocol and is now known only as SOAP.

- *Universal Description, Discovery, and Integration (UDDI)* is a set of rules for registering and retrieving details about a business and its services. Some of the registered details are in the form of WSDL definitions.

WSDL and SOAP have become enormously important, with UDDI less so. The three standards are interrelated to some extent. [Figure 5.1](#) depicts the relationship.

Figure 5.1. Established SOA standards



As the figure suggests, a WSDL definition describes a Web service. A UDDI-compliant service registry can help companies to discover that description. At run time, SOAP-based messages carry business data that conforms to the description.

The World Wide Web Consortium (W3C) is the sponsoring organization for WSDL and SOAP. The Organization for the Advancement of Structured Information Standards (OASIS) has an equivalent role for UDDI.

WSDL

WSDL is an XML format that tells how to access a Web service. Increasingly, this format is used to describe the interface of any kind of service.

You use a WSDL definition to communicate the service interface to developers, who use the information to invoke the service. A WSDL definition also ensures that runtime access is handled correctly on both the requester and service sides of the transmission.

A WSDL definition is cumbersome, resembling a box with many wrapped objects inside, some embedded in others. If you were rewriting the words of an old Stevie Wonder song, you might say, "Isn't she clunky?"

"Yes" is the short answer, and that's why the WSDL definition is usually created for you by an automated tool, though you may want to customize the definition. Among the reasons for the complexity:

- To allow flexibility at WSDL development time. The fine distinctions between one element type and the next allow for greater reuse of different kinds of information (for example, messages).
- To allow vocabulary extensions that are precisely targeted to support new functionality, including functionality that WSDL's designers may not have foreseen.
- To support a wide range of runtime behaviors. In particular, a WSDL definition can describe a complex service interface:
 - A service can include several operations that interact with requesters. A particular order-processing service can include three operations, for example: one for receiving an order from an existing corporate customer, one for receiving an order from an unknown individual, and one for reporting on the status of an order.
 - A single operation also might support multiple message exchange patterns (MEPs), as when the message sent in one invocation requires a response but the message sent in a second invocation does not.

Definitions written in WSDL 1.1 are commonplace and usually support only the following MEPs, which were covered in [Chapter 2](#):

- a one-way pattern, in which the requester invokes the service with an input message but does not receive a response
- a request-response pattern, in which the requester invokes the service with an input message and receives a response, which may be a fault message

In the future, use of WSDL 2.0 is likely to result in a more widespread use of other MEPs.

You can include a WSDL definition in a single file or divide the definition into several files. [Listing 5.1](#) shows the outline of a typical WSDL 1.1 definition.

Listing 5.1. Outline of a WSDL 1.1 definition

```
<definitions>
  <types> </types>
  <message>
    <part> </part>
  </message>
  <portType>
    <operation>
      <input> </input>
      <output> </output>
      <fault> </fault>
    </operation>
  </portType>
  <binding>
    <operation> </operation>
  </binding>
  <service>
    <port> </port>
  </service>
</definitions>
```

In the next sections, we review the WSDL 1.1 definition for a service called `getMotorVehicleRecord`, which is provided by a fictional Department of Motor Vehicles (DMV). Highlight Insurance invokes this service to verify the details in an insurance-policy request. [Listing 5.2](#) shows the complete WSDL definition.

Listing 5.2. WSDL definition for the `getMotorVehicleRecord` service

```
<wsdl:definitions
  name="MotorVehicleRecordsService"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.dmv.org/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

```

xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.dmv.org/">

<wsdl:types>
  <xsd:schema targetNamespace="http://www.dmv.org/">
    <xsd:element name="getMotorVehicleRecord">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="VIN" type="xsd:string"/>
          <xsd:element name="State" type="xsd:string"/>
          <xsd:element name="Category" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="getMotorVehicleRecordResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="RequestID" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</wsdl:types>

<wsdl:message name="getDMVRecordRequest">
  <wsdl:part element="tns:getMotorVehicleRecord"
    name="DMVRecordRequest" />
</wsdl:message>
<wsdl:message name="getDMVRecordResponse">
  <wsdl:part element="tns:getMotorVehicleRecordResponse"
    name="DMVRecordResponse" />
</wsdl:message>

<wsdl:portType name="DMVRecord">
  <wsdl:operation name="getMotorVehicleRecord">
    <wsdl:input message="tns:getDMVRecordRequest" />
    <wsdl:output message="tns:getDMVRecordResponse" />
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="MotorVehicleRecordsSOAPBinding"
  type="tns:DMVRecord">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="getMotorVehicleRecord">
    <soap:operation soapAction=""/>
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="MotorVehicleRecordsService">
  <wsdl:port binding="tns:MotorVehicleRecordsSOAPBinding"
    name="MotorVehicleRecordsSOAPPort">
    <soap:address location="http://www.dmv.org/" />
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

We review the two kinds of details included in a WSDL definition: service-interface details and additional access details. For a complete description of WSDL, see the following W3C documents:

- *Web Services Description Language (WSDL) 1.1*, available at <http://www.w3.org/TR/wsdl>.
- *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, available at

<http://www.w3.org/TR/wsdl20>.

Service-Interface Details

The following sections identify WSDL elements that are used to describe a service interface.

The definitions Start-tag

The `definitions` element is the root element of the WSDL definition. Listing 5.3 shows the start-tag of the `definitions` element in our example.

Listing 5.3. Sample definitions start-tag

```
<wsdl:definitions
  name="MotorVehicleRecordsService"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.dmv.org/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.dmv.org/">
```

That start-tag includes several attributes, and understanding all but the first requires an understanding of namespaces, which were described in Chapter 4.

The first attribute, `name`, is solely for documentation. The last attribute, `targetNamespace`, specifies the target namespace, which contains each name you're adding to the WSDL file. The target namespace contains the value of the `name` attribute of each `message` element, for example. The target namespace specified in the `definitions` start-tag, however, does not include the names you add in the file's XML Schema definition (XSD), which has a `targetNamespace` attribute that applies solely to that Schema. In many cases, the two target namespaces are the same.

Namespace declarations in the `definitions` start-tag indicate a default namespace (which is specific to WSDL) and define a set of prefixes that can be used to reference other namespaces. Our example features three prefixes: `tns` refers to the target namespace identified in the `definitions` element; `xsd` refers to the target namespace identified in the XML Schema; and `soap` refers to a SOAP-specific namespace.

The types Element

The WSDL definition's `types` element describes the data available for building the input and output messages. In our example, that element includes the XML Schema definition shown in Listing 5.4.

Listing 5.4. XML Schema definition

```
<wsdl:types>
  <xsd:schema targetNamespace="http://www.dmv.org/">
    <xsd:element name="getMotorVehicleRecord">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="VIN" type="xsd:string"/>
          <xsd:element name="State" type="xsd:string"/>
          <xsd:element name="Category" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="getMotorVehicleRecordResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="RequestID" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</wsdl:types>
```

In keeping with the preferred message format (called *document-literal wrapped*), this XSD begins with a *wrapper element* for the request message. That element has the same name as the operation being invoked and includes or references a complex type that describes the message.

The previous `types` element is appropriate for our example because the collection of data types is probably unique to a specific service interface. An alternative is appropriate, however, when you're defining a data-type collection that is likely to be reused. We'll continue with this example to show you the alternative `types` element ([Listing 5.5](#)).

Listing 5.5. Alternative types element

```
<wsdl:types>
  <xsd:schema>
    <xsd:import namespace="http://www.dmv.org/"
      schemaLocation="GetMVRecord.xsd">
    </xsd:import>
  </xsd:schema>

  <xsd:element name="getMotorVehicleRecord">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="recordSearchInfo"
          type="xsd1:MVRecordSearchInfo"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="getMotorVehicleRecordResponse">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="response"
          type="xsd1:MVRecordResponse"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</wsdl:types>
```

We've used the preferred message format again, but in this case we've referenced the complex types, stored the types separately, and used an import statement to make the types available in the WSDL file. This approach yields several benefits. It lets the business developer change the types without affecting the WSDL file. In addition, the types can be used in multiple WSDL files; can be the basis of extensions and restrictions, as shown in [Chapter 4](#); and can be the basis of a shortcut used in Business Process Execution Language (BPEL), as shown in [Chapter 8](#). The types also can be stored in a repository that is made available throughout the business. However implemented, the repository can guide designers and programmers in a variety of projects.

[Listing 5.6](#) shows the separately stored types in an XSD file. We comment further on the message format later.

Listing 5.6. Types stored in an XSD file

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.dmv.org/"
  xmlns:tns="http://www.dmv.org/">

  <xsd:complexType name="MVRecordSearchInfo">
    <xsd:sequence>
      <xsd:element name="VIN" type="xsd:string" />
      <xsd:element name="State" type="xsd:string" />
      <xsd:element name="Category" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="MVRecordResponse">
    <xsd:sequence>
      <xsd:element name="RequestID" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

The message Element

Each `message` element in the WSDL definition defines an input, output, or fault message and includes one or more `part` elements, each referring to a data type. Listing 5.7 shows the `message` elements for our example.

Listing 5.7. Sample message elements

```
<wsdl:message name="getDMVRecordRequest">
  <wsdl:part element="tns:getMotorVehicleRecord"
    name="DMVRecordRequest" />
</wsdl:message>
<wsdl:message name="getDMVRecordResponse">
  <wsdl:part element="tns:getMotorVehicleRecordResponse"
    name="DMVRecordResponse" />
</wsdl:message>
```

In some cases, you'll see multiple `part` elements in a `message` element, usually meaning that each `part` element represents a parameter of the service and that the order of `part` elements corresponds to the order of parameters. The best practice in most situations, however, is to organize the WSDL definition as our example is organized: each message includes only one `part` element, and the `types` element identifies the parameters, if any.

In any case, the content of a `message` element is used only if referenced by a descendant of a `portType` element, which we review next.

The portType Element

The `portType` element defines the service interface and can include multiple `operation` elements, each describing how to invoke a specific operation. Listing 5.8 shows the `portType` element for our example.

Listing 5.8. Sample portType element

```
<wsdl:portType name="DMVRecord">
  <wsdl:operation name="getMotorVehicleRecord">
    <wsdl:input message="tns:getDMVRecordRequest" />
    <wsdl:output message="tns:getDMVRecordResponse" />
  </wsdl:operation>
</wsdl:portType>
```

Included in an `operation` element are none, some, or all of the following subordinate elements:

- an `input` element, which defines an input message (but not every service requires an input beyond the operation name)
- an `output` element, which defines an output message, if any
- zero to many `fault` elements, each defining a fault message (that is, an error message)

In most cases, each subordinate element refers to a `message` element. In a complex case, the `operation` element may include a `parameterOrder` attribute to override the parameter order that a related `message` element specified.

WSDL 2.0 replaces the `portType` element with the `interface` element, which is similar.

Additional Access Details

We now review WSDL elements that cause the data to be formatted in a particular way and to be sent to a specific location over a specific transport protocol.

As before, WSDL allows for complexity. You may want to support a wide variety of requesters, including some that process SOAP-based messages and some that do not. You also may want to associate a service with multiple locations, as is useful to distinguish between test- and production-level versions of a service or to ensure that a service is available when a network fails or when many requesters are attempting access.

In the usual case, all requester messages are formatted in the same way and are sent to a single location over a single transport protocol.

The binding Element

The `binding` element associates a `portType` element with most of the additional details needed to structure a runtime transmission. Listing 5.9 shows an example.

Listing 5.9. Sample binding element

```
<wsdl:binding name="MotorVehicleRecordsSOAPBinding"
  type="tns:DMVRecord">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="getMotorVehicleRecord">
    <soap:operation soapAction="" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

The `type` attribute in the `binding` element refers to a `portType` element by name.

Some child elements of the `binding` element represent extensions to WSDL, as needed for a particular message format (such as SOAP) and transport protocol (such as HTTP).

When you work with SOAP over HTTP, you may be asked to choose a binding `style` (*rpc* or *document*) and, less often, a `use` (*literal* or *encoded*).

The binding style represents a message sub-format:

- A value of *rpc* means that the operation name goes into the transmitted message.
- A value of *document* means that the operation name does not go into the transmitted message.

The `binding` element should specify *document* in almost all cases. Incidentally, the values *rpc* and *document* only reflect how to structure a message and do not correspond to the two kinds of service interface described in Chapter 2.

The `use` value concerns how the runtime transmission is structured for data-type validation:

- *literal* means that the transmitted data will conform to the details you specify in the WSDL `types` element and will be structured simply, as in this example.

```
<VIN>A123</VIN>
```

- *encoded* means that the transmitted data will reference data types in attributes, as in this example:
-

```
<VIN xsi:type="xsd:string">A123</VIN>
```

The binding element should specify *literal* in almost all cases.

For more information about your formatting choices, go to <http://www.ibm.com/developerworks>, and search for the quoted string "Which style of WSDL should I use?"

Let's briefly mention two other details:

- Each `operation` element in the `binding` element is associated with a same-named `operation` element in the `portType` element.
- The `soapAction` attribute is placed in the transmitted message for inspection by a receiving server but is being discontinued in WSDL 2.0. In most cases, assign an empty string.

Not shown in our example is reference to the SOAP `Header` elements that include Quality of Service details. You can specify an *explicit header*, in which case QoS details are defined in the `binding` element and in the elements that describe the service interface. The effect in one sense is quite simple: the requester includes QoS-related arguments in the service invocation. When an explicit header is in use, however, you're mixing QoS and business detail, and the mixing is undesirable:

- Over time, your company wants to benefit from a division of labor, with different people working on different kinds of detail, at best in different files that are combined to form a single WSDL definition. This division of labor allows use of people who specialize in security or other issues.
- The developer of the requester code shouldn't need to code QoS-specific arguments or even necessarily to know about the type of QoS data that must be transmitted. As much as possible, you want the developer to focus on business issues.

As an alternative, you can specify an *implicit header*, in which case the QoS details are described only in the `binding` element and have no effect on service parameters. Some analysts suggest that this option is the best available, as it separates QoS data definition from parameter definition, and an SOA runtime product can ensure that QoS details are included in the transferred message.

The problem in both cases is that you lose flexibility. Describing QoS data in the WSDL definition means that you're required to use QoS details whenever you deploy the service. You might want to deploy a service and use a security scheme in some cases (for example, when connecting with a business partner) but avoid a security scheme in others (for example, when the requester and service are in the same company).

The direction in the industry is to describe QoS details outside the WSDL definition. This book emphasizes the greater flexibility that comes from making QoS decisions at configuration time or even later, as when the requester and service negotiate the rules of their interaction at run time.

We'll return to the QoS issue later in the book, when we describe Service Component Architecture.

The service Element

The last major element of a WSDL definition is `service`, which completely describes the structure of every possible transmission. Listing 5.10 shows an example.

Listing 5.10. Sample service element

```
<wsdl:service name="MotorVehicleRecordsService">
  <wsdl:port binding="tns:MotorVehicleRecordsSOAPBinding"
    name="MotorVehicleRecordsSOAPPort">
    <soap:address location="http://www.dmv.org/" />
  </wsdl:port>
</wsdl:service>
```


Included are a set of `port` elements, each of which references a binding element by name and includes an address. The meaning of multiple `port` elements is that the related service can be accessed at any of several locations.

A WSDL port represents a deployed endpoint. An endpoint reference (which allows access of an endpoint at run time) is different from a port description primarily because the reference includes details that are meaningful only at run time.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

SOAP

SOAP is an XML format for transmitting data to and from a Web service. An automated tool creates the SOAP file, which includes details from the WSDL definition. For details on SOAP that go beyond our introduction, see the following primer: <http://www.w3.org/TR/2003/REC-soap12-part0-20030624>.

SOAP Format

The major elements in the transmitted SOAP message are as follows:

- the **Envelope** element, which is the root and usually includes namespace declarations.
- the **Header** element, which includes Quality of Service details. The **Header** element is optional; if it is present, each of its immediate children is called a *header block*.
- the **Body** element, which includes business data.

Listing 5.11 depicts the overall structure of a SOAP message.

Listing 5.11. Structure of a SOAP message

```
<Envelope>
  <Header>
    <!-- header blocks go here -->
  </Header>
  <Body>
    <!-- business data goes here -->
  </Body>
</Envelope>
```

Example

We created the SOAP output shown in Listing 5.12 using IBM Rational Application Developer.

Listing 5.12. Sample SOAP output

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:q0="http://www.dmv.org/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <q0:getMotorVehicleRecord>
      <VIN>A123</VIN>
      <State>NC</State>
      <Category>sport</Category>
    </q0:getMotorVehicleRecord>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The **Envelope** element start-tag includes namespace details. The prefixes **SOAP-ENV**, **xsd**, and **xsi** are commonplace,

even though this specific SOAP transmission doesn't use them. Use of the `q0` prefix (in place of the WSDL-specific `tns` prefix) is a detail specific to Rational Application Developer. At the other end of the transmission, the namespace Universal Resource Identifier (URI) is important, but the identity of the prefix that references the URI is not.

Header Blocks

As Web-based SOA runtimes become more advanced, header blocks will become more important; and the content of those blocks will increasingly conform to the open-standard *WS-** specifications being developed by various organizations.

The header block can include details that are needed to fulfill the kind of QoS options described in [Chapter 2](#). The block can also indicate whether conditional processing is required, as when the service returns a log of its activity during some invocations but not others.

You can set header blocks explicitly in the WSDL file that contributes to the SOAP message. More likely, however, is that your SOA runtime product adds header blocks in accordance with data structures and values that are established at configuration time.

The extra details provided by header blocks may not be meaningful at a particular intermediary or at the final destination. To allow you to control a variety of cases, SOAP lets you set the following attributes in any header block:

- The `mustUnderstand` attribute indicates whether a given *SOAP engine* (a kind of XML processor) must handle the header block. By default, the value of `false` is transmitted, which means that handling is not required.
- The `actor` attribute identifies the SOAP engine to which the `mustUnderstand` attribute applies. By default, `mustUnderstand` applies to the message's final destination.

A SOAP engine that handles a header block must remove that block, although the processor can always add the equivalent header block or any other.

In SOAP 1.2, the `role` attribute replaces `actor` and identifies, not an intermediary, but a specific role whose name and purpose are defined by an intermediary. As in Version 1.1, however, the default behavior is to process the header block at the message's final destination.

Like business data, the data in a header block usually conforms to an XML Schema definition.

SOAP Body

The SOAP body includes business data, such as that shown in [Listing 5.13](#).

Listing 5.13. Sample SOAP body

```
<SOAP-ENV:Body>
  <q0:getMotorVehicleRecord>
    <VIN>A123</VIN>
    <State>NC</State>
    <Category>sport</Category>
  </q0:getMotorVehicleRecord>
</SOAP-ENV:Body>
```

The data is structured in a way that reflects a WSDL format called *document-literal wrapped*, which means that

- on receiving the message, the service accesses the operation name (`getMotorVehicleRecord`) and can easily dispatch the message to that operation
- the elements lack the encoded data-type attributes (as shown earlier) that would otherwise degrade performance
- every data value can be validated by an XML Schema definition

SOAP at Run Time

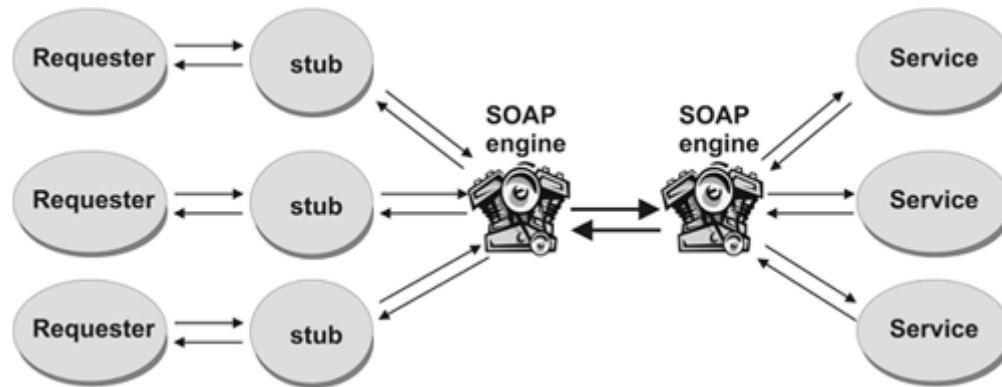
We now outline how the SOAP engine fits into a larger scheme.

The main purpose of the SOAP engine is to convert data between a language-specific format and the format used in the transmitted message. A data conversion is required on both the requester and service sides of the transmission.

When the requester is written in Java, the runtime events are often as shown in [Figure 5.2](#). On the client side, the requester invokes a JAX-RPC stub, which is Java code that your development environment created for you, in most

cases. The Java code is based on service-specific details in a WSDL definition and lets you invoke the service as if you were invoking a local function.

Figure 5.2. SOAP at run time



The stub frees the requester from interacting directly with the SOAP engine. In response to the stub's access, the engine converts data from the Java data types into a SOAP format and transmits the data.

On receiving a return message, if any (it may be business data or an error message), the stub uses the information from the WSDL definition again, in this case to convert data from a SOAP format into the data types expected by the requester.

On the service side, the SOA runtime code responds to its invocation by using a service-side WSDL definition to create data that the service can use. That WSDL definition is equivalent to the definition used during creation of the client-side stub.

The SOA runtime code submits the data to the service. When the service returns data, the SOA runtime first uses the service-side WSDL definition to convert data from the native-language data types into a SOAP format. The service then returns the message to the client.

When intermediaries are in use, the message is passed from one SOAP engine to another, and data conversions occur as needed when a message is transmitted first over one transport protocol, then over another.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

UDDI

UDDI is a set of rules for registering and retrieving details about a business and its services. As you design a program, you might search a UDDI registry for Web services to invoke in your code. You also might create a program that queries a UDDI registry at run time to access a set of similar services because (for example) each provides price information for a specific product. The latter use of UDDI is less common, however.

A UDDI registry might include details on business services (not just on software), but the major purpose of such a registry is to publicize what Web services are available in a particular domain such as a business or industry. The publicity exposes redundant Web services and promotes reuse.

Other purposes are possible, especially in products that build on the UDDI standard. Those UDDI-compliant products provide sophisticated user interfaces so that users are better able to register, review, and compare information. The products may track the planning and fulfillment of project tasks when a company is updating Web services so that the software complies with Service Level Agreements (SLAs) or industry standards. UDDI-compliant products also may hold additional details. Use of the word *repository* in some products suggests that SLAs, WSDL definitions, and supporting documentation of all kinds are immediately available and are not merely on a remote site that is referenced by the UDDI registry.

Among the categories of information in a UDDI registry entry:

- *business entity*, which usually includes
 - basic information such as business name and address
 - a set of classifications, such as by industry or by a category of products offered
 - a reference to the services provided by the business
- *publisher assertion*, which indicates a business relationship, as when one business is a subsidiary of another
- *business service*, which usually includes the service name and description
- *binding template*, which includes a service's access point such as a phone number (as might be appropriate for a non-software service) or a URL (for a Web service)

When interacting with a UDDI registry, you may be exposed to the following terms:

- *tModel* (technical model) is a data structure that references a specification such as a WSDL definition.
- *Category bag* is a list of entries. Each entry identifies an instance of a category, as in the following examples:
 - One category bag indicates that a business is in a particular geographical area (such as Connecticut) and handles a particular product (such as insurance).
 - A second category bag indicates that a service is described with a particular kind of specification (such as a WSDL definition) and fulfills a particular kind of purpose (such as providing a stock quote).
- *Identifier bag* is a list of similar entries — for example, a list of corporate tax IDs for use in different jurisdictions.

SOAP is the basis for the transfer of data to and from a registry. In the message shown in [Listing 5.14](#), the UDDI directive `get_businessDetail` retrieves business-specific information that was stored at some previous time.

Listing 5.14. Sample message containing a UDDI directive

```
<Envelope xmlns=
  "http://schemas.xmlsoap.org/soap/envelope/">
```

```
<Body>
  <get_businessDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
    <businessKey=
      "uddi:AB0E435D-890B-1358-6DE2-6349816453F5">
    </businessKey>
  </get_businessDetail>
</Body>
</Envelope>
```

For further details about UDDI, see the following Web site, which is operated by OASIS: <http://www.uddi.org>.