
Develop effective Notes 6, 7, and 8 consumers for .NET Web services

by Thomas Køcks



Thomas Køcks
Consultant

Thomas Køcks, a Domino consultant working in Denmark, has been developing Lotus Notes and Domino applications for more than 12 years. Thomas is a Java developer and an IBM Certified Professional for Lotus Software. His current projects include Web applications and WebSphere configurations for the IBM subsidiary EDB Gruppen A/S. You can contact Thomas at tkn@lotusdomino.dk.

Building Lotus Domino Web services that publish information from Notes databases has become a common task for many developers, thanks to the introduction in Domino 7 of the Web service design element. However, many developers are still using Domino applications only as providers of Web services, not as consumers of Web services, although the demand for consuming information is at least as great as the demand for providing information. One reason developers have been slow about writing Notes clients that can consume Web services (consumers or WS clients for short) is that doing so with release 6 or 7 Notes/Domino applications is still a bit complicated. Difficulties can arise because of inconsistencies between the standards used by Notes 6 or 7 and the standards used by some Web services providers.

Each integrated development environment (IDE) that can generate the Web Services Definition Language (WSDL) file for a Web service has peculiarities about how it interprets different standards. For example, the IDE may define objects differently from Java or LotusScript in Notes 6 or 7; arrays might or might not contain null elements; and the provider or consumer IDE's handling of special or complex data types can cause interoperability problems across environments.

Visual Studio .NET 2003 is the IDE companies most commonly use to generate WSDL files for Web services running in a Microsoft .NET framework environment. (From here on, I'll refer to this IDE as VS .NET and the framework and language as .NET.) Developers of Notes WS clients are very likely to encounter .NET Web services, so you need to know what problems might occur with consuming these services and what you can do about them.

Two standards that cause a great many consumer problems are usually not in the control of consumer developers: the routing style and the SOAP¹

¹ SOAP is a protocol for exchanging information in a decentralized, distributed environment. It's maintained by the World Wide Web Consortium (www.WC3.org). The word SOAP is no longer an acronym; it encompasses protocols for simple object access and a service-oriented architecture.

protocol used for requests and responses.² For example, .NET (and other) Web service providers increasingly use SOAP 1.2 because it offers better developer support than SOAP 1.1, but, as you'll see, it's not possible for Notes 6 or 7 Java-based consumers to handle responses generated with SOAP 1.2. In this article, you'll learn how to discover a Web service's SOAP setting and what to do in the case of incompatibility with Notes 6 or 7. You'll also learn how to identify the routing styles used by Web services, the problems these settings can trigger, and what you can do to avoid or correct the issues in Notes 6 or 7 clients.

Difficulties that arise with these standards are also specific to the programming techniques for crafting consumers. Understanding the issues that popular techniques bring to consuming .NET Web services, and what you can do to eliminate these issues, helps you to design a successful Notes 6 or 7 consumer. To illustrate the problems and solutions, I developed Notes 7 consumer examples (that also work in Notes 6) using the two most common approaches to consuming Web services: LotusScript code using a Microsoft SOAP object and Java code using proxies generated from the WSDL file. Both of these approaches have been explained in detail in previous articles of *THE VIEW*³ (albeit with earlier Domino releases). Therefore, my focus in this article is not on the example code, which is simple, but on the results of consuming in these two ways so you can better understand your best WS client design choices.

This article also introduces you to the new Web service consumer feature in Notes 8. I describe how easy it is to create Notes 8 consumers using Domino Designer 8 beta 2, and how the Notes 8 consumer programs easily handle the .NET settings that cause difficulties in Notes 6 and 7. You'll gain insight into the value of upgrading your Notes/Domino

² The provider developer chooses the routing style and the SOAP protocol settings in VS .NET when generating the WSDL file.

³ For more information on using the Microsoft SOAP Toolkit for consumers, see the article "Creating and Using Web Services with Domino" (*THE VIEW*, May/June 2003). For details on using Java proxies, see the articles "Consume Web services in Lotus Notes applications, Part 1: Preparing the Java proxy" (*THE VIEW*, July/August 2006) and "Consume Web services in Lotus Notes applications, Part 2: Deployment" (*THE VIEW*, September/October 2006).

environment from seeing the consumer code, techniques, and results in the examples.

Note that this article isn't just for developers of Web consumer clients — if you provide Web services, you can pick up knowledge about the consequences your Web service design choices have for consumers. This knowledge can help you to anticipate possible problems and lead you to provide appropriate help for consumers. This article also provides you with some tricks for converting the Web services that you provide or have control over to SOAP 1.1, if that is required by your consumers.

All of the LotusScript and Java consumer agent code and the WSDL files for the example Web services are in the .NET Web Services database (WebServ.nsf) which you can download from *THE VIEW* Web site.⁴

This article assumes a basic knowledge of the standards involved in Web services, such as SOAP, WSDL, and so on. (A list of resources at the end of this article offers more information about some of the key concepts involved in creating Web service consumers.) To establish a common foundation for understanding the results from the consumer examples, I start by explaining how .NET Web service providers usually set two crucial settings.

The basics of SOAP and routing settings in .NET Web services

This section familiarizes you with the routing style and SOAP protocol settings every Web service has to provide and explains which routing style and SOAP protocol .NET Web service providers choose most often.

⁴ You can download the WebServ.nsf database for the agent code and WSDL files at www.eVIEW.com. From the home page, click Search → Browse Issue → November/December 2007 → Develop effective Notes 6, 7, and 8 consumers for .NET Web services. Scroll to the bottom of the page for the download files.

Routing styles

When creating a Web service in VS .NET, a developer must choose a routing style for structuring a SOAP request and a SOAP return message. VS .NET stores this setting in the web.config file. Regardless of the IDE, this attribute has two possible values:

- **Document service** — defined by SoapDocumentServiceAttribute in the Web service code. For example, the code line SoapDocumentService(RoutingStyle = SoapServiceRoutingStyle.SoapAction) indicates that the Web service or Web service consumer can validate the Extensible Markup Language (XML) in a request or response message, respectively, using a schema. Some refer to this style as literal or doc/literal. With Document service routing, a Web service provider can structure the contents of the message body in various ways, and there can be any number of parameters or documents in the <env:Body> node of a response message.
- **Remote Procedure Call (RPC) service** — defined by SoapRPCServiceAttribute in the Web service code. For example, the code line SoapRPCService(RoutingStyle = SoapServiceRoutingStyle.SoapAction) indicates that the contents of the XML message body are structured using rules specified in a SOAP protocol. Some developers refer to this style as encoding or RPC/encoded.

RPC is the oldest way of generating Web services and is widely deployed. Note that the RPC service routing style is a special case of the Document service routing style. There is only one child of the <env:Body> node, which is the RPC element. All sub-elements are encapsulated within this single element, and the sub-elements have the same names as their counterpart parameters in the .NET code.

The SOAP protocols

From the IDE, every Web service provider specifies the SOAP protocol to use for generating responses. The IDE stores the setting (HTTPSoap for SOAP 1.1 or HTTPSoap12 for SOAP 1.2) in the web.config file. The SOAP 1.2 protocol, released by the World Wide Web Consortium in June 2003, has more precise rules than SOAP 1.1 for processing SOAP messages. It also has improved error messages for helping Web service developers. Web service development tools like Delphi and VS .NET have the support necessary for creating Web services compliant with SOAP 1.2. Web service developers often prefer SOAP 1.2 because these changes make it easier to work with.

In **Figure 1**, you can compare two XML representations of the same fault code encoded by each of the SOAP protocols. Note that the structures of the XML documents differ from each other. Notice also that the

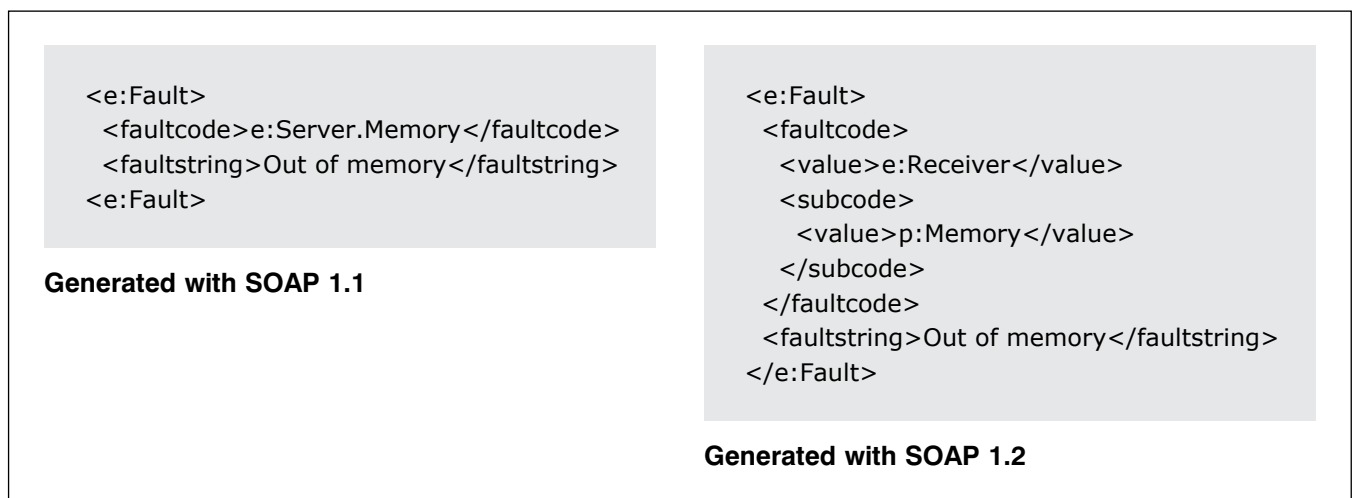


Figure 1 Comparison of documents generated with different SOAP protocols for the same fault code

SOAP 1.2 structure provides more levels in the code hierarchy.

A Web service set to use SOAP 1.2 in its responses must be called by a client that is capable of handling SOAP 1.2 responses. Axis, the framework that Java code typically uses when handling SOAP messages, offers only the SOAP 1.1 protocol; Axis2, which was released early in 2007, uses SOAP 1.2. Notes 6 and 7 use Axis. Notes 8 is capable of using either Axis or Axis2.

Note that Stubby, the Notes Java stub generator that many Notes developers use, currently uses SOAP 1.1. Therefore, as of this writing, it's not possible to use Stubby to generate stubs when writing a Notes 6 or 7 Java agent that complies with SOAP 1.2. The good news, though, is that Notes 8 developers don't need Stubby or any other IDE — Domino Designer 8 creates its own LotusScript or Java proxies that are fully compatible with SOAP 1.2 or SOAP 1.1.

Before getting to the consumer test runs, you might want to know how the Web service examples and the consumers were set up. The next two sections describe enough about the Web service designs for you to understand what the consumer examples test. I recommend that, at a minimum, you read the subsections "The SOAP protocol definitions" and "The routing service definitions," which show you

what to look for in a WDSL file to discover a Web service's SOAP version and routing style.

The Web service examples

To build all of the components for the Web service side, I used Microsoft's VS .NET and the Microsoft .NET framework software development kit (SDK). I built one .NET Web service and generated four different flavors of WSDL files from the same Web service so I could compare outcomes in the discussions. (I don't discuss how to build a Web service provider with VS .NET in this article.) I ran test Web services on my own Internet Information Services (IIS) server; you can see their addresses, [http://vmarhwebsrv/CarsTestWS/\[nameOfWebService\]](http://vmarhwebsrv/CarsTestWS/[nameOfWebService]), throughout the figures.

The base .NET Web service is very simple. It has three methods the WS clients can call to show variations in how responses with different data types (single-value string, multi-value string, and complex) are consumed in different circumstances:

- **getApprovedCar(int myCar)** — receives a simple integer XML type (xsd:int) parameter as an argument and returns the make of a car approved for resale as a single value of the string XML type (xsd:string). **Figure 2** shows example

THE XML REQUEST :

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tns="http://schemas.edbgruppen.dk/2007/1/CarsTestWS/CarsRPC" xmlns:types="http://schemas.edbgruppen.dk/2007/1/CarsTestWS/CarsRPC/encodedTypes" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <tns:getApprovedCar>
      <carId xsi:type="xsd:int">1</carId>
    </tns:getApprovedCar>
  </soap:Body>
</soap:Envelope>
```

Continues on next page

Figure 2 Using the getApprovedcar (int myCAR) method

THE XML RESPONSE:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.
w3.org/2001/XMLSchema" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:
tns="http://schemas.edbgruppen.dk/2007/1/CarsTestWS/CarsRPC" xmlns:types="http://schemas.edb-
gruppen.
dk/2007/1/CarsTestWS/CarsRPC/encodedTypes" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <tns:getApprovedCarResponse>
      <getApprovedCarResult xsi:type="xsd:string">Saab</getApprovedCarResult>
    </tns:getApprovedCarResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 2 (continued)

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.
w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getApprovedCarsResponse xmlns="http://schemas.edbgruppen.dk/2007/1/CarsTestWS/Cars">
      <getApprovedCarsResult>
        <string>Saab</string>
        <string>Renault</string>
        <string>Volvo</string>
        <string>Ford</string>
        <string>Porsche</string>
        <string>Mitsubishi</string>
      </getApprovedCarsResult>
    </getApprovedCarsResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 3 The XML response to a getApprovedCars method call

code for an XML request using the integer 1 as the method's parameter. The response is a single car make, Saab.

- **getApprovedCars** — takes no parameter as an argument and returns as a string array (xsd:string) a list of cars that have been approved for
- **getCar(int myCar)** — receives an integer parameter as an argument and returns a complex XML type, the class Car with three string properties:

resale. For example, a response to a call to this method would look like the XML response in **Figure 3**.

make, model, and color. For example, **Figure 4** shows the response to calling this method with the parameter 1. This returned value is more complex than a string array and requires more handling.

From this base code in VS .NET, I generated different WSDL files.

Four WSDL files vary SOAP protocols and routing styles

I created four different WSDL files from the .NET code for the Web service I just described. I set two WSDL files to use the SOAP 1.1 protocol for encoding responses to the consumer client, and I set the other two WSDL files to encode responses with the SOAP 1.2 protocol. Then, for each pair of WSDL files using the same SOAP protocol, I set one file to use the RPC service routing style and the other file to use the Document service routing style. The four WSDL files are summarized in **Figure 5**.

To design an effective consumer, you must first recognize these settings within the WSDL file and know how they influence responses. To save space, I show file excerpts; you can examine the four WSDL files in their entirety in the WSDLFilesNumbered.rtf file (downloadable at THE VIEW Web site).

So how can you tell from a WSDL file what SOAP and routing settings the Web service is using?

The SOAP protocol definitions

Figure 6 shows a comparison of the different SOAP encodings of the bindings in two of the WSDL files for the example Web service. Both files use the Document routing style, so that's not a factor here. *Org_Cars.wsdl* represents the Web service generating SOAP 1.2 responses, and *Org_CarsSoap11.wsdl* represents the Web service generating SOAP 1.1 responses.

The `<soap:address location="[URL]">` tag identifies the SOAP binding address that the client's Web server must use to direct all requests to a certain endpoint address. (The endpoint in all of the examples is the Web service defined by these WSDL files.)

In the first part of the location tag, `soap:` or `soap12:` identifies the SOAP protocol that the client's Web server must use to access the endpoint (SOAP 1.1 or SOAP 1.2, respectively).

Notice in the *Org_Cars.wsdl* file that these tags identify *both* SOAP protocols. That's because SOAP 1.2 is backward compatible, and any Web service using it to generate responses can take request calls formatted using either SOAP version.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.
w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getCarResponse xmlns="http://schemas.edbgruppen.dk/2007/1/CarsTestWS/Cars">
      <getCarResult>
        <Model>93</Model>
        <Make>Saab</Make>
        <Color>Black</Color>
      </getCarResult>
    </getCarResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 4 The XML response to a `getCar` method call

The xmlns:soap12= attribute in the <wsdl:definitions> header tags of both files defines a SOAP 1.2 namespace, the context in which the methods are defined when tools like Rational Application Developer (RAD) use the WSDL file for developing stubs or proxy classes. Stubby doesn't use SOAP 1.2, so you can anticipate a problem with Stubby generating stubs from the WSDL file because of the file's reference to SOAP 1.2 unless the developer is able to take action to change the Web service. (I'll explain more when discussing the consumers.)

Now let's examine the differences in how the routing services are defined in the WSDL files.

The routing service definitions

Figure 7 shows a comparison of two WSDL files that use different routing styles. Both files use SOAP 1.2 so the SOAP version is not a factor. On the left is an excerpt of Org_CarsRPC.wSDL, the file using the RPC service. On the right is the corresponding part of Org_Cars.wSDL, the file using the Document service.

WSDL file name in the download	Routing style	SOAP protocol for responses
Org_Cars.wSDL	Document service	SOAP 1.2
Org_CarsRPC.wSDL	RPC service	SOAP 1.2
Org_CarsSOAP11.wSDL	Document service	SOAP 1.1
Org_CarsRPC11.wSDL	RPC service	SOAP 1.1

Figure 5 The four WSDL files for the same .NET Web service

```

<?xml version="1.0" encoding="utf-8" ?>
- <wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tns="http://schemas.edbgruppen.dk/2007/1/CarsTestWS/CarsSoap11"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  targetNamespace="http://schemas.edbgruppen.dk/2007/1/CarsTestWS/CarsSoap11">
+ <wsdl:types>
+ <wsdl:message name="getApprovedCarSoapIn">
+ <wsdl:message name="getApprovedCarSoapOut">
+ <wsdl:message name="getCarSoapIn">
+ <wsdl:message name="getCarSoapOut">
+ <wsdl:message name="getApprovedCarsSoapIn">
+ <wsdl:message name="getApprovedCarsSoapOut">
+ <wsdl:message name="getApprovedCarHttpPostIn">
+ <wsdl:message name="getApprovedCarHttpPostOut">
+ <wsdl:message name="getCarHttpPostIn">
+ <wsdl:message name="getCarHttpPostOut">
+ <wsdl:message name="getApprovedCarsHttpPostIn"/>
+ <wsdl:message name="getApprovedCarsHttpPostOut">
+ <wsdl:portType name="CarsSoap">
+ <wsdl:portType name="CarsHttpPost">
+ <wsdl:binding name="CarsSoap" type="tns:CarsSoap">
+ <wsdl:binding name="CarsSoap12" type="tns:CarsSoap">
+ <wsdl:binding name="CarsHttpPost" type="tns:CarsHttpPost">
- <wsdl:service name="Cars">
- <wsdl:port name="CarsSoap" binding="tns:CarsSoap">
  <soap:address location="http://vmarhwebsrv/CarsTestWS/" />
</wsdl:port>
- <wsdl:port name="CarsSoap12" binding="tns:CarsSoap12">
  <soap12:address location="http://vmarhwebsrv/CarsTestWS/" />
</wsdl:port>
- <wsdl:port name="CarsHttpPost" binding="tns:CarsHttpPost">
  <http:address location="http://vmarhwebsrv/CarsTestWS/" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
  
```

Org_Cars.wSDL uses SOAP 1.2

```

<?xml version="1.0" encoding="utf-8" ?>
- <wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://schemas.edbgruppen.dk/2007/1/CarsTestWS/CarsSoap11"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  targetNamespace="http://schemas.edbgruppen.dk/2007/1/CarsTestWS/CarsSoap11">
+ <wsdl:types>
+ <wsdl:message name="getApprovedCarSoapIn">
+ <wsdl:message name="getApprovedCarSoapOut">
+ <wsdl:message name="getCarSoapIn">
+ <wsdl:message name="getCarSoapOut">
+ <wsdl:message name="getApprovedCarsSoapIn">
+ <wsdl:message name="getApprovedCarsSoapOut">
+ <wsdl:portType name="CarsSoap11Soap">
+ <wsdl:binding name="CarsSoap11Soap" type="tns:CarsSoap11Soap">
- <wsdl:service name="CarsSoap11">
- <wsdl:port name="CarsSoap11Soap" binding="tns:CarsSoap11Soap">
  <soap:address location="http://vmarhwebsrv/CarsTestWS/CarsSoap11.asmx" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
  
```

Org_CarsSOAP11.wSDL uses SOAP 1.1

Figure 6 Comparing the binding sections of WSDL files using different SOAP protocols but the same routing style (Document service)

Let's take for example the `getApprovedCar(int myCar)` method defined in the .NET Web service. Recall that the Web service receives a simple type parameter as an argument and returns to the client the make of a car as a single value (a simple string, `xsd:string`). Notice some differences in how the RPC and Document services are defined for the methods in the WSDL files:

- `<wsdl:service name>` sets the name of the Web service being called. The WSDL file with Document service routing uses the name "Cars" and the WSDL file with RPC service uses the name "CarsRPC."
- `<wsdl:part name>` under `<wsdl:message name="getApprovedCarSoapIn">` sets the name of the request message going into the Web service. It defines the parameter expected from the WS client for calling the `getApprovedCar()` method. For the XML code using the Document style, the line is:

```
wsdl:part name='parameters' element='tns:
getApprovedCar'
```

For the Web service using RPC, the equivalent line is:

```
wsdl:part name='carId' type='s:Int'
```

The difference in these lines results in a slightly different response, as you'll see later. Essentially, the WSDL file for the Web service using the Document style (the Document service) defines `wsdl:part` name as the parameters of the element `tns:getApprovedCar` (`tns` is this namespace). Therefore, the consumer (an agent in the examples) has to look in the types section of the WSDL document for a specific definition of the parameter(s) allowed for `getApprovedCar`.

The Document style uses loose coupling between the calls from the client and responses from the Web service, and the Web service calling function must take a parameter defined with a complex type as shown in **Figure 8** (in contrast to the integer of the RPC style). If the Web service provider changes the number of parameters or the type of a parameter (e.g., from `int` to `string`), the consumer developer just changes the object being called, not the calls themselves. A complex type parameter, though, is not Java-friendly, so providers and consumers programmed with Java often adopt the RPC routing style.

The RPC style requires the client to send a value for the variable `carId` using the XML integer type `s:Int`, which makes the response descriptive and Java-friendly (i.e., with RPC routing, the WSDL names in the call map one-to-one, following Java naming

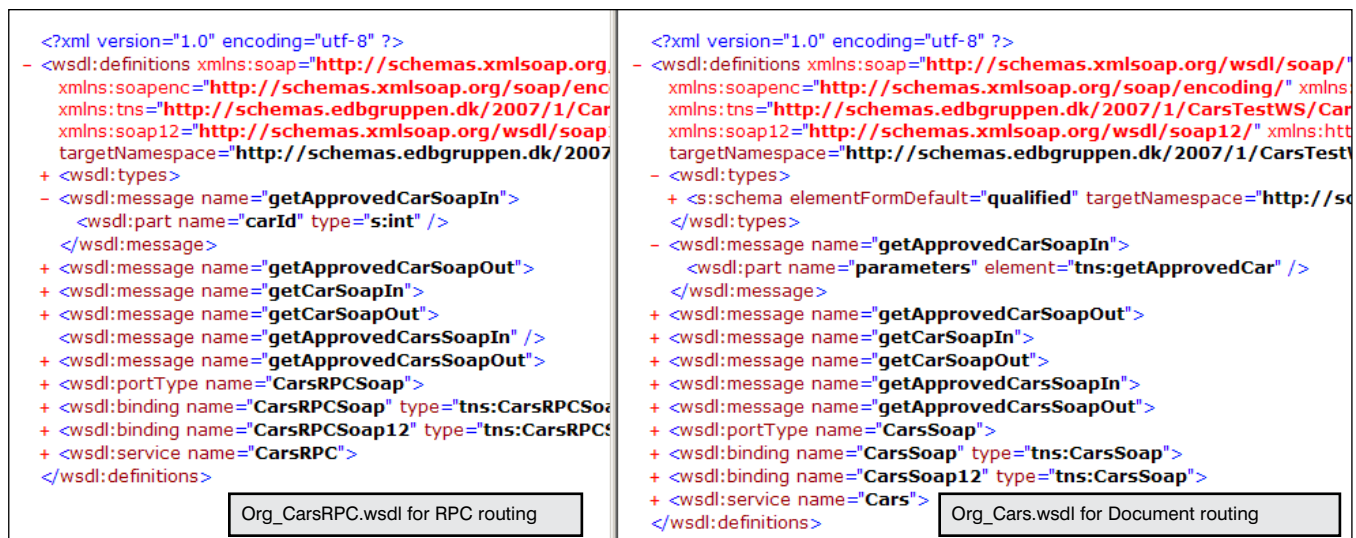


Figure 7 Comparing the `getApprovedCar` Web service method definitions in WSDL files using different routing styles but the same SOAP protocol (1.2)

conventions). This tight coupling makes the parameter definition more descriptive, as the type of parameter is included in the part name. However, if the Web service changes to a different type of parameter or adds or removes parameters, the consumer developer has to change the calls to the Web service.

Now you know what the settings look like in the WSDL files and some of the different ways they influence a Web service's response to calls. Next, I'll briefly explain the two calling techniques used in the tests.

The consumer examples: Demonstrating two calling techniques

There are several ways to make Notes act as a Web service consumer. Most techniques are complex; for the purpose of testing the results from the different flavors of the example Web services just described, I use the two simplest (and therefore most common) techniques:

- Using a Microsoft (MS) SOAP object and writing LotusScript calls to the Web service.⁵ LotusScript developers usually prefer to work with the MS SOAP object for creating Notes

⁵ To learn more about this technique, read "Creating and testing Web services with the new design element in Domino 7" (*THE VIEW*, March/April 2006).

consumers because it's easy to implement and doesn't require an IDE. However, this technique is more vulnerable than using a Java agent because the Microsoft .NET Framework has deprecated this dynamic link library (DLL) object (I say more about this later.) Another drawback is that calls made with this method take longer to execute than calls made with Java.

- Using proxy classes and writing Java calls to the Web service. The developer uses an IDE to generate Java proxy classes from a WSDL file, imports the resulting Java Archive (JAR) file to a Domino application, and uses the proxy class methods from the JAR file in Java calls to the Web service. (See the "Generating Java proxy classes" sidebar for more information.)

Each design approach has different SOAP or routing issues to address when you use it to consume Web services:

- Because Java isn't involved in the LotusScript example, the SOAP version isn't an issue with the LotusScript/MS SOAP object technique, so the LotusScript consumer tests illuminate what the programmer has to handle for each of the two different routing styles.
- Because Notes 6 and 7 use Axis, I tested Notes consumers with providers that use SOAP 1.1. The Java consumer tests reveal what happens in the Notes consumer when the .NET Web service is set to use either routing style.

```
<s:element name="getApprovedCar">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="carId" type="s:int" />
    </s:sequence>
  </s:complexType>
</s:element>
```

Figure 8 Org_Cars.wsdl with Document routing, the complex type definition for generating responses to getApprovedCar requests

- Given that a SOAP 1.2 Web service theoretically should be able to respond to calls formatted as SOAP 1.1, I tested that as well, using the SOAP 1.2 Web service with the RPC routing style.

I ran the tests using Notes 7 (those results apply equally to Notes 6) and again in Notes 8. Let's begin with the Notes 7 tests.

Note!

All of the example consumer code consists simply of Web agents that are run from the Domino Designer client used to create them. This article focuses on illustrating issues and what can be done about them, not on creating implementation-worthy consumer code.

Using Notes 6 or 7 as a .NET Web service consumer

For each of the two calling techniques, I explain enough about the consumer agents to enable Notes developers to understand exactly what each agent tests. Then I explain the results, highlighting problems encountered and what, if anything, the developer can do to mitigate them.

Consuming SOAP 1.1 services with different routing styles from LotusScript with the MS SOAP object

To generate results using the MS SOAP object, I wrote the SOAP.NET agent using Domino Designer 7 and ran it from a Notes 7 client. (The agent can also run in Notes 6 or 6.5.) The agent called each of the two SOAP 1.2 Web services that use different styles of routing the responses. **Figure 9** shows the agent code (also in the download), which is explained next.

Generating Java proxy classes

There are several solutions I could have used for generating Java proxy classes from WSDL files. The most common solutions Notes 6 or 7 developers use are the following:

- The Web service feature in Eclipse or Rational Application Developer (RAD). This solution requires installation of Eclipse or RAD,* both of which offer a choice of Axis or Axis2. However, Axis2 is incompatible with the Axis-based Java in Notes 6 and 7.
- The open-source project Stubby,** which is a Notes database for creating Apache Axis files called “stubs” (another name for proxy classes — in a client-server model, the client implements stubs of the interface's methods and the server implements the interface's methods in full). This solution is free, it does not require installation of Eclipse or RAD, and Notes developers can use the Axis-based stubs it generates in Notes/Domino 6+ applications without modifying the client or server. Therefore, it's popular among Notes/Domino shops. However, it only recognizes SOAP 1.1, and it fails to generate any stubs for Web services set to SOAP 1.2.

These two solutions have one thing in common: They both generate the proxy classes with the Apache Axis framework and the Apache WSDL2Java tool. For the Java consumer agents that run in Notes 6 and 7, I chose to use Stubby to generate SOAP 1.1-compatible stubs.

* If you have access to the LotusSphere 2007 sessions, look at BP201.pdf by Paul Calhoun for an example using Eclipse/RAD.

** To learn more about this technique, read “Consume Web services in Lotus Notes applications, Part 1: Preparing the Java proxy” (*THE VIEW*, July/August 2006).

- **Lines 2 – 6** declare the variables to call each of the Web services. The MS SOAP object is defined as client with the data type Variant. (Because LotusScript supports only dynamic binding,⁶ it's best to declare a Component Object Model, or COM, variable in this manner.)
 - **Lines 7 – 8** instantiate two string variables with the URLs of the two WSDL files for the Web service. The wsdl variable points to the file using Document routing, and wsdlRPC points to the file using RPC routing.
 - **Line 9** instantiates client, the MS SOAP object.
 - **Line 10** initializes the MS SOAP object with the desired Web service to call. In this case, it's using the WSDL file with the Document routing style.
- ⁶ Dynamic (or late) binding links a routine or object to types at runtime, whereas static (or early) binding assigns types to variables and expressions at compilation time. If you call a method and make a mistake in the method name, early binding tells you when you compile the calling program (or when your IDE automatically compiles the program as you type), but dynamic binding will not tell you.

```

1. Sub Initialize
2. Dim wsdl As String
3. Dim wsdlRPC As String
4. Dim strResponse As String
5. Dim i As Integer
6. Dim client As Variant

7. wsdl="http://vmarhwebsrv/CarsTestWS/Cars.asmx?WSDL"
8. wsdlRPC="http://vmarhwebsrv/CarsTestWS/CarsRPC.asmx?WSDL"

9. Set client = CreateObject("MSSOAP.SoapClient30")
10. Call client.MSSoapInit(wsdl)

11. ' Call first Web service method
12. MsgBox client.getApprovedCar(1)

13. ' Call second Web service method
14. Forall myCar In client.getApprovedCars()
15.     MsgBox myCar
16. End Forall

17. ' Call third Web service method
18. Dim Nodes As Variant
19. Set Nodes = client.getCar(1)

20. For i = 0 To Nodes.length()-1
a. strResponse = strResponse + Nodes.item(i).nodeName() + _
b. "=" + Nodes.item(i).text() + Chr(13)
21. Next
22. MsgBox strResponse
23. End Sub

```

Figure 9 The SOAP.NET LotusScript agent code

- **Lines 11 – 12** call the Web service’s `getApprovedCar` method with the parameter 1. You might recall from the overview earlier that the Web service should return a message box with the first car from the sample database, Saab.
- **Lines 13 – 16** call the next Web service method, `getApprovedCars`. This calling method does not require any parameters and returns a list of cars as a string array to a message box.
- **Lines 17 – 22** call the last Web service method. These lines loop through the nodes of the XML tree returned from the Web service. The temporary variable `strResponse` stores information about each node and returns that information to a message box.

To call Web service routing with RPC, I changed **line 10** to the following:

```
Call client.MSSoapInit(wsdlRPC)
```

This syntax change to RPC doesn’t change much in the LotusScript code, and both routing styles work. However, when called, the Web services returned slightly different results.

The results were fine for the `getApprovedCar` and `getApprovedCars` method calls, so I won’t show them. However, for the `getCar` method, the Web service returned different results for each routing style (see **Figure 10**). The RPC service response includes the model identifier and a type, and the Document service response lacks these items.

The routing styles have different ways of structuring the response when the Web service returns a complex XML type (in this case, the Car class with three string properties). This difference matters when you’re manipulating the response data with LotusScript. To assign Web service responses to correct variables and objects in Notes 6 or 7, you must look at the WSDL files and determine the routing style of the Web service you are calling (Document or RPC) and find out the exact structure of the response. The best way to determine the structure of the response is to capture the response before trying to use it (i.e., print the response to a message box or in a variable or use the `tcpmon` tool, discussed in the “`tcpmon`” sidebar.) If your code instantiates a response object in LotusScript that matches an RPC-style response, that code won’t work if the Web service returns a Document-style response. You must create the response object based on the expected response.

Later, you’ll see that when using stubs or proxies for Java calls, you map the objects directly, meaning you access object properties, such as `.getModel`, to access the car’s model name. You don’t need to create the response objects — Stubby generates them for you from the WSDL file, in accord with the file’s routing style setting.

Another important issue with the viability of this design approach involves the MS SOAP Toolkit, which has the following limitations:

- Requires the Win32 operating system

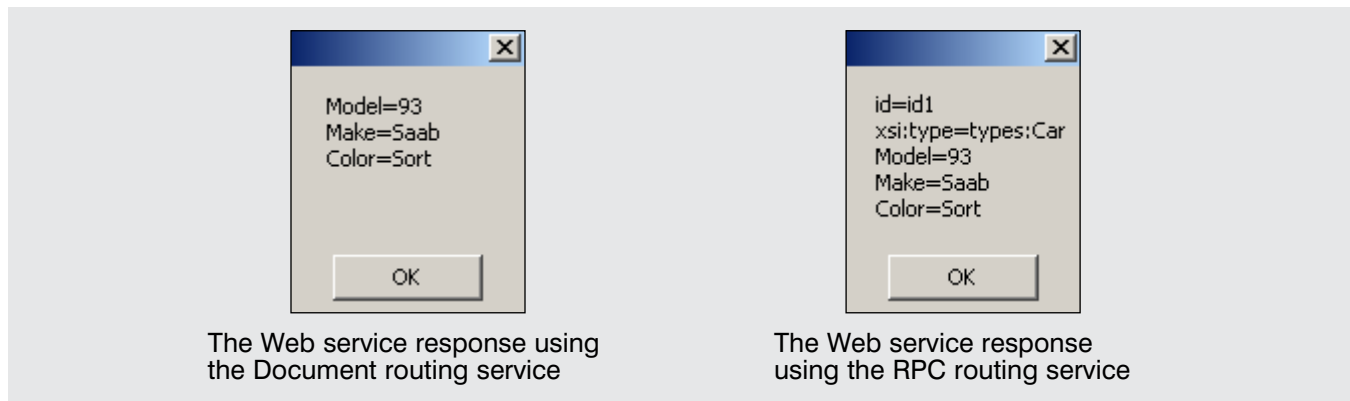


Figure 10 Compare the information returned to a LotusScript agent in Notes 7 from Web services using the same method but different routing styles.

tcpmon: Troubleshoot Web service calls from Notes 6, 7, or 8 consumers

The free, open-source tcpmon* utility monitors the calls sent to and the responses received from Web services. Web service development environments, including Domino Designer, do not usually provide such a tool. The tcpmon utility is supported on Windows XP/2000, MacOSX, Linux, and Solaris. You can run it directly from a browser or download it as an executable file.

I find tcpmon essential for troubleshooting my Web service call designs. A developer sets up tcpmon as a listener between a Web service consumer (the requesting client) and a provider. The client connects to a local Transmission Control Protocol (TCP) port where tcpmon is also connected. When you make a call from the client, the process flow is as follows: consumer client → tcpmon → provider → tcpmon → consumer client. During this flow, tcpmon performs the following tasks:

- Interprets and shows the Extensible Markup Language (XML) request from the consumer client. You see whatever XML format is being transmitted (e.g., tcpmon shows if the call or request is in SOAP format)
- Regenerates and shows the call to the real Web service
- Interprets and shows the XML that the Web service returns
- Returns the result to the original consumer client

Below, you can see the results of a sample call using tcpmon listening on port 8088. The top results section of the window shows message statistics for port 8088, such as status of the call and elapsed time to get the response. The middle section shows the SOAP request message that I sent directly to the Web server using the Sender tab of tcpmon. The last section shows the SOAP response that the Web service returned through tcpmon, which tcpmon then passed on to the consumer.

* You can find more information on tcpmon at this Web page: <https://tcpmon.dev.java.net/>

The screenshot shows the TCPMon utility window with the following content:

```

TCPMon
Admin | Sender | Port 8088
[Stop] Listen Port: 8088 Host: fead.webservice Port: 80 [Proxy]
-----
State | Time | Request Host | Target Host | Request... | Elapsed Time
-----
Most Recent
Done | 2007-05-22 13:14:26 | localhost | fead.webservice.udv.geodata.dk | POST /jsservice/fead.aspx?WSX HTTP/1.0 ... 531
-----
[Remove Selected] [Remove All]
-----
<?xml version="1.0" encoding="utf-8"?><soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
<soap:Body>
<DeleteLocalisation xmlns="http://fead.webservice.geodata.dk/">
<DeleteLocalisationRequest>
<localisation>
<localisationIdentifier>min uliniknogle</localisationIdentifier>
<systemIdentifier>GoPro ESDN</systemIdentifier>
</DeleteLocalisationRequest>
</DeleteLocalisation>
</soap:Body>
</soap:Envelope>
-----
HTTP/1.1 200 OK
Connection: close
Date: Tue, 22 May 2007 11:15:19 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
X-AspNet-Version: 2.0.50727
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 414
<?xml version="1.0" encoding="utf-8"?><soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsi="h
-----
[XML Format] [Save] [Resend] [Switch Layout] [Close]

```

The tcpmon utility for monitoring TCP connections

- Provides limited support for MS Windows Server 2003
- The Microsoft .NET framework has deprecated the toolkit; .NET no longer supports versions of the MS SOAP Toolkit prior to 3.0, and it will only support MS SOAP 3.0 until March 31, 2008⁷

Because the MS SOAP object is deprecated, I strongly recommend using Java (which has no dependency on the MS SOAP object) if the .NET Web service uses SOAP 1.1. (I'm getting a little ahead of the results here.) If you plan to consume .NET Web services using SOAP 1.2, I recommend that you move to Notes 8, which has no problem with SOAP 1.2 Web services, whether you use a LotusScript or Java WS client.

If you can't move to Notes 8 yet and want to use the MS SOAP solution, you have two implementation choices. One involves downloading and installing the MS SOAP Toolkit 3.0 on all Notes clients that call the Web service. You can use the toolkit's installation

wizard to automate that task for you. However, many companies have concerns about adding applications to maintain and support, so your IT department may not be receptive to installing this toolkit on every client.

Another approach you could use is to let the Domino Web server handle the SOAP requests and instantiate the MS SOAP object. In this solution, you create a LotusScript agent (a servlet) on the Web server to handle all calls to the .NET Web service. The LotusScript client then uses HTTP requests to communicate with the LotusScript agent instead of directly with the Web service. In this scenario, you don't need to install the MS SOAP toolkit on each client, although you do have to create more code. The additional code must handle authentication differently and manage the client-to-servlet communications. You also must make sure that the Domino Web server is running on a Win32 operating system that supports the MS SOAP object.

There is an alternative approach that enables you to use a LotusScript consumer. See the sidebar "Another design approach: Create a custom .NET control with a COM wrapper to use with LotusScript or Java code" for more information on this method.

⁷ For more information on Microsoft support for MS SOAP 3.0, visit this Web site: <http://support.microsoft.com/kb/811215/>

Another design approach: Create a custom .NET control with a COM wrapper to use with LotusScript or Java code

To make Notes 7 act as the Web service consumer, you can create a custom .NET control with a COM wrapper and then write LotusScript code that calls the .NET class library using the COM wrapper. With this technique, you can use Java or LotusScript (without the deprecated SOAP object) for the consumer code. This technique works well.

I didn't test it for you in this article because it doesn't raise issues and it's not one of the top two design choices Notes developers use.

Although it works well, there are several downsides to this technique:

- You need to maintain and update your own COM object.
- As with the SOAP toolkit, you need to distribute and register the COM object wherever it is used — meaning with all clients, if you are calling the Web service directly from the Notes client. You can work around this in the same manner you work around solving SOAP distribution, mentioned in the section "Using Notes 6 or 7 as a .NET Web service consumer."
- This approach is slower than calling a Web service natively using Java.

Consuming SOAP 1.1 services with different routing styles from SOAP 1.1 Java agents

To test consuming services with different routing styles from Java, I created and ran two SOAP 1.1-compatible Java agents. The PROXY.NET SOAP1.1 agent calls `Org_CarsSOAP11.wsdl` (the Document service using SOAP 1.1). The PROXY.NET SOAP1.1 RPC agent calls `Org_CarsRPC11.wsdl` (the RPC service using SOAP 1.1).

For each WSDL file, I followed this process to create the corresponding Java consumer agent:

1. From the Notes client, use the Stubby Notes database to generate stubs for the WSDL file. Detach the JAR file to the local drive. (See the sidebar “Using Stubby” for a quick overview of how to do this.)
2. In Domino Designer, create a new Java agent for using the stubs. Copy the sample code from Stubby into the programming pane. Insert response-handling code and calls to the Web service.
3. Attach the new JAR file to the Java agent.
4. Save and compile the agent.
5. From Domino Designer, run the Java agent.

Note!

Refer to the “Routing styles” sidebar to see how a provider sets a Web service to use either RPC or Document style.

To see how the agents work with definitions from the WSDL files, let’s examine the part of the PROXY.NET SOAP1.1 RPC agent code that calls the `getCar` method (shown in **Figure 11**).

Lines 1 – 5 import packages that are needed to call the Web service. These lines define standard packages created by Stubby. Note that to deploy this agent as a

Using Stubby

For information about the Stubby database and how it generates proxy classes, refer to the OpenNTF.org Web site.* You can download Stubby by clicking the link under the Releases section of the page.

Teaching Stubby, the Java stub generator Notes application, is not the purpose of this article, so I’ll just outline the steps to follow for creating the stubs for a Web Services Definition Language (WSDL) file:

1. Open Stubby in your Notes client. In the default view in Stubby, click the Create New Doc button.
2. In the ‘WSDL file’ field, type the path to the WSDL file.
3. Click the Generate Stub Files button. This action generates proxy classes using information from the WSDL file and creates a Java Archive (JAR) file.
4. Select the ‘generated files’ tab to view the JAR, Java, and other files that were just created. The figure on the next page shows this tab after I generated the stubs for `CarsRPCSoap11.wsdl`.
5. Detach the JAR file from the ‘generated files’ tab to the local C: drive.

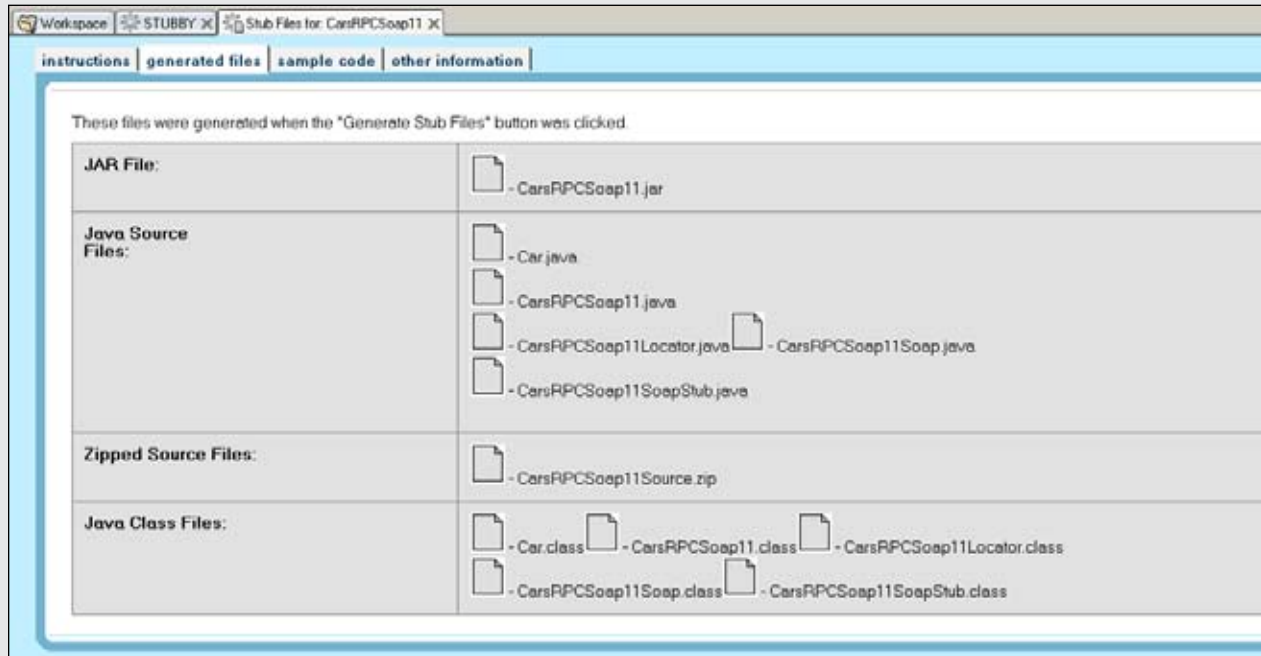
Now the JAR files are accessible for attaching to the Java agent that calls the Web service represented by this WSDL file.

* www.openntf.org/Projects/pmt.nsf/ProjectLookup/Stubby

Continues on next page

Continued from previous page

Also note that Stubby generates sample code (available on the 'sample code' tab) that includes everything necessary for calling the Web service. Copy and paste the sample Java code into the agent's programming pane in Domino Designer and use it as-is.



The 'generated files' tab in Stubby shows the JAR, JAVA, SOURCE.ZIP, and CLASS files just created.

```

1. import lotus.domino.*;
2. import dk.edbgruppen.schemas._2007._1.CarsTestWS.CarsRPCSoap11.*;
3. import lotus.domino.axis.client.Stub;
4. import lotus.domino.axis.message.SOAPHeaderElement;
5. import javax.xml.soap.SOAPElement;

6. public class JavaAgent extends AgentBase {
7. public void NotesMain() {
8. try {
9.     Session session = getSession();
10.     AgentContext agentContext = session.getAgentContext();
11.     Car myCar = new Car();

11.     // The Locator class knows how to access our Web service

```

Continues on next page

Figure 11 The PROXY.NET SOAP1.1 RPC Java agent code — excerpt

```

12.      CarsRPCSoap11Locator loc = new CarsRPCSoap11Locator();
13.      CarsRPCSoap11Soap service = loc.getCarsRPCSoap11Soap();

14.      myCar=service.getCar(1);
15.      System.out.println(myCar.getMake());

16.  } catch(Exception e) {
17.      e.printStackTrace();
18.  }
19.  }
20. }

```

Figure 11 (continued)

```

- <s:complexType name="Car">
- <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" form="unqualified" name="Model" type="s:string" />
  <s:element minOccurs="0" maxOccurs="1" form="unqualified" name="Make" type="s:string" />
  <s:element minOccurs="0" maxOccurs="1" form="unqualified" name="Color" type="s:string" />
</s:sequence>

```

Figure 12 A complex type Car object defined in a WSDL file

consumer application to Notes clients, I would not need any other Java files (JAR, etc.).

Lines 6 and 7 define the public class, which is a default when creating a new Java agent in Notes.

Lines 8 and 9 provide the agent context, which Domino Designer automatically generates when creating the Java agent.

Line 10 instantiates a new instance of the Car object defined in the .NET Web service (the relevant definition from the WSDL file is shown in **Figure 12**). Note that this Car object is a complex type name that the Web service defined with properties.

Lines 11 – 13 create a new instance of the Locator class from the JAR files, which are the glue that enables the Java code in the consumer to call the Web service. (These two lines of code are from the ‘sample code’ tab in Stubby, which provides usable sample code after generating stubs from a WSDL file.)

Line 14 calls the Web service with the getCar(int) method. This method is described in the test Web service site’s description depicted in **Figure 13**. Web service developers typically provide consumer developers with method descriptions like these, which identify the RPC style the Web service uses. **Line 14** in **Figure 11** also associates the response from the Web service call getCar(int) with the agent’s instance of the Car object (myCar).

Line 15 defines how to handle the response. Before coding the response handling, you can find out what’s required by examining the associated Java file. In this case, I looked at the Cars.java file that Stubby generated when creating the stubs. The file provides several properties and methods, including getMake (circled in **Figure 14**). The getMake property returns a string, (java.lang.String). Therefore, in **Figure 11**, **line 15** calls and prints this string to the terminal using the property System.out.println.

CarsRPCSoap11

Click [here](#) for a complete list of operations.

getCar

Test

The test form is only available for requests from the local machine.

SOAP 1.1

The following is a sample SOAP 1.1 request and response. The **placeholders** shown need to be replaced with actual values.

```

POST /CarsTestWS/CarsRPCSoap11.asmx HTTP/1.1
Host: vmarhwebserv
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://schemas.edbgruppen.dk/2007/1/CarsTestWS/CarsRPCSoap11/getCar"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <tns:getCar>
      <carId xsi:type="xsd:int">int</carId>
    </tns:getCar>
  </soap:Body>
</soap:Envelope>
        
```

Figure 13 The Web service description for the getCar method

instructions | generated files | sample code | other information

These files were generated when the "Generate Stub Files" button was clicked.

JAR File:	- CarsRPCSoap11.jar
Java Source Files:	- Car.java - CarsRPCSoap11.java - CarsRPCSoap11Locolor.java - CarsRPCSoap11SoapStub.java
Zipped Source Files:	- CarsRPCSoap11Source.zip
Java Class Files:	- Car.class - CarsRPCSoap11.class - CarsRPCSoap11Soap.class

Car.java - Notepad

```

package dk.edbgruppen.schemas._2007_1.CarsTestWS.CarsRPCSoap11;

public class Car implements java.io.Serializable {
    private java.lang.String model;
    private java.lang.String make;
    private java.lang.String color;

    public Car() {
    }

    public java.lang.String getModel() {
        return this.model;
    }

    public void setModel(java.lang.String model) {
        this.model = model;
    }

    public java.lang.String getMake() {
        return this.make;
    }

    public void setMake(java.lang.String make) {
        this.make = make;
    }

    public java.lang.String getColor() {
        return this.color;
    }

    public void setColor(java.lang.String color) {
        this.color = color;
    }

    private java.lang.Object __equalsCalc = null;
    public synchronized boolean equals(java.lang.Object obj) {
        if (!(obj instanceof Car)) return false;
        Car other = (Car) obj;
        if (obj == null) return false;
        if (this == obj) return true;
        if (__equalsCalc != null) {
            return (__equalsCalc == obj);
        }
        __equalsCalc = obj;
        boolean _equals;
        _equals = true &&
            ((this.model==null && other.getModel()==null) ||
            (this.model!=null &&
            this.model.equals(other.getModel())) &&
            ((this.make==null && other.getMake()==null) ||
            (this.make!=null &&
            this.make.equals(other.getMake()))) &&
        
```

Figure 14 Viewing the properties and methods of the Car.java file in Stubby

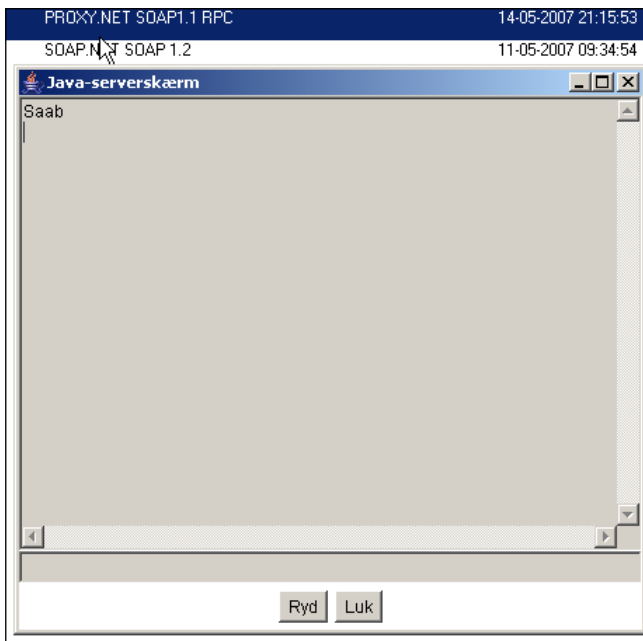


Figure 15 Response from running the PROXY.NET SOAP1.1 RPC agent in Domino Designer

Lines 16 – 20 catch any errors that might arise and close class and method brackets.

As you might expect, the results were fine for the `getApprovedCar` and `getApprovedCars` method calls, so again, I won't show them. However, as before, the `getCar` method returned different results for each routing style. **Figure 15** shows the PROXY.NET SOAP1.1 RPC agent results for the complex type response to the `getCar` method, and **Figure 16** shows the results of running the same method request in the PROXY.NET SOAP1.1 agent. The agent calling the RPC service successfully handled the response, but the agent calling the Document service failed to handle the response.

Why did the Document service response fail to display in Notes 7?

Figure 14 shows the Java Source Files section of the 'generated files' tab in Stubby, which had just generated the stubs for the WSDL file with the

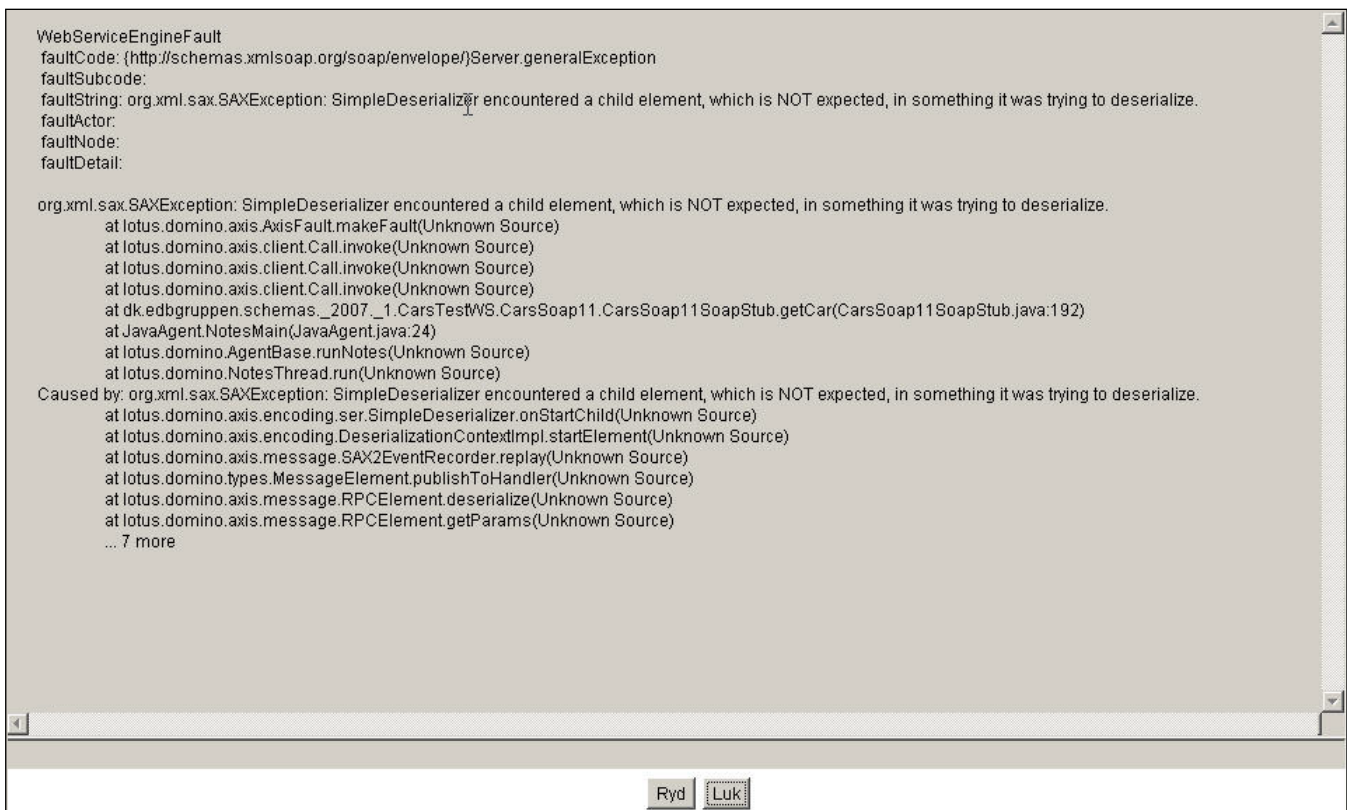


Figure 16 Response from running the PROXY.NET SOAP1.1 agent in Domino Designer

Document service. In the figure, Car.java is opened so its code is visible in the right-hand pane. Notice that the stubs actually use the `javax.xml.rpc.Service` package, which only supports RPC. In other words, Stubby seems to have interpreted the WSDL file as set to RPC, although it's set to use Document style. The end result is that there was a child element in the Document service response that the agent didn't expect — the Document style response was not formatted as the agent expected.

The disconnect in converting some data types correctly from a .NET Web service response to Java objects is not confined to Stubby. The problem seems to be in interpretation between Axis and a .NET Web service. I can't provide you a list of data types that work and don't work because it could differ depending on the IDE or conversion tool, the Axis version, and so on — too many factors! However, in my experience, a practice that most often provides good results is using the oldest or less complex data types in exchanges between a Java consumer and a .NET Web service. By this rule, for example, you might expect that arrays are sometimes structured differently because they are more complex.

I believe that the best solution lies with the .NET developer. That person should either write Web service methods that don't give responses likely to have translation problems or try to restructure the WSDL file generated from the .NET Web service so that parameters and responses are simpler and understandable for the WSDL2Java processes used by RAD, Eclipse, Stubby, and other tools.

If you can't arrange to match the data type to the type that the proxy class expects, don't give up. If you're knowledgeable about constructing Java classes, it might be possible to make slight modifications to the proxy class and still get it to work. I have developed Web service consumers that successfully mix automatically generated proxy classes with one or two manually modified classes.

The best way to go about this solution is to remove the problematic Java file from the JAR file on the local C: drive and then, from Stubby, import the Java file alone directly into the Notes Java agent, in the same way you import the JAR file (refer to the "Using

Stubby" sidebar). Then from Domino Designer, you can modify the Java file's code to accommodate what's required. For example, say Stubby's Java file interprets a parameter in a response as an object instead of an array, like this:

```
someFunction(java.lang.Object obj)
```

The consumer developer could try modifying the object to an array, like this:

```
someFunction(int[] array)
```

You can use this solution to solve problems in calls as well as in responses (e.g., the Web service expects an array in the calling method, but the tool generating the Java proxy converts the array to an object).

Consuming SOAP 1.2 services from SOAP 1.1 Java agents

Theoretically, testing the SOAP 1.1-compatible Notes consumer running against a Web service set to SOAP 1.2 should work because, as you saw earlier, SOAP 1.2 is designed to be backward compatible. To test this theory, I made two minor changes to the PROXY.NET SOAP1.1 RPC agent shown in **Figure 11**:

- I added the following line after **line 11** so that the calls point to the SOAP 1.2 RPC Web service on my test server.

```
URL url = new URL("http://vmarhwebsrv/
CarsTestWS/CarsRPC.asmx");
```

- I changed **line 12** by adding url as the method parameter:

```
CarsRPCSoap11Soap service = loc.
getCarsRPCSoap11Soap(url);
```

Because these changes are very simple, I didn't include the modified agent in the download files.

This test had disappointing results. As soon as the altered agent started calling the Web service methods,

Routing styles

If you develop calls to Web services from any Java Web service consumer, you'll probably find it easier to consume Web services that use the RPC routing type because Axis, which is present in Domino, Rational Application Developer (RAD), and many developer tools, handles many of the RPC elements for you.

Below is some C# (C Sharp) Web service code for a simple Hello World Web service written with VS .NET. The SoapRpcService attribute in **line 7** specifies the RPC routing style for this Web service.

```

1. using System;
2. using System.Web;
3. using System.Web.Services;
4. using System.Web.Services.Protocols;

5. [WebService(Namespace = "http://tempuri.org/")]
6. [WebServiceBinding(ConformsTo = WsiProfiles.None)]
7. [SoapRpcService(RoutingStyle = SoapServiceRoutingStyle.SoapAction)]
8. public class Service : System.Web.Services.WebService

9. {
10.     public Service () {

11.         //Uncomment the following line if using designed components
12.         //InitializeComponent();
13.     }

14.     [WebMethod]
15.     public string HelloWorld() {
16.         return "Hello World";
17.     }
18. }
```

Line 8 indicates the beginning of the Web service, which is public in this example. **Lines 10 – 17** indicate the methods that the Web service contains.

Some consumer programmers complain that the RPC routing style couples the consumer with the provider too tightly. The WSDL file provides more information about the structure and tells the consumer more about using the data than the standard Document service does. For example, say an RPC service method tells you to use an integer as the parameter and a Document service method tells you to specify the same parameter as a type that's defined elsewhere. If you want to change the parameter to take two values, you have to change every call your code makes to the RPC service method, but you only have to change the object passed to the Document service method, not the calls. In other words, the Web service consumer developer has less flexibility when the Web service method uses RPC.

The only thing the provider needs to do to convert this source code to Document service routing is to change the embedded word Rpc in **line 7** to Document, as follows:

```
SoapDocumentService(RoutingStyle = SoapServiceRoutingStyle.SoapAction)
```

I got errors for the responses to every method, such as “Server did not recognize the value of HTTP Header SOAPAction.” The errors meant that the request did not match what the Web service was expecting. Resolving errors like these requires looking in the Java files to find the structure expected by the Web service and using tcpmon to see the call the consumer sent to the Web service before the failures occurred. The goal is to create a request that suits the Web service, which probably involves altering the Java files involved, just as in the previous section.

An alternative is to try to get the Web service provider to change the SOAP headers (see the sidebar “How to change a SOAP 1.2 service to a SOAP 1.1 service”). If you don’t own or can’t influence the Web service, then this solution is not practical for you.

Your best option for a Notes 6 or 7 consumer of SOAP 1.2 Web services may be to create a custom .NET control with a COM wrapper. A COM wrapper is capable of consuming a Web service that uses SOAP 1.2. For more information on this technique, refer to the sidebar “Another design approach: Create a custom .NET control with a COM wrapper to use with LotusScript or Java code.” Note that you don’t have to use Java with this technique — for example, I have written such objects as .NET code (Visual Basic/C#) and used them from LotusScript agents.

Summary of advice for Notes 6 and 7 developers

Consuming with LotusScript and the MS SOAP object

If you are creating an MS SOAP and LotusScript Web service client, then it’s possible to consume a .NET Web service that uses SOAP 1.1 or SOAP 1.2.

However, you must check the routing style the Web service uses for responses. You must create the response object based on the expected response, given the routing style.

Regardless of current success, though, be aware that the MS SOAP object is deprecated. You will need to reprogram the consumer code when this object fails to work. The only alternative for maintaining a LotusScript solution in Notes 6 and 7 is to create a custom .NET control with a COM wrapper.

Consuming with Java

If the Web service uses SOAP 1.2, Stubby won’t be able to generate stubs at all. Eclipse and RAD can generate proxy classes for SOAP 1.2 Web services, but the proxy classes wouldn’t work in Notes 6 or 7 because Java in these releases is based on Axis.

If the Web service uses SOAP 1.1, generating stubs with Stubby and running Java agents is an easy way of calling a .NET Web service from Notes. However, Stubby assumes that the .NET Web service you are calling is RPC-enabled. If the Web service uses Document routing, calls and responses with simple data types are likely to succeed, but the calls or responses using complex data types (regardless if generated in .NET, RAD, or Eclipse) might generate errors in Notes 6 or 7. In such cases, you must manually alter the relevant classes by adding the correct child elements to make the call or handle the response.

If you are not an expert Java programmer, I don’t recommend trying to consume SOAP 1.2 Web services or non-RPC services with Java agents in Notes 6 or 7, as the technique for correcting problems is complicated and error-prone. If you provide or have influence over a Web service that uses SOAP 1.2, to consume it reliably in Notes 6 or 7 with Java, you would have to change its headers or ask the provider to change the service to use SOAP 1.1. Few developers have the means to do so, but for those of you who can either change the Web service directly or influence the provider to change it, you can find out exactly what to change in the sidebar “How to change a SOAP 1.2 service to a SOAP 1.1 service.” It explains the subtle and often elusive points that must be addressed in order for the change to be successful.

How to change a SOAP 1.2 service to a SOAP 1.1 service

For those of you who have control or influence over the Web service you want to consume with Notes 6 or 7, this sidebar provides a summary of the steps to change a Web service from SOAP 1.2 to SOAP 1.1. To follow along as an exercise, you can start with the Web service that I used to generate the `Org_Cars.wsdl` and `Org_CarsRPC.wsdl` files, both of which are set to SOAP 1.2. Here are the basic steps:

1. In Visual Studio (VS) .NET, alter the `web.config` file to use SOAP 1.1 for generating responses.
2. Generate the new Web Services Definition (WSDL) file.
3. In a text editor, remove the SOAP 1.2 reference in the WSDL file to conform to Stubby requirements.

When the WSDL file is ready, you can use the edited WSDL file in Stubby to generate the SOAP 1.1-compatible Java-based consumer code in the normal manner. Don't forget to test the client code to make sure it can consume the Web service.

Steps 1 and 3 are the tricky steps, so let's take a closer look at changing the SOAP version in the `web.config` file and editing the header in the WSDL file.

Changing the SOAP version in the `web.config` file

Follow these steps to alter the `web.config` file:

1. Open the `web.config` file in VS .NET.
2. Change the version of SOAP from `HTTPSoap12` to `HTTPSoap` by going to the `<protocols>` section and using the 'remove name' and 'add name' properties, as shown here:

```
<protocols>
  <remove name="HttpGet"/>
  <remove name="HttpPost"/>
  <remove name="HttpSoap12"/>
  <add name="HttpSoap"/>
</protocols>
```

Changing the SOAP protocol version in the `web.config` file

This change ensures that the Web service only provides binding for SOAP 1.1.

3. Regenerate the WSDL file.

Changing the `web.config` file in VS .NET does most of the job of removing SOAP 1.2 references, but the resulting WSDL file doesn't conform to the standard that Stubby expects because VS .NET leaves a reference to the SOAP 1.2 namespace in the header. You have to manually remove this reference.

Removing the SOAP 1.2 reference in the WSDL file's header

Before using the WSDL file in Stubby, use a text editor to remove the reference to SOAP 1.2 in the header, which refers to the SOAP 1.2 namespace. Note that Stubby generates this line in the WSDL file

Continues on next page

Continued from previous page

even if you changed the web.config file in VS .NET. As discussed earlier, the header already has a line referring to xmlns:soap (the SOAP 1.1 assumption for backward compatibility). The image below highlights this line in the header of a WSDL file. Simply delete the line with any text editor.

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tns="http://schemas.edbgruppen.dk/2007/1/CarstestWS/CarsRPC" xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
targetNamespace="http://schemas.edbgruppen.dk/2007/1/CarstestWS/CarsRPC"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <s:schema elementFormDefault="qualified"
targetNamespace="http://schemas.edbgruppen.dk/2007/1/CarstestWS/CarsRPC">
      <s:import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
      <s:import namespace="http://schemas.xmlsoap.org/wsdl/" />
      <s:complexType name="Car">
```

The SOAP 1.2 namespace line in the header of the WSDL file

After you remove the SOAP 1.2 reference in the header, the WSDL file for the RPC Web service looks similar to the CarsRPCSoap11.wsdl file that I provide with the download files.

You can now run this WSDL file through Stubby again to generate new stubs.

Tip!

To change a Web service from one routing style to another, the provider changes the attribute “SOAP call for RoutingStyle” set in the Web service code.

Using Notes 8 as a .NET Web service consumer

By now, you know that consuming Web services in Notes 6 or 7 is possible in many circumstances. It’s easier in some situations than others. There are many issues the developer has to take into consideration in order to come up with a successful consumer design. The good news is that with Notes 8, these concerns are over. Notes/Domino 8 contains all of the functionality necessary for consuming .NET Web services well. I don’t have room in this article for complete coverage

of the Domino Designer 8 Web service features that support providers and consumers, but I can give you a couple of consumer agent examples that point out some of the features that make LotusScript and Java Notes 8 agents successful consumers of .NET Web services.

Tip!

Domino Designer 8 supports the Web Services Interoperability (WS-I) standards, which aren’t supported by RPC services. This topic is beyond the scope of this article, but good for Web service developers to know about. For more information, visit the WS-I Web site⁸ or read the IBM developerWorks article on WS-I Compliant Web services in Domino 8.⁹

⁸ www.ws-i.org

⁹ www.ibm.com/developerworks/lotus/library/domino8-WS-I

The examples in this section use the WSDL file named Cars.wsdl; it's the WSDL file for the Document routing flavor of the basic .NET Web service that uses SOAP 1.2 for generating responses. These are the settings that gave poor results in the Notes 6 or 7 examples earlier.

You might be wondering why we aren't testing consuming SOAP 1.2 Web services in Notes/Domino 8. The answer is that Notes/Domino 8 uses the Axis2 framework to generate Java or LotusScript proxy classes, and Axis2 supports responses generated with either SOAP 1.1 or SOAP 1.2. I haven't found any errors so far, so I have none to show you here.

I ran the Web services for these tests in the same way as for the Notes 6 and 7 tests.

Consuming with LotusScript

Instead of using an MS SOAP object for calling Web services with LotusScript, I used LotusScript proxy classes. Domino Designer 8 automatically generates the LotusScript (or Java, if preferred) proxies for an imported WSDL file in a special script library.

Generating this library was easy. In Domino Designer 8, I created a new LotusScript script library named PROXYClasses. I opened the library's Declarations work pane (see **Figure 17**), clicked the WSDL button at the bottom of the screen, and chose Import WSDL from the available actions to import Cars.wsdl from my local C: drive. Domino Designer 8 automatically generated LotusScript classes from the

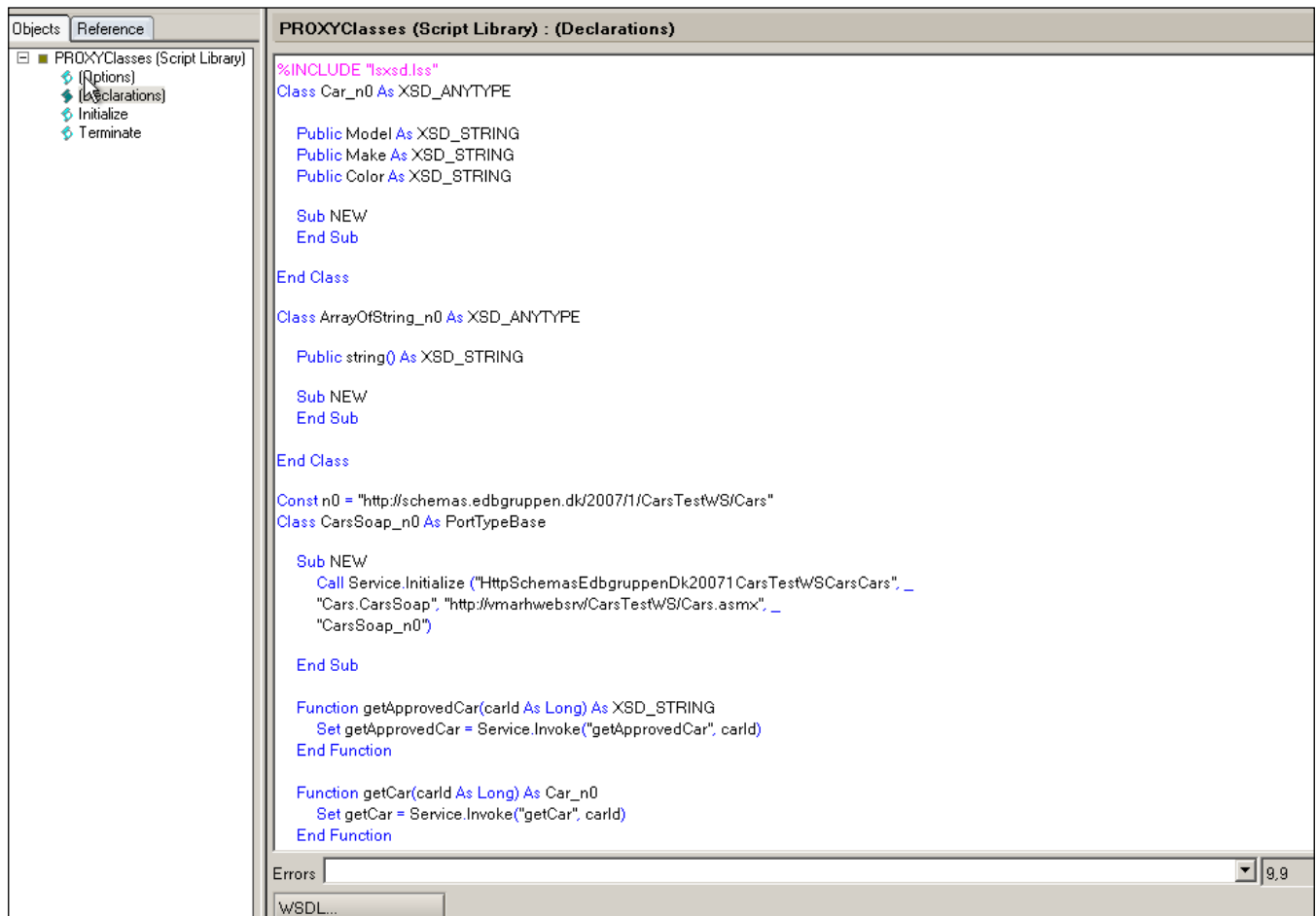


Figure 17 Domino Designer 8 can automatically convert an imported WSDL file into proxy classes in the declarations of a LotusScript script library.

imported WSDL file and displayed them in the library's Declarations pane. Note that the script library contains only back-end classes (after all, the consumer developer doesn't need the Web service UI to code requests). Therefore, the script library can work with agents in the Notes 8 client or Domino 8 server.

Note that Domino Designer 8 automatically includes the `lsxsd.lss` file in all of its LotusScript proxy library declarations. This file contains predefined LotusScript classes to use with Web services. Note that the `Model` variable has a special data type: `XSD_STRING`. The `lsxsd.lss` file defines `XSD_STRING` as having the methods `GetValueAsString` and `SetValueFromString(String)`, which Domino uses to get and set variables within calls to the Web service as the correct data types for calling or for interpreting Web service responses regardless of routing style. For further information about the data type mapping Domino Designer uses between the WSDL file and Java or LotusScript, look in the Domino Designer 8 Help files under "Java and LotusScript mappings."

When the script library was ready, I created a new LotusScript agent named ".NET LSAgent" with the code shown in **Figure 18**. This code does the equivalent of the code in **Figure 11** — i.e., the response should show the make of a car. By looking at the LotusScript library, I determined the exact names of the Car class properties (`Model`, `Make`, and `Color`) that Domino Designer generated from the WSDL file. Notice that the script library shown in **Figure 17**,

displays property names with the suffix `_n0`. I'm not sure yet why Notes choose to suffix the names with `_n0` — perhaps it's because they need to differ from the names in the WSDL file.

Line 2 of **Figure 18** declares the endpoint `myMake`, which must be a Variant, due to the fact that the Web service returns the make as `XSD_STRING` (see **Figure 17**).

Line 3 declares `myCar` as a `New Car_n0`, the response that the Web service returns to the `getCar` method.

Line 4 declares the SOAP call to the Web service. It declares the variable `mySoap`, of type `CarsSoap_n0`, which is a class defined in the proxy classes.

Line 5 calls the Web service method `getCar` with the integer 1, which asks the Web service to return the first car in its list of cars (objects). The response is returned to the variable object `myCar`.

Line 6 gets the property `Make` from the object `myCar`. As you can see in the declarations in **Figure 17**, the `Make` property has a data type of `XSD_STRING` and sets the variable `myMake` with the result.

`XSD_STRING` is a special Web service data type, hence it is necessary to use the method `GetValueAsString` to convert the `XSD_STRING` to a normal LotusScript string. **Line 7** uses the `GetValueAsString` method to return a string from the `myMake` variable defined as `XSD_STRING`.

```

1. Sub Initialize
2.   Dim myMake As Variant
3.   Dim myCar As New Car_n0()
4.   Dim mySoap As New CarsSoap_n0()

5.   Set myCar=mySoap.getCar(1)
6.   Set myMake= myCar.Make

7.   MsgBox myMake.GetValueAsString()
8. End Sub

```

Figure 18 The LotusScript code for the .NET LSAgent agent

Msgbox displays the string (converted from XSD_STRING) in a message box to the user.

When I ran the LotusScript agent from Domino Designer 8, the call caused the Web service to correctly return the make of the car, which the agent displayed in a message box.

Consuming with Java proxy classes

I repeated the same process for creating the Java consumer agent in Domino Designer 8. I created the PROXYClassesJava script library and imported the Cars.wsdl from my desktop to populate the library's declarations (see **Figure 19**). These proxy classes are based on the Document style WSDL file, and the outcome is similar to that of the Stubby/Notes 7 example, except that the result is pure Notes; i.e., you

don't use any other tools than Notes, and you can manipulate the objects returned directly in Notes.

Then I created a new Java agent, called ".NET JavaAgent," with the code shown in **Figure 20**.

This agent code is very similar to the Java agent for Notes 7, with these differences:

- **Line 1** imports the lotus.domino package, which allows the code to call Domino objects from a Java program.
- **Line 2** includes the proxy classes that Domino Designer 8 generated when importing the local WSDL file.
- **Line 3** is required for the new URL referenced in **line 12**, because the Java package java.net defines the type "URL."

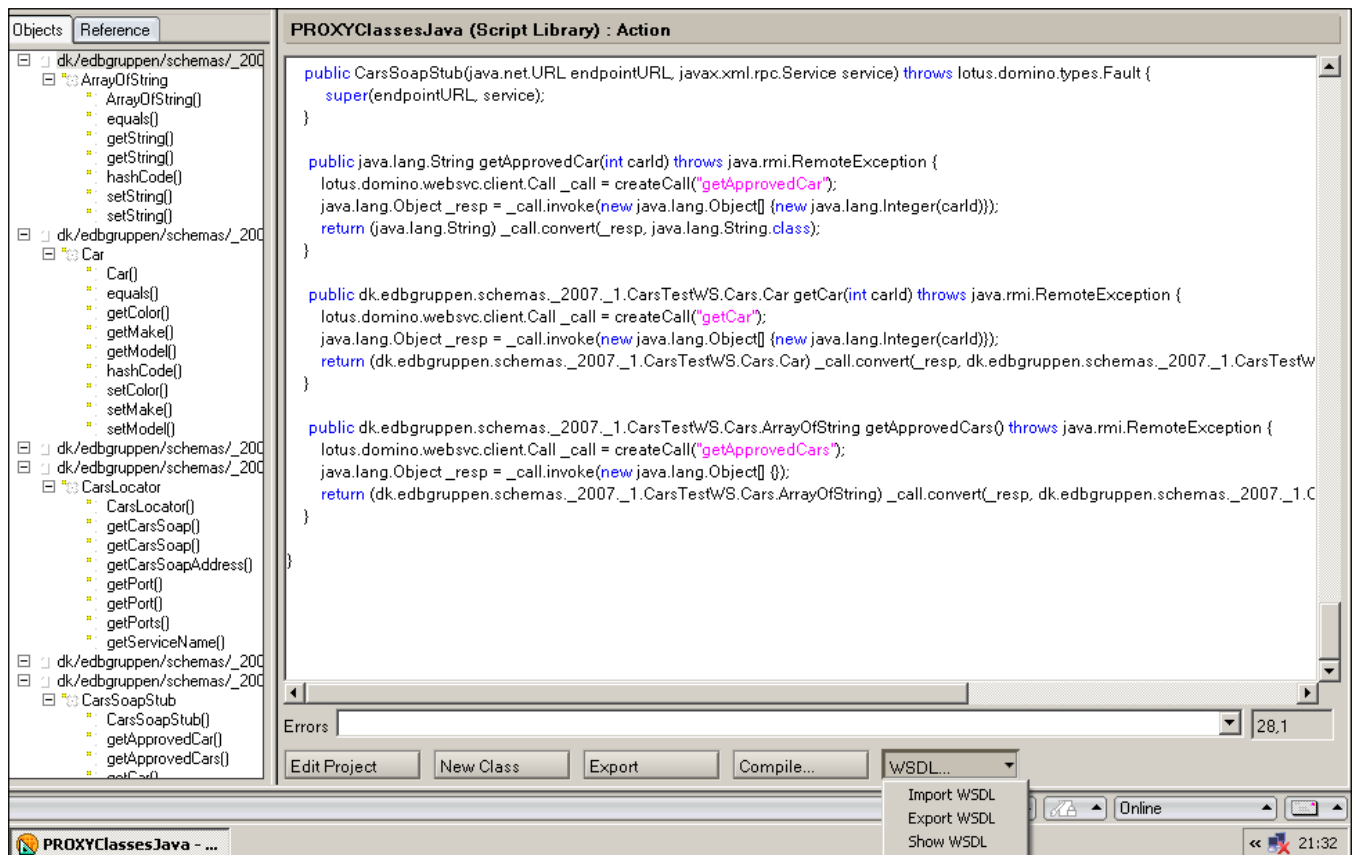


Figure 19 Domino Designer 8 can automatically convert an imported WSDL file into classes in the declarations of a Java script library.

- The Locator class has a different name in the Notes/Domino 8 script library — it's CarsLocator instead of CarsRPCSoap11Locator. This is because Stubby generated the Notes 7 example, while Notes 8 natively generated the Notes 8 code.

When I ran this Java agent from Domino Designer 8, I got the expected system output message “Saab” in my Java console.

Notice that a Notes 8 consumer developer does not need to know whether a Web service is RPC-enabled or Document-enabled. Notes 8 uses Axis2, so it has no problems handling SOAP 1.2 Web services and Document services. The consumer developer never has to tinker with the proxy classes Domino Designer 8 generates from a WSDL file.

Contrast this ease with developing the same consumer agent for Notes 6 or 7, where I had to manipulate certain classes from stubs so that they could handle the Document service.

Tip!

If you don't know the properties and methods that you can call from the Java or LotusScript agent, examine the proxy classes in the script library you created from the WSDL file. These classes usually specify what you need to know about the methods and properties.

```

1. import lotus.domino.*;
2. import dk.edbgruppen.schemas._2007._1.CarsTestWS.Cars.*;
3. import java.net.*;

4. public class JavaAgent extends AgentBase {
5.     public void NotesMain() {
6.         try {
7.             Session session = getSession();
8.             AgentContext agentContext = session.getAgentContext();
9.             Car myCar = new Car();

10.             // The Locator class knows how to access our Web service
11.             CarsLocator loc = new CarsLocator();
12.             URL url = new
13.             URL("http://vmarhwebsrv/CarsTestWS/Cars.asmx");

14.             CarsSoap service = loc.getCarsSoap(url);
15.             myCar=service.getCar(1);
16.             System.out.println(myCar.getMake());

17.         } catch(Exception e) {
18.             e.printStackTrace();
19.         }
20.     }
21. }

```

Figure 20 The Java code for the .NET JavaAgent agent

Consuming RPC and/or SOAP 1.1 Web services

I repeated these exercises to develop LotusScript and Java agents that call the RPC-enabled Web services and the SOAP 1.1-enabled Web services. I didn't have to modify either the WSDL files before generating proxies or the Domino Designer-generated proxy classes used by the agents.

The results

All of the agent calls were successful. In fact, in my daily work so far, I haven't come across any problems with calling Web services from Notes 8, other than the caveats that Domino Designer 8 lists in its Help files under Web services → Unsupported constructs. For example:

- The anyAttribute element extends the schema with attributes not specified in the schema.
- Likewise, the data type 'list' is not supported from LotusScript mappings to the WSDL file.

Developer summary on Notes 8 consumers of .NET Web services

In Notes 8, your LotusScript and Java consumer agents should be able to call any .NET Web service and interpret its responses correctly regardless of whether the Web service's settings are for RPC or Document routing style, SOAP 1.1 or SOAP 1.2. There is no need to craft code that can handle responses with complex data types — the proxy methods Domino generates correctly turn all responses into objects that the agent code can use directly.

Note that there still is no easy way of generating Axis2 proxy classes that will work with Notes/Domino releases earlier than 8. To my knowledge, you can't take the special Web-enabled script libraries generated by Domino Designer 8 and use them in agents that work in earlier Notes clients.

Conclusion

In this article, you explored some subtle but crucial issues involved in developing a Notes consumer of .NET Web services. In particular, I showed you the difficulties that can arise in Note 6 or 7 when using the two most popular ways of consuming Web services: MS SOAP objects with LotusScript, or Java with stubs generated with Stubby. The difficulties arise from incompatibilities in SOAP protocols, Axis versions, and routing styles. You learned how to identify potential issues by looking at settings visible in the WSDL files, and you discovered what you can and can't do to get around the issues. You also observed that the easiest .NET Web service to consume from Notes 6 or 7 uses SOAP 1.1 and the RPC routing style.

In addition, you discovered how to tune a Web service to be more friendly with Notes 6 and 7 when you have enough control or influence over the provider. You also learned about the tcpmon tool, a great utility that helps developers troubleshoot problems with Web service calls.

You now know the design practices that make Notes 6 and 7 consumers of .NET services more successful. For example, if you still choose to design consumers using the MS SOAP object (despite looming obsolescence), you know that the best way to do so is to have the consumer client connect to a servlet on the Web server, which then uses the MS SOAP object to communicate with the Web service. That way, you only have to maintain the MS SOAP object connection on the server, instead of at each of the consumer clients. You also saw that Domino Designer and Notes 8 provide the most problem-free, fastest, and easiest way to implement Notes as a Web service consumer.

Further reading

For some helpful resources, please see the next page.

Resources

SOAP

For information on creating and consuming Web services with the MS SOAP Toolkit, I recommend the following articles:

Michael Thomas Mohen, “Creating and Using Web Services with Domino” (*THE VIEW*, May/June 2003).

Michael Thomas Mohen, “Creating and testing Web services with the new design element in Domino 7” (*THE VIEW*, March/April 2006).

For information on Microsoft support for the SOAP Toolkit 3.0, visit this Web site:
<http://support.microsoft.com/kb/811215/>

Java proxies

For details on using Java proxies to consume Web services, I recommend these articles:

Jochen Finkbeiner, “Consume Web services in Lotus Notes applications, Part 1: Preparing the Java proxy” (*THE VIEW*, July/August 2006).

Jochen Finkbeiner, “Consume Web services in Lotus Notes applications, Part 2: Deployment” (*THE VIEW*, September/October 2006).

Stubby, tcpmon, and WS-I

For information or to download Stubby, refer to this Web site:
www.openntf.org/Projects/pmt.nsf/ProjectLookup/Stubby

You can find more information on tcpmon at this Web page:
<https://tcpmon.dev.java.net/>

The WS-I Web site:
www.ws-i.org/

For information on WS-I Compliant Web services in Domino 8, read the article
“Engineering WS-I compliant Web services for IBM Lotus Domino V8” (IBM developerWorks):
www.ibm.com/developerworks/lotus/library/domino8-WS-I/