Wolfram *Mathematica*® Tutorial Collection

WEB SERVICES USER GUIDE

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Content authored by:
Chris Williamson

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

Contents

Introduction to Web Services

The World Wide Web is increasingly being used for communication between applications. The programmatic interfaces made available over the web for application-to-application communication are often referred to as web services. There are many types of applications that can be considered web services but interoperability between applications is enhanced most by the use of familiar technologies such as XML and HTTP. These technologies allow applications using differing languages and platforms to interface in a familiar way.

The web services client for *Mathematica* allows users to call operations that are based remotely on other platforms or languages that are not immediately accessible to *Mathematica*. This opens up a whole new realm of functionality and data to users of *Mathematica*.

There are several keys technologies that enable *Mathematica* to interact with web services. Following is an explanation of each technology.

XML and HTTP

XML and HTTP are the foundation for calling web services with *Web Services Link*. A user interacts with the web service interfaces by sending XML messages over HTTP. XML and HTTP are useful for creating and sending messages because they are flexible and widely supported on many platforms and languages. This interoperability allows applications to communicate using differing languages and platforms using these common technologies. *Web Services Link* supports XML 1.0 and HTTP 1.1. The XML functionality is supplied by the XML capabilities built into *Mathematica.* The HTTP functionality is supplied by the Apache Commons HTTP client. The user may find knowledge of XML and HTTP to be helpful, however, it is not required. For more information about HTTP, please see http://www.w3.org/Protocols/. For more information about XML, please see http://www.w3.org/XML/.

SOAP

Because XML and HTTP are so flexible, a web service message may be represented and communicated in many different ways. Therefore it is important to have standards that are common among all platforms and languages that use a web service. Simple Object Access Protocol (SOAP) is one standard that formally sets the conventions governing the format and processing rules of XML messages and how they are used with HTTP. *Web Services Link* supports SOAP 1.1. *Mathematica* users are not required to know any specifics of SOAP. For more information, please see http://www.w3.org/TR/SOAP/.

WSDL

Another useful technology that is associated with web services is Web Service Description Language (WSDL). WSDL formally sets the format for describing a web service using XML. By providing a Uniform Resource Identifier (URI) to a WSDL, applications can discover what operations a particular web service provides and how an operation’s messages look. With this information *Web Services Link* can generate SOAP messages using input from *Mathematica* functions. A web services operation may be used without a WSDL, however, the WSDL is key for promoting a web service. *Web Services Link* supports WSDL 1.1. *Mathematica* users are not required to know any specifics of WSDL. For more information, please see http://www.w3.org/TR/wsdl.

XML Schema

Data types are a very important component of web services because different data types may be used to compose SOAP messages. WSDL uses XML Schema to describe data types used in a web service. *Web Services Link* uses the XML Schema package to dynamically bind *Mathematica* expressions to XML. This package validates data against a schema to make sure it correctly matches the definition for a data type. *Web Services Link* supports XML Schema 1.0. *Mathematica* users are not required to know any specifics of XML Schema. For more information, please see http://www.w3.org/XML/Schema.

WS-I Basic Profiles

While SOAP and WSDL are great starts for implementing web services, there are compatibility issues that result from these specifications. WS-I Basic Profiles address all compatibility issues by specifying implementation guidelines on how core web services specifications should be used together to develop interoperable web services. This specification gives guidelines for WSDL, SOAP, and Universal Description, Discovery and Integration (UDDI) implementations. The focus of this specification is easier integration among varying implementations of web services and their clients. The client uses these guidelines for the implementation. A web service that follows these guidelines is more likely to work correctly with the client. *Web Services Link* supports WS-I Profiles 1.1. *Mathematica* users are not required to know any specifics of Basic Profiles. For more information, please see http://www.ws-i.org/.

Getting Started with Web Services

Loading the Package

Web Services Link is a *Mathematica* add-on application. Before any functions from the package can be used, it must be loaded as follows.

```
In[1]:= << WebServices`
```

Installing a Web Service

`InstallService` will install the web service operations defined by a supplied WSDL as *Mathematica* functions. The functions created are returned in a list.

```
In[2]:= InstallService[
         "http://localhost:8080/webMathematica/Examples/WebServices/Hello.m?wsdl"]
Out[2]= {HelloWorld}
```

Naming a Web Service

The functions created by `InstallService` are placed in a context based on the service name and port name specified by the WSDL. A user can change this context by supplying a valid *Mathematica* context to `InstallService`.

```
In[3]:= InstallService[
         "http://localhost:8080/webMathematica/Examples/WebServices/Echo.m?wsdl",
         "EchoService`"]
Out[3]= {EchoBase64Binary, EchoBoolean, EchoDate, EchoDateTime, EchoDatum, EchoDatumArray,
         EchoExpression, EchoInteger, EchoIntegerArray, EchoMathML, EchoReal, EchoString, EchoTime}
```

```
In[4]:= Context[EchoString]
Out[4]= EchoService`
```

By default the context of a web service is added to the `$ContextPath`. This allows users to call the functions without having to specify the context. However, there is a danger that the functions installed by `InstallService` will collide with other functions already defined in *Mathemat-*

ica on `$ContextPath`. If a user does not want the web service context added to `$ContextPath`, then the user should set `AllowShortContext -> False`. If this option is set to `False`, then the entire context will need to be specified in order to call a function.

```
In[5]:= InstallService[
         "http://localhost:8080/webMathematica/Examples/WebServices/Plot.m?wsdl",
         AllowShortContext → False]
Out[5]= {PlotServiceService`PlotServicePort`PlotExpression,
         PlotServiceService`PlotServicePort`PlotString}
```

Inspecting Web Service Functions

`$InstalledServices` may be used to determine which web service functions are installed. This is a list of all the web service functions installed.

```
In[6]:= $InstalledServices
Out[6]= {EchoBase64Binary, EchoBoolean, EchoDate, EchoDateTime, EchoDatum, EchoDatumArray,
         EchoExpression, EchoInteger, EchoIntegerArray, EchoMathML, EchoReal, EchoString,
         EchoTime, HelloWorld, PlotServiceService`PlotServicePort`PlotExpression,
         PlotServiceService`PlotServicePort`PlotString}
```

The usage message for each function may be inspected to discover how the function works. `? SymbolName` may be used to get the usage message for a symbol. For instance, `? HelloWorld` retrieves the usage message for `HelloWorld`. The usage message for a web service function is populated with the documentation provided by the WSDL.

```
In[7]:= ? HelloWorld
```

```
String HelloWorld[]

Simple HelloWorld web service example.
```

Each usage message contains a function signature that may be used to call the function. For instance, `HelloWorld` does not require any parameters and returns a string.

Executing a Web Service Function

The usage message should be a starting point for determining what to input as parameters and what to expect as a result. Once this is determined, the function may be executed. A web service function may be executed just like any other *Mathematica* function.

```
In[8]:= HelloWorld[]
Out[8]= Hello! The current date is {2007, 3, 29, 14, 15, 17.5561239}
```

Basic Examples of Web Services

DataTypes: Echo

Often a web service requires using data other than a simple string. This data could be something simple like an integer or a real. Often the data is more complex and is a combination of simple data types. This example demonstrates how to use different data types.

```
In[9]:=  << WebServices`

In[10]:= InstallService[
          "http://localhost:8080/webMathematica/Examples/WebServices/Echo.m?wsdl"]
Out[10]= {EchoBase64Binary, EchoBoolean, EchoDate, EchoDateTime, EchoDatum, EchoDatumArray,
          EchoExpression, EchoInteger, EchoIntegerArray, EchoMathML, EchoReal, EchoString, EchoTime}
```

For each of the examples, the usage message may be used to determine the data types used by a web service.

Simple Data Types

Simple data types are the easiest data types to use in *Web Services Link*. *Mathematica* users should be familiar with the symbols `String`, `Integer`, `Real`, `True`, and `False`. Each of these may be used with a web service function.

Here is an example of using `True` with a web service.

```
In[11]:= ? EchoBoolean

         (True | False) EchoBoolean[bool:(True | False)]

         Echos True or False

In[12]:= EchoBoolean[True]
Out[12]= True
```

Here is an example of using `String` with a web service.

```
In[13]:= ? EchoString

    String EchoString[str_String]

    Echos a String.

In[14]:= EchoString["Hello."]
Out[14]= Hello.
```

Here is an example of using `Integer` with a web service.

```
In[15]:= ? EchoInteger

    Integer EchoInteger[int_Integer]

    Echos an Integer.

In[16]:= EchoInteger[13]
Out[16]= 13
```

Here is an example of using `Real` with a web service.

```
In[17]:= ? EchoReal

    Real EchoReal[real_Real]

    Echos a Real.

In[18]:= EchoReal[2.3333]
Out[18]= 2.3333
```

Nonstandard Simple Data Types

Web Services Link provides some data types that may be mapped to the XML Schema of a web service. `SchemaDate`, `SchemaTime`, `SchemaDateTime`, `SchemaExpr`, `SchemaMathML`, and `SchemaBase64Binary` are each simple data types that may be used with a web service.

Here is an example of using `SchemaDate` with a web service.

```
In[19]:= ? EchoDate
```

```
SchemaDate EchoDate[d_SchemaDate]

Echos a date.
```

```
In[20]:= ? SchemaDate
```

```
SchemaDate[year_Integer, month_Integer,
    day_Integer] represents a date that may be used with XMLSchema.
```

```
In[21]:= EchoDate[SchemaDate[2005, 7, 4]]
Out[21]= SchemaDate[2005, 7, 4]
```

Here is an example of using `SchemaTime` with a web service.

```
In[22]:= ? EchoTime
```

```
SchemaTime EchoTime[t_SchemaTime]

Echos a time.
```

```
In[23]:= ? SchemaTime
```

```
SchemaDateTime[hour_Integer, minute_Integer,
    second_Real] represents a time that may be used with XMLSchema.
```

```
In[24]:= EchoTime[SchemaTime[5, 0, 0.]]
Out[24]= SchemaTime[5, 0, 0]
```

Here is an example of using `SchemaDateTime` with a web service.

```
In[25]:= ? EchoDateTime
```

```
SchemaDateTime EchoDateTime[dt_SchemaDateTime]

Echos a date time.
```

```
In[26]:= ? SchemaDateTime
```

```
SchemaDateTime[year_Integer, month_Integer, day_Integer, hour_Integer,
    minute_Integer, second_Real] represents a dateTime that may be used with XMLSchema.
```

```
In[27]:= EchoDateTime[SchemaDateTime @@ Date[]]
Out[27]= SchemaDateTime[2007, 3, 29, 14, 19, 36.9501]
```

Examples of `SchemaExpr`, `SchemaMathML`, and `SchemaBase64Binary` follow in the next section.

Compound Data Types

Web Services Link generates symbols that are used to build compound data types. Compound data types are *Mathematica* expressions that contain instances of simple data types and/or compound data types. Compound data types are created in *Mathematica* using a symbol representing the data type as the head of a *Mathematica* expression. The members of the data type are represented using rules in the body of the expression. The left-hand side of the rule specifies the name of a member of the data type. The right-hand side specifies the value of the member. The value may be a simple or compound value.

Here is an example of a web service that uses a compound data type.

```
In[28]:= ? EchoDatum
```

```
Datum EchoDatum[d_Datum]

Echos a datum.
```

```
In[29]:= ? Datum
```

```
TYPE PROPERTIES
Name      Datum
Namespace http://localhost:8080/webMathematica/Examples/WebServices/Echo.m
Global    True
Array     False

TYPE ELEMENTS
Name  Type    MinOccurs MaxOccurs Default Fixed Symbol
value Integer 1         1         Null    Null  value$488
text  String  1         1         Null    Null  text$493
```

`NewSchemaInstance` may be used to generate an instance of `Datum`.

```
In[30]:= datum = NewSchemaInstance[Datum, {"value" → 5 !, "text" → "Hello!"}]
Out[30]= Datum[value → 120, text → Hello!]

In[31]:= EchoDatum[datum]
Out[31]= Datum[value → 120, text → Hello!]
```

Here is an example of how to retrieve data from a compound data type.

```
In[32]:= datum["value"]
Out[32]= 120
```

`ReplaceSchemaInstanceValue` may be used to set a value for a compound data type.

```
In[33]:= ReplaceSchemaInstanceValue[datum, "value" → 6!]
Out[33]= Datum[value → 720, text → Hello!]
```

Arrays

Web Services Link supports arrays using `List`. To use an array, you need only to use a *Mathematica* `List` consisting entirely of the data type defined by the array.

Here is an example of a web service that uses an array of simple data types.

```
In[34]:= ? EchoIntegerArray
```

```
{___Integer} EchoIntegerArray[array:{___Integer}]\n\nEchos an integer array.
```

```
In[35]:= EchoIntegerArray[{1, 2, 3}]
Out[35]= {1, 2, 3}
```

Here is an example of a web service that uses an array of compound data types.

```
In[36]:= ? EchoDatumArray
```

```
{___Datum} EchoDatumArray[array:{___Datum}]

Echos a datum array
```

```
In[37]:= datum = NewSchemaInstance[Datum, {"value" → 5!, "text" → "Hello!"}]
Out[37]= Datum[value → 120, text → Hello!]
```

```
In[38]:= EchoDatumArray[{
           ReplaceSchemaInstanceValue[datum, "value" → 7!],
           ReplaceSchemaInstanceValue[datum, "value" → 8!],
           ReplaceSchemaInstanceValue[datum, "value" → 9!]}]
Out[38]= {Datum[value → 5040, text → Hello!],
          Datum[value → 40 320, text → Hello!], Datum[value → 362 880, text → Hello!]}
```

Images: Plot

Strings are a natural way to transmit graphics within web services. In order to ensure its integrity, binary data is encoded in base64. This example demonstrates how to use binary data types.

```
In[39]:=  << WebServices`

In[40]:=  InstallService[
            "http://localhost:8080/webMathematica/Examples/WebServices/Plot.m?wsdl"]
Out[40]=  {PlotExpression, PlotString}

In[41]:=  ? PlotString
```

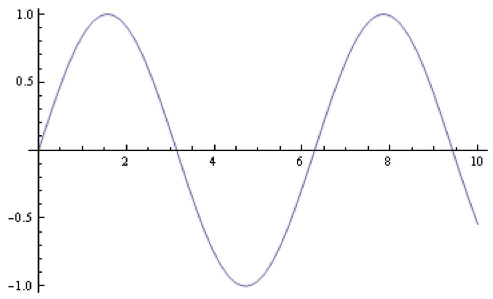
```
SchemaBase64Binary PlotString[expr_String,limit_Integer]

Plots a function of x from 0 to a limit. The
     function is passed in as an InputForm string and the result is a GIF.
```

In the `Plot` example, `PlotString` takes a string and an integer as parameters and returns a base64 binary string. The function converts the string into a *Mathematica* expression and then plots this as a function of x from 0 to a limit specified by the integer. The service returns a list of base64 binary bytes wrapped in a head of `SchemaBase64Binary`. This head is used to identify this data as base64 binary. `FromCharacterCode` can be used to decode the base64 binary data into a string consisting of GIF data. This string may be imported to a GIF with `ImportString`.

```
In[42]:=  ImportString[FromCharacterCode[First[PlotString["Sin[x]", 10]]], "GIF"]
```

Out[42]=



Mathematical Data: Integrate

Web Services Link supports mathematical data using strings, typeset images, MathML, or *Mathematica* expressions. This example demonstrates using mathematical data.

```
In[43]:= << WebServices`

In[44]:= InstallService[
          "http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m?wsdl"]
Out[44]= {IntegrateExpression, IntegrateExpressionToGIF, IntegrateMathML,
          IntegrateMathMLToGIF, IntegrateString, IntegrateStringToGIF}
```

Strings

Mathematical data can be used with a web service by using *Mathematica* expressions within a string. The string data type has some limitations for communicating mathematical data. For one, it is one-dimensional. If you use `InputForm` or `FullForm` to represent the data, information about its presentation may be left out. It is also worth noting that while all web services support strings, they may not all support parsing *Mathematica* expressions from a string. However, sending a string to web service is probably the simplest way to communicate mathematical data, if not the most advanced.

This example demonstrates using strings to communicate mathematical data.

```
In[45]:= ? IntegrateString

         String IntegrateString[str_String]

         Integrates an equation with respect to x. The function is
            passed in as an InputForm string and the result is an InputForm string.

In[46]:= result = IntegrateString["Sin[x]"]
Out[46]= -Cos[x]
```

Web Services Link does not take a string and automatically convert it into a *Mathematica* expression. This is because all strings are not necessarily *Mathematica* expressions.

```
In[47]:= result // InputForm
Out[47]= "-Cos[x]"
```

But because this is *Mathematica,* the result can easily be turned into an expression.

```
In[48]:=  expr = ToExpression[result]
Out[48]=  -Cos[x]
```

Typeset Images

Mathematical data can be presented with a web service by using images. An image has one major limitation for communicating mathematical data. An image cannot be easily manipulated or processed. This means data cannot be easily taken from it except visually and it cannot be changed and updated easily. Images are often useful, however, because typeset equations are not supported on nearly as many systems as images. So generating an image of a typeset equation is a nice way to present typeset equations on systems that do not support them.

The following example demonstrates using images to communicate mathematical data. The service returns a list base64 binary bytes wrapped in a head of `SchemaBase64Binary`. This head is used to identify this data as base64 binary. `FromCharacterCode` can be used to decode the base64 binary data into a string consisting of GIF data. This string may be imported to a GIF with `ImportString`.

```
In[49]:=  ? IntegrateStringToGIF

          SchemaBase64Binary IntegrateStringToGIF[str_String]

          Integrates an equation with respect to x. The
               function is passed in as an InputForm string and the result is a GIF.
```

```
In[50]:=  ImportString[FromCharacterCode[First[IntegrateStringToGIF["x^2"]]], "GIF"]
```

Out[50]= $\frac{x^3}{3}$

Mathematica Expression

Mathematical data can be presented with a web service by using *Mathematica* expressions. Expressions are how data is represented in *Mathematica*. Expressions address some of the limitations of strings and images. However, expressions do have a few limitations of their own, the biggest being that they are not supported on all systems. However, since expressions are the standard for using mathematical data with *Mathematica*, they can be very useful for interfacing with other *Mathematica* systems.

The following example demonstrates using *Mathematica* expressions to communicate mathematical data. The service returns an expression wrapped in a head of `SchemaExpr`. This head is used to identify the following data as an expression.

```
In[51]:= ? IntegrateExpression
```

```
SchemaExpr IntegrateExpression[expr_SchemaExpr]

Integrates an equation with respect to x. The function is passed
     in as an InputForm Expression and the result is an InputForm Expression.
```

```
In[52]:= IntegrateExpression[SchemaExpr[Sin[x]]]
Out[52]= SchemaExpr[-Cos[x]]
```

Web Services Link encodes *Mathematica* expressions as ExpressionML within web services messages. The ExpressionML is automatically converted into an expression.

MathML

Mathematical data can be presented with a web service by using MathML. MathML is the standard for representing mathematical data across systems. MathML addresses some of the limitations that strings and images have. However, MathML does have a few limitations of its own, the biggest being that it is not supported on all systems. Another limitation is its complexity. MathML is more difficult to look at than something like `InputForm`. However, since it is the standard for using mathematical data outside of *Mathematica*, it can be very useful for interfacing with other systems that do not use *Mathematica*.

This example demonstrates using MathML to communicate mathematical data. The service returns symbolic MathML wrapped in a head of `SchemaMathML`, which is used to identify the data as MathML.

```
In[53]:= ? IntegrateMathML
```

```
SchemaMathML IntegrateMathML[mathml_SchemaMathML]

Integrates an equation with respect to x. The
     function is passed in as an MathML and the result is MathML.
```

```
In[54]:= result =
          IntegrateMathML[SchemaMathML[XML`MathML`ExpressionToSymbolicMathML[Sin[x]]]]
Out[54]= SchemaMathML[XMLElement[{http://www.w3.org/1998/Math/MathML, math},
           {}, {XMLElement[{http://www.w3.org/1998/Math/MathML, mrow},
             {{http://www.w3.org/2000/xmlns/, ns0} → http://www.w3.org/1998/Math/MathML},
             {XMLElement[{http://www.w3.org/1998/Math/MathML, mo}, {}, {-}],
              XMLElement[{http://www.w3.org/1998/Math/MathML, mrow}, {},
               {XMLElement[{http://www.w3.org/1998/Math/MathML, mi}, {}, {cos}],
                XMLElement[{http://www.w3.org/1998/Math/MathML, mo}, {}, {}],
                XMLElement[{http://www.w3.org/1998/Math/MathML, mo}, {}, {(}],
                XMLElement[{http://www.w3.org/1998/Math/MathML, mi}, {}, {x}],
                XMLElement[{http://www.w3.org/1998/Math/MathML, mo}, {}, {)}]}]}]}]]
```

Web Services Link does not take a MathML expression and automatically convert it into a *Mathematica* expression, because not all MathML instances are necessarily supported by *Mathematica* expressions. But because this is *Mathematica,* the result can easily be turned into an expression.

```
In[55]:= XML`MathML`SymbolicMathMLToExpression[First[result]]
Out[55]= -Cos[x]
```

SOAP Headers

SOAP headers are an important component to web services because they allow web services to process metadata separate from the body of a SOAP message. The body will contain information directly related to the web service operation being invoked. However, a header may contain more general content that is used across any number of the web service operations. For example, a header may contain a username and password. It is useful to include the username and password as a header, because a separate component can be used to process the username and password without needing to know anything about the body.

Web Services Link supports headers by including the header as part of the function signature of an operation. So it will look like any other parameter of the operation. If an operation requires two strings defining the username and password as headers, then these two strings will be expected when calling this function.

```
In[56]:= << WebServices`

In[57]:= InstallService[
          "http://localhost:8080/webMathematica/Examples/WebServices/SOAPHeaders.m?wsdl"]
Out[57]= {Authenticate}

In[58]:= ? Authenticate
```

```
String Authenticate[Username_String,Password_String]

Simple authentication web service example.
```

```
In[59]:= Authenticate["anonymous", "password"]
Out[59]= Hello anonymous.
```

Sessions

Sessions are used by web services to persist user data over multiple calls to a web service. For example, they can be used to save a user's settings, data, or other things. Sessions are supported by *Web Services Link*. However, there are no special functions or options. If a web service provides operations that use sessions, then sessions should be used by the Java client.

Here is an example that uses sessions. This example returns how many times a user has accessed the operation.

```
In[60]:= << WebServices`

In[61]:= InstallService[
          "http://localhost:8080/webMathematica/Examples/WebServices/Session.m?wsdl"]
Out[61]= {GetAccesses}

In[62]:= ? GetAccesses
```

```
Integer GetAccesses[]

Gets the number of times this function has been accessed.
```

```
In[63]:= GetAccesses[]
Out[63]= 0
```

Additional calls to `GetAccesses` increments the counter.

```
In[64]:= GetAccesses[]
Out[64]= 1
```

Authentication

Authentication is used by some web services to validate a user. This is based on functionality provided by HTTP. Digest and basic authentication are supported. A user can set the username and password by using the `Username` and `Password` options. These can be passed into any web service function and used for authentication.

```
In[65]:= << WebServices`

In[66]:= InstallService[
          "http://localhost:8080/webMathematica/Examples/WebServices/Hello.m?wsdl",
          "Username" → "user", "Password" → "password"]
Out[66]= {HelloWorld}
```

HTTP authentication is reused for supplemental calls to a URL. The username and password are stored in memory and will be required every time the Java Virtual Machine (JVM) is shut down.

```
In[67]:= HelloWorld[]
Out[67]= Hello! The current date is {2007, 1, 31, 12, 32, 35.848595}
```

Timeouts

A timeout may be required when calling a web service operation. A user can set the timeout by using the `Timeout` option. This can be passed into any web service function. The value of the option must be a positive integer specifying the timeout in milliseconds.

```
In[68]:= << WebServices`

In[69]:= InstallService[
          "http://localhost:8080/webMathematica/Examples/WebServices/Hello.m?wsdl"]
Out[69]= {HelloWorld}
```

Here is an example setting the timeout to 10 milliseconds.

```
In[70]:= HelloWorld[Timeout → 10]

              InvokeServiceOperation::native : An error occurred: Read timed out ≫

Out[70]= $Failed
```

Here is an example setting the timeout to 1 second.

```
In[71]:= HelloWorld[Timeout → 1000]
Out[71]= Hello! The current date is {2007, 3, 29, 14, 38, 39.6414826}
```

Advanced Topics in Web Services

Working with Messages

Sometimes it is useful to work directly with the request message that is sent to a web service. This message may be retrieved using the `ToServiceRequest` function. `ToServiceRequest` is called by providing a web service function as the first parameter and its parameters as the additional parameters. The function will not invoke the web service operation. Rather the function will return the symbolic XML representation of the request message that would generally be sent to the web service.

```
In[72]:= << WebServices`

In[73]:= InstallService[
          "http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m?wsdl"]

Out[73]= {IntegrateExpression, IntegrateExpressionToGIF, IntegrateMathML,
          IntegrateMathMLToGIF, IntegrateString, IntegrateStringToGIF}

In[74]:= symbolicXML = ToServiceRequest[IntegrateString, "Sin[x]"]

Out[74]= XMLObject[Document][{},
          XMLElement[{soapenv, Envelope}, {{xmlns, soapenv} → http://schemas.xmlsoap.org/soap/envelope/,
            {xmlns, xsd} → http://www.w3.org/2001/XMLSchema,
            {xmlns, xsi} → http://www.w3.org/2001/XMLSchema-instance,
            {xmlns, soapenc} → http://schemas.xmlsoap.org/soap/encoding/},
           {XMLElement[{soapenv, Body}, {}, {XMLElement[{ns0, IntegrateString},
                {{xmlns, ns0} → http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m},
                {XMLElement[{, str}, {}, {Sin[x]}]}]}]}], {}]
```

A user may change the request message to have different values or different attributes. In fact, the user may create an entirely new message. However, the message must still follow the format specified in the SOAP specification. Using `ToServiceRequest` gives the user a nice starting point. The request message can then be invoked using the `InvokeServiceOperation` function. The first parameter of this function is the endpoint URI or the symbol representing a web service function. The web service function specifies the endpoint URI internally along with other options for invoking the service. The second argument is the symbolic XML representation of the request message.

```
In[75]:= InvokeServiceOperation[IntegrateString, symbolicXML]

Out[75]= XMLObject[Document][{}, XMLElement[{http://schemas.xmlsoap.org/soap/envelope/, Envelope},
           {{http://www.w3.org/2000/xmlns/, soapenv} → http://schemas.xmlsoap.org/soap/envelope/},
           {XMLElement[{http://schemas.xmlsoap.org/soap/envelope/, Body}, {},
             {XMLElement[{http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m,
                IntegrateStringReturn}, {{http://www.w3.org/2000/xmlns/, ns1} →
                 http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m},
               {XMLElement[{, element}, {}, {-Cos[x]}]}]}]}], {}]
```

The result of this call is the response message. Sometimes it is useful to work directly with the response message. However, web service functions by default return a deserialized version of the response message. The original response message may be retrieved using the `InvokeServiceOperation` function. The first parameter is the web service function, and the function's parameters follow. `InvokeServiceOperation` invokes the service and returns the entire response message in symbolic XML. This will allow the user to customize how the data is deserialized into *Mathematica* representation.

```
In[76]:= response = InvokeServiceOperation[IntegrateString, "Sin[x]"]

Out[76]= XMLObject[Document][{}, XMLElement[{http://schemas.xmlsoap.org/soap/envelope/, Envelope},
           {{http://www.w3.org/2000/xmlns/, soapenv} → http://schemas.xmlsoap.org/soap/envelope/},
           {XMLElement[{http://schemas.xmlsoap.org/soap/envelope/, Body}, {},
             {XMLElement[{http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m,
                 IntegrateStringReturn}, {{http://www.w3.org/2000/xmlns/, ns1} →
                  http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m},
                {XMLElement[{, element}, {}, {-Cos[x]}]}]}]}], {}]
```

Once a web service function is called using `InvokeServiceOperation`, the results come back in the form of a SOAP envelope. `FromServiceResponse` can be used to extract the results from the SOAP envelope. The data will be deserialized into the proper *Mathematica* types specified by the operation symbol.

```
In[77]:= FromServiceResponse[IntegrateString, response]

Out[77]= -Cos[x]
```

Working without a WSDL

Web services will work without a WSDL file, and sometimes the user may be forced into a situation where a WSDL file is not available. Although it is much more convenient to use web services with a WSDL file, it is possible to use the *Mathematica* web services client without a WSDL. A user can build the SOAP message from scratch in *Mathematica* using symbolic XML.

`ToServiceRequest` can also be used to build a SOAP message using XML Schema definitions. `DefineSchema` can be used to define type definitions for a web service. In this example, the `IntegrateString` web services have a root element of `IntegrateString`, have a namespace of http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m, and take a string named `str` as a parameter.

```
In[78]:= << WebServices`

In[79]:= DefineSchema[
          "http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m",
          {
           IntegrateString : IntegrateStringType[str_String]
          }
         ]
Out[79]= {IntegrateString}
```

`NewSchemaInstance` can be used to create an instance of the type just defined and this can be passed into `ToServiceRequest`, which takes two parameters. First, it takes a list of parameters for the body of the message. Second, it takes a list of parameters for the header.

```
In[80]:= ToServiceRequest[{NewSchemaInstance[IntegrateString, {"str" → "Sin[x]"}]}, {}]
Out[80]= XMLObject[Document][{},
          XMLElement[{soapenv, Envelope}, {{xmlns, soapenv} → http://schemas.xmlsoap.org/soap/envelope/,
            {xmlns, xsd} → http://www.w3.org/2001/XMLSchema,
            {xmlns, xsi} → http://www.w3.org/2001/XMLSchema-instance,
            {xmlns, soapenc} → http://schemas.xmlsoap.org/soap/encoding/},
           {XMLElement[{soapenv, Body}, {}, {XMLElement[{ns1, IntegrateString},
                {{xmlns, ns1} → http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m},
                {XMLElement[{, str}, {}, {Sin[x]}]}]}]}], {}]
```

`ToServiceRequest` supports two options as well. First, `OperationName` can be used to wrap an element that identifies a web service operation around the parameters. This is often useful when using RPC-style web services or when mapping document-style web services to functions. Second, `EncodingStyleURI` may be used to set the encoding style. The only encoding style supported is SOAP encoding as demonstrated next. This adds type attributes to the parameters.

```
In[81]:= msg = ToServiceRequest[{"str" → "Sin[x]"}, {},
            OperationName →
             {"http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m",
              "IntegrateString"},
            EncodingStyleURI → {"http://schemas.xmlsoap.org/soap/encoding/"}
           ]
Out[81]= XMLObject[Document][{},
          XMLElement[{soapenv, Envelope}, {{xmlns, soapenv} → http://schemas.xmlsoap.org/soap/envelope/,
            {xmlns, xsd} → http://www.w3.org/2001/XMLSchema,
            {xmlns, xsi} → http://www.w3.org/2001/XMLSchema-instance,
            {xmlns, soapenc} → http://schemas.xmlsoap.org/soap/encoding/},
           {XMLElement[{soapenv, Body}, {{soapenv, encodingStyle} →
                http://schemas.xmlsoap.org/soap/encoding/}, {XMLElement[{ns0, IntegrateString},
                {{xmlns, ns0} → http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m},
                {XMLElement[{, str}, {{xsi, type} → xsd:string}, {Sin[x]}]}]}]}], {}]
```

`InvokeServiceOperation` can then be used to invoke the service with the message that has been generated. `InvokeServiceOperation` requires two parameters. First the endpoint is required, which specifies the URL where the message will be sent. The second parameter is the message. A couple of options are supported as well. `SOAPActionURI` may be used to set a SOAP

Action header. `TransportStyleURI` may be used to set the transport style (although http is the only transport currently supported). `Username` and `Password` may be used for authentication and `Timeout` may be used to set a timeout.

```
In[82]:= response = InvokeServiceOperation[
           "http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m", msg]
Out[82]= XMLObject[Document][{}, XMLElement[{http://schemas.xmlsoap.org/soap/envelope/, Envelope},
           {{http://www.w3.org/2000/xmlns/, soapenv} → http://schemas.xmlsoap.org/soap/envelope/},
           {XMLElement[{http://schemas.xmlsoap.org/soap/envelope/, Body}, {},
             {XMLElement[{http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m,
                IntegrateStringReturn}, {{http://www.w3.org/2000/xmlns/, ns1} →
                 http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m},
               {XMLElement[{, element}, {}, {-Cos[x]}]}]}]}], {}]
```

`FromServiceResponse` can be used to process a SOAP message using XML Schema definitions. `DefineSchema` can be used to define a return type for a web service. In this example, the `IntegrateString` returns an element with a string named `element`. `FromServiceResponse` takes the message as a parameter and the option `ReturnType` is used to specify the return type.

```
In[83]:= DefineSchema[
          "http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m",
          {
           IntegrateStringReturnType[element_String]
          }
         ]
Out[83]= {IntegrateStringReturnType}
```

```
In[84]:= FromServiceResponse[response, ReturnType → IntegrateStringReturnType]
Out[84]= -Cos[x]
```

`InstallServiceOperation` can be used to install these definitions as a *Mathematica* function. It takes a symbol as its first parameter that the function definition will be associated with. It takes a URL as its second parameter that will define where a message is sent. The third parameter is a list of parameters for the function. These parameters are defined using XML Schema element definition symbols defined using `DefineSchema`. The fourth parameter is a list of header parameters. Finally, options may be used. Each option from `ToServiceRequest`, `InvokeServiceOperation`, and `FromServiceResponse` is valid.

```
In[85]:= InstallServiceOperation[
          integrateString,
          "http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m",
          TypeElements[ElementType[IntegrateString]],
          {},
          OperationName →
           {"http://localhost:8080/webMathematica/Examples/WebServices/Integrate.m",
            "IntegrateString"},
          ReturnType → IntegrateStringReturnType]
Out[85]= integrateString
```

Once the function is installed, it may be used just like other functions installed using *Web Services Link*.

```
In[86]:= ? integrateString
```

```
String integrateString[str_String]
```

```
In[87]:= integrateString["Cos[x]"]
Out[87]= Sin[x]
```

Debugging

Web Services Link provides a few symbols that can help in debugging. The debugging symbols are defined in the following table.

Symbol	Description
`$PrintServiceRequest`	use the `Print` function to print the message in XML format sent to a web service
`$PrintServiceResponse`	use the `Print` function to print the message in XML format received from a web service before it is deserialized into a rule syntax expression
`$PrintWSDLDebug`	print information about web services being installed that can be used to debug; information printed includes option values, parameter signatures, and the endpoint URI for web service operation
`$PrintShortErrorMessages`	specify whether error messages will be shortened for the user to avoid long intimidating error messages

Debugging symbols.

Example Applications

Amazon Web Services Example

Amazon.com is a well-known web retailer that specializes in books, music, movies, and many other products. Amazon has made a web service available that allows developers to interface with their database of products. A user can search for specific products, authors, artists, and so on. A query will return information (price, description, location, and so on) about the products that are found. This example demonstrates a *Mathematica* interface to the Amazon web service.

The Amazon web services are good demonstrations of using web services to retrieve data. This example searches the Amazon database for books written by certain authors, published by a certain manufacturer, or containing a certain keyword. The example code then places the results in a new notebook. The example has one option. Setting the `ShowPictures` option to `True` will force the code to retrieve and display the cover image of the book.

To try this example, evaluate all the initialization cells. (You can do this using **Evaluation ▸ Evaluate Initialization Cells**.) Then go to the examples here.

■ Code

```
BeginPackage["WebServices`Examples`Amazon`", {"WebServices`", "JLink`"}];

AuthorSearch;
ManufacturerSearch;
KeywordSearch;
ShowPictures;

Begin["`Private`"];

InstallService["http://soap.amazon.com/schemas3/AmazonWebServices.wsdl"];

Options[AuthorSearch] = {ShowPictures → False};

AuthorSearch[keyword_String, pages_Integer, options___?OptionQ] :=
 Module[{result}, result = AuthorSearchRequest[
    AuthorRequest["author" → keyword, "page" → ToString[pages], "mode" → "books",
     "tag" → "webservices-20", "type" → "lite", "devtag" → "D3VZJS46JD5U9D"]];
  If[result === $Failed, Return[$Failed]];
  result = result["Details"];
  NotebookPut[
   resultNotebook["Amazon Author Search", keyword, result, options]]
 ]
```

```
ManufacturerSearch[keyword_String, pages_Integer, options___?OptionQ] :=
  Module[{result}, result = ManufacturerSearchRequest[ManufacturerRequest[
      "manufacturer" → keyword, "page" → ToString[pages], "mode" → "books",
      "tag" → "webservices-20", "type" → "lite", "devtag" → "D3VZJS46JD5U9D"]];
   If[result === $Failed, Return[$Failed]];
   result = result["Details"];
   NotebookPut[
    resultNotebook["Amazon Manufacturer Search", keyword, result, options]]
  ];

KeywordSearch[keyword_String, pages_Integer, options___?OptionQ] :=
  Module[{result}, result = KeywordSearchRequest[KeywordRequest[
      "keyword" → keyword, "page" → ToString[pages], "mode" → "books",
      "tag" → "webservices-20", "type" → "lite", "devtag" → "D3VZJS46JD5U9D"]];
   If[result === $Failed, Return[$Failed]];
   result = result["Details"];
   NotebookPut[
    resultNotebook["Amazon Keyword Search", keyword, result, options]]
  ];

resultNotebook[title_String, keyword_String, result_List, options___?OptionQ] :=
 Notebook[Flatten[{
    Cell[title, "Subtitle"],
    Cell["Results for: " <> keyword, "Section"],
    resultCells[#, options] & /@ result
   }], CellGrouping → Manual, WindowTitle → (title <> ": " <> keyword)]

resultCells[d_, options___?OptionQ] :=
 Module[{url, asin, productName, authorNames, releaseDate, manufacturer,
   imageUrlMedium, listPrice, ourPrice, usedPrice, showPictures, cellList},
  {showPictures} = {ShowPictures} /. Flatten[{options}] /. Options[AuthorSearch];
  url = d["Url"];
  asin = d["Asin"];
  productName = d["ProductName"];
  authorNames = d["Authors"];
  releaseDate = d["ReleaseDate"];
  manufacturer = d["Manufacturer"];
  imageUrlMedium = d["ImageUrlMedium"];
  listPrice = d["ListPrice"];
  ourPrice = d["OurPrice"];
  usedPrice = d["UsedPrice"];
  authorNames = ToString[authorNames];
  authorNames = StringDrop[authorNames, 1];
  authorNames = StringDrop[authorNames, -1];
  If[TrueQ[showPictures], imageUrlMedium = getImageData[imageUrlMedium];
   If[StringMatchQ[imageUrlMedium, "GIF*"], imageCell =
     ToExpression[ExportString[ImportString[imageUrlMedium, "GIF"], "MGFCell"]],
    imageCell = ToExpression[ExportString[
       ImportString[imageUrlMedium, "JPEG"], "MGFCell"]]];];
  {Cell[CellGroupData[Flatten[{
       Cell[TextData[{
          ButtonBox[StyleBox[productName, FontWeight → "Bold"],
           ButtonData → {URL[url], None}, ButtonStyle → "Hyperlink"],
          " by ", StyleBox[authorNames, FontSlant → "Italic"]}],
        "Text", ShowGroupOpenCloseIcon → True], If[TrueQ[showPictures],
        Cell[TextData[imageCell], "Text"], {}], Cell[TextData[
         {StyleBox["Release Date: ", FontWeight → "Bold"], releaseDate, "\n",
          StyleBox["ASIN: ", FontWeight → "Bold"], asin, "\n",
          StyleBox["List Price: ", FontWeight → "Bold"], listPrice, "\n",
          StyleBox["Our Price: ", FontWeight → "Bold"], ourPrice}], "Text"]}],
     Closed]]}]
```

```
getImageData[url_String] :=
 JavaBlock[Module[{u, connection, stream, input = "", numRead = 0,
    bytesRead = 0, buf}, u = JavaNew["java.net.URL", url];
   connection = u@openConnection[];
   stream = connection@getInputStream[];
   If[stream == $Failed, Return[URLError["Failed to open URL stream"]]];
   buf = JavaNew["[B", 5000];
   While[(numRead = stream@read[buf]) > 0, AddTo[bytesRead, numRead];
    input = StringJoin[input, ToString[FromCharacterCode[
        If[# < 0, # + 256, #] & /@ Take[JavaObjectToExpression[buf], numRead]]]];];
   stream@close[];
   input]]

End[];
EndPackage[];
```

Examples

```
AuthorSearch["Stephen Wolfram", 1]

ManufacturerSearch["Addison-Wesley", 1]

KeywordSearch["Mathematica", 1, ShowPictures → True]
```

Google Web Services Example

Google.com is a well-known web search engine. Google has made a web service available that allows developers to interface with their search engine within their own applications. A user can search for any topic on the web. A query will return data about the web pages that are found. This example demonstrates a *Mathematica* interface to the Google web service.

The Google web service provides a good demonstration of the use of web services to retrieve data. This example searches the Google database for web pages containing a certain keyword. The example code then places the results in the notebook.

To try this example, evaluate all the initialization cells. (You can do this using **Evaluation ▸ Evaluate Initialization Cells**.) Then go to the examples here.

■ Code

```
<< WebServices`

InstallService["http://api.google.com/GoogleSearch.wsdl"];

Options[GoogleSearch] = {
   MaxHits → 10,
   Filter → True,
   IncludeURL → True
  };

$backgrounds = {RGBColor[0.960784, 0.878431, 0.666667],
   RGBColor[0.964706, 0.929412, 0.839216]};

GoogleSearch::err = "Search Error.";
```

```
GoogleSearch::nomatch = "No page containing `1` found.";

GoogleSearch[str_String, opts___] :=
 Module[{lis, results, matches, filterQ, urlQ},
  {matches, filterQ, urlQ} =
   {MaxHits, Filter, IncludeURL} /. Flatten[{opts, Options @ GoogleSearch}];

  results = GoogleSearchService`GoogleSearchPort`doGoogleSearch[
    "3HB82PdQFHITKPmbmm5G/9aTfVHx/m95", str,
    0, matches, filterQ, "", True, "", "", ""];
  If[results === $Failed,
   Message[GoogleSearch::err],
   lis = results["resultElements"];
  ];
  If[lis === Null,
   Message[GoogleSearch::nomatch, str],
   $bg = 0;
   Scan[CellPrint, searchResultCells[lis, urlQ]]]
 ]

searchResultCells[lis_, urlQ_] :=
 cellTemplate[formatSingleResult[#, urlQ]] & /@ lis

cellTemplate[lis_] :=
 Cell[TextData @ Flatten @ {lis}, "Print",
  FontFamily → "Times",
  CellMargins → {{Inherited, Inherited}, {0, 0}},
  CellDingbat → ToString[++$bg],
  Background → Part[$backgrounds, Mod[$bg, 2, 1]]
 ]

formatSingleResult[result_, urlQ_] :=
 Block[{u, t},
  {
   " ",
   webLink[
    StyleBox[formatTitle @result["title"], FontWeight → "Bold"], result["URL"]],
   If[urlQ, {"\n", webLink[StyleBox[result["URL"], "SmallText",
       FontColor → GrayLevel[0.4]], result["URL"]]}, {}]
  }
 ]

webLink[content_, url_] :=
 ButtonBox[content,
  ButtonData :→ {URL[url], None},
  BaseStyle → "Hyperlink"
 ]

formatTitle[str_String] := translateAllEntities[str]

decodeString[str_] := StringReplace[str, {
   "&lt;" → "<",
   "&gt;" → ">",
   "&amp;" → "&",
   "&quot;" → "\"",
   "\\" → "\\[Backslash]"
  }]

translateAllEntities[str_] := StringReplace[str, Reverse /@
    System`Convert`MLStringDataDump`$UnicodeToHTML4Entities] // decodeString
```

Example

Find the top 10 wolfram.com web pages containing all the given strings.

```
GoogleSearch["Mathematica"]
```

```
 1  Wolfram.com - <b>Mathematica</b>
    http://www.wolfram.com/

 2  <b>Mathematica</b>: The Way the World Calculates
    http://www.wolfram.com/products/mathematica/index.html

 3  <b>Mathematica</b> Policy Research, Inc.: Home
    http://www.mathematica-mpr.com/

 4  <b>Mathematica</b> - Wikipedia, the free encyclopedia
    http://en.wikipedia.org/wiki/Mathematica

 5  <b>Mathematica</b> Information Center
    http://library.wolfram.com/

 6  The Integrator--Integrals from <b>Mathematica</b>
    http://integrals.wolfram.com/

 7  Wolfram Research Documentation Center: A Collection of Online <b>...</b>
    http://documents.wolfram.com/

 8  Wolfram <b>Mathematica</b> Documentation
    http://documents.wolfram.com/mathematica/

 9  <b>Mathematica</b> Graphics Gallery
    http://gallery.wolfram.com/

10  Technical Support: <b>Mathematica</b>: Questions About <b>Mathematica</b>
    http://support.wolfram.com/mathematica/
```

Search for pages containing all the given strings, and return only the top two results.

```
GoogleSearch["Hermite polynomial", MaxHits → 2]
```

```
 1  <b>Hermite Polynomial</b> -- from Wolfram MathWorld
    http://mathworld.wolfram.com/HermitePolynomial.html

 2  <b>Hermite polynomials</b> - Wikipedia, the free encyclopedia
    http://en.wikipedia.org/wiki/Hermite_polynomials
```

Display less verbose results and do not filter near-duplicate content and host crowding.

```
GoogleSearch["Chebyshev", IncludeURL → False, Filter → False]
```

1	<b>Chebyshev</b> summary
2	<b>Chebyshev</b> Polynomial of the First Kind -- from Wolfram MathWorld
3	Pafnuty <b>Chebyshev</b> - Wikipedia, the free encyclopedia
4	<b>Chebyshev</b> polynomials - Wikipedia, the free encyclopedia
5	<b>Chebyshev&#39;s</b> inequality - Wikipedia, the free encyclopedia
6	<b>Chebyshev</b> filter - Wikipedia, the free encyclopedia
7	<b>Chebyshev</b> Polynomial
8	One tailed version of <b>Chebyshev&#39;s</b> inequality - by Henry Bottomley
9	<b>Chebyshev</b> biography
10	<b>Chebyshev</b> Polynomials

XMethods Web Services Example

XMethods.com is a website that lists publicly available web services. It is a great place for finding web services and advertising web services that you provide. This example queries the XMethods database and builds a notebook listing of each web service listed on XMethods. Each web service is listed with the title, description, and an `InstallService` function that can be used to install and use a service.

This example demonstrates using the XMethods Query Service to discover and use many of the web services available. This example searches the XMethods database for the web services registered. Once the query is finished, a user can conveniently browse through the services and install the services that may be interesting. If `GetServiceSummaries` is called again, the code will match service IDs with cell tags. If an ID exists as a cell tag, no new cells will be added. If an ID does not exist, the appropriate cells are added for the service.

To try this example, evaluate all the initialization cells. (You can do this using **Evaluation ▸ Evaluate Initialization Cells**.) Then go to the examples here.

■ Code

```
BeginPackage["WebServices`Examples`XMethods`", {"WebServices`", "JLink`"}];

GetServiceSummaries;

Begin["`Private`"];

InstallService["http://www.xmethods.net/wsdl/query.wsdl"];

GetServiceSummaries[] := Module[{result}, result = getAllServiceSummaries[];
  If[result === $Failed, Return[$Failed]];
  notebook = EvaluationNotebook[];
  If[NotebookFind[notebook, "Results", All, {CellTags}] === $Failed,
   SelectionMove[notebook, After, EvaluationCell];
   NotebookWrite[notebook, Cell["Results", "Section", CellTags → {"Results"}]];];
  printResult /@ result;
  NotebookFind[notebook, "Results", All, {CellTags}];]

printResult[result_] :=
 Module[{name, id, shortDescription, wsdlURL, publisherID},
  name = result["name"];
  id = result["id"];
  shortDescription = result["shortDescription"];
  wsdlURL = result["wsdlURL"];
  publisherID = result["publisherID"];
  notebook = EvaluationNotebook[];
  If[NotebookFind[notebook, id, All, {CellTags}, AutoScroll → False] === $Failed,
   NotebookFind[notebook, "Results", All, {CellTags}, AutoScroll → False];
   SelectionMove[notebook, After, Cell, AutoScroll → False];
   NotebookWrite[notebook, Cell[CellGroupData[
      {Cell[BoxData[RowBox[{name, StyleBox[" from ", FontWeight → "Plain"],
           StyleBox[publisherID, FontWeight → "Plain", FontSlant → "Italic"],
           ButtonBox["(wsdl)", ButtonData → {URL[wsdlURL], None},
            ButtonStyle → "Hyperlink"]}]], "Subsection",
        ShowGroupOpenCloseIcon → True, CellDingbat → None, CellTags → {id}],
       Cell[shortDescription, "Text", ShowCellBracket → False,
        CellMargins → {{55, Inherited}, {Inherited, Inherited}},
        CellFrame → True, Background → GrayLevel[0.850004]],
       Cell[BoxData[RowBox[{"InstallService[\"" <> wsdlURL <> "\"]"}]],
        "Input", ShowCellBracket → False, CellFrame → True,
        CellMargins → {{55, Inherited}, {Inherited, Inherited}},
        Background → GrayLevel[0.850004]]}, Open]]];];]

End[];
EndPackage[];
```

Example

```
GetServiceSummaries[]
```

TerraService Web Services Example

Terraservice.net is a website that provides access to aerial imagery and topographical maps of the United States. A web service has been provided that allows developers to access this data. Developers can use this data to provide maps and aerial imagery in their applications. This allows users of *Mathematica* to use this data in *Mathematica*. This example demonstrates an interface to TerraService using *GUIKit* and *Mathematica*.

Microsoft TerraServer Web Service is a programmable interface to the popular Microsoft TerraServer online database of high-resolution United States Geological Survey (USGS) aerial imagery (DOQs) and scanned USGS topographical maps (DRGs). The user can specify the place to display by using the **Place Search** field. The place is expected to be a single string containing the city or well-known place name, state name, and country name separated by comma characters, such as "San Francisco, CA, USA". Any of the three values may be missing. The navigation panel can be used to move in a particular direction. The **Latitude** and **Longitude** fields can be used to specify a place at a particular latitude and longitude. The user can zoom in and zoom out using the `Scale` drop-down menu. The user can switch between aerial photographs and topological maps using the `Theme` drop-down menu.

More information about TerraService may be found at http://terraservice.net/.

To try this example, you must have *GUIKit* installed. Then go to the examples here and evaluate the cell.

```
Needs["GUIKit`"];
GUIRun["TerraService"]
```