User Guide
Version 2.11

Note: This documentation is for Genero 2.11. See the corresponding on-line documentation at the Web site **http://www.4js.com/online_documentation** for the latest updates. Please contact your nearest support center if you encounter problems or errors in the on-line documentation.

Genero Web Services Extension

Table Of Contents

Introduction to Web Services

This page provides an introduction to Web Services with the Genero Web Services Extension (GWS). It is intended to help those using GWS for the first time to understand basic Web Services concepts, and to quickly start their development with the Genero tools.

Summary:

Concepts

In general, Web services are a standard way of communicating between applications over an intranet or the Internet. They define how to communicate between two entities:

- A server that exposes services
- A client that consumes the services

For example, a server could expose a "StockQuotation" service that responds to an operation "getQuote". For the "getQuote" operation, the input message is a stock symbol as a string, and the output message is a stock value as a decimal number.

The "getQuote" operation could be a function written in Genero BDL and published on the server. This function retrieves the stock value for the stock symbol passed in, and returns this stock value.

On the client side, the Web service client application calls the function as if it were a local function, passing the stock symbol and storing the returned value in a variable. For example, if the Web Service operation is named **WebService_StockQuotation_getQuote** and the local variable is **svalue,** you would call the Web Service as follows:

```
01 LET svalue = WebService_StockQuotation_getQuote( "MyStockSymbol" )
```

For more information on the steps for creating a Web Service server or client, refer to the Tutorial: Writing a GWS Server application and Tutorial: Writing a Client application respectively.

Web Services and SOA

Service Oriented Architecture (SOA) is a philosophy of how to connect systems and exchange data to solve business problems. Rather than concentrating on a specific task or transaction, SOA addresses how to use data from various sources, reduce human work, and mitigate the effects of change in a business process and its supporting systems.

The SOA defines the services to be provided, and Web Services are the means of implementing those services. Web Services provide a platform-neutral technology to connect multiple systems in a flexible manner, where the platform-neutrality helps insulate a SOA from changes to the underlying systems.

Web Services work by answering requests for information and returning well defined, structured XML documents. Because XML is simple text and Web Services can be invoked via the hypertext transfer protocol (HTTP), it does not matter what platform runs the Web Service, or what platform receives the XML document.

An SOA's resilience to change is accomplished by adhering to good Web Services design practices:

- Build a Web Service that performs a specific task
- Have a rigid structure for the data

Web Services tell exactly how to ask for the information in an XML document written using the Web Services Descriptive Language (WSDL). This self-describing document describes the service the Web Service will perform and how to form the request for its data. Each Web Service must have an associated WSDL document, so that developers and applications know what to expect from the Web Service, and how to invoke it.

Migrating to SOA and Web Services

Developing an SOA and moving to Web Services is an iterative and evolutionary process, and requires work and diligent design. When switching to Web Services from another integration method, it is recommended to initially focus on shorter term business benefits, targeting an SOA and Web Services project that has tangible goals with measurable benefits.

Once an SOA starts to contain useful services, these services can be arranged together in a workflow that automates a business process. Web Services can be reused to answer new questions, implemented as new business services in an SOA. A well-defined Web Service does not contain business logic or business process information. Because each Web Service in an SOA can be called individually to perform a specific task, they can be arranged (orchestrated) together to perform many different business functions. As a result, companies with a mature SOA in place can change business processes through configuring of the orchestration software as opposed to programming individual links between systems.

Planning a Web Service

When creating a Web Service, you not only have to think of the immediate task at hand, but you should also consider growth. You likely want the Web Service to be flexible; to be able to handle different types of input. Prepare the Web Service for what is probable. Developers should think bigger than the needs of a single application. You should think of reusing existing services, and imagine how your services can be reused by colleagues. Security will likely play a larger role than it did previously with existing in-house application infrastructures with programmed links between systems; you will need to become versed in security issues.

Keep the goals of SOA in mind when designing and coding Web Services: flexibility, reusability, and interoperability.

Web Services Standards

Web services are platform-independent and programming language-independent. The World Wide Web consortium defines the Web services standards. For more information about these standards, refer to the "Web services" section of their web site at http://www.w3.org. The Four J's Web Services Extension supports the WSDL 1.1 specification of March 15, 2002 and some previous specifications.

The standards involved in what is commonly called "Web services" are:

XML

XML (Extensible Markup Language) defines a machine-independent way of exchanging data. For example, an XML representation of the following BDL data structure:

```
01 DEFINE Person
02 RECORD Attribute (XMLName="Person")
03    FirstName VARCHAR(32) Attribute (XMLName="FirstName"),
04    LastName  VARCHAR(32) Attribute (XMLName="LastName"),
```

```
05    Age       INTEGER Attribute (XMLName="Age")
06  END RECORD
```

could be:

```
<Person>
 <FirstName>John</FirstName>
 <LastName>Smith</LastName>
 <Age>35</Age>
</Person>
```

The record definition uses a Genero version 2.0 feature that allows you to specify XML attributes for data types.

XML Schema

XML Schema defines the elements, entities, and content model of an XML document. For example, for the above document, the schema could say that the XML document contains an element "Person", and that each "Person" contains one and only one element "FirstName", "LastName", and "Age". The XML Schema has additional capabilities, such as data type control and content restrictions.

An XML Schema allows an XML document to be validated for correctness.

SOAP

SOAP (Simple Object Access Protocol) is a high-level communication protocol between the server and the client. It defines the XML data flow between the server and the client. The "StockQuote" service mentioned in the Concepts section will exchange messages using the following syntax:

Request

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

Response

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getQuoteResponse>
      <stockValue>999.99</stockValue>
    </getQuoteResponse>
  </soap:Body>
</soap:Envelope>
```

SOAP relies on a lower-level protocol for the transport layer.

Genero Web Services use SOAP over HTTP, and can also perform low-level XML and TEXT over HTTP communications on the client side. This allows communication between applications using the core Web technology, taking advantage of the large installed base of tools that can process XML delivered plainly over HTTP as well as SOAP over HTTP.

WSDL

The WSDL (Web Services Description Language) file describes the services offered by a server. It contains:

- The description of the operations offered by the server, with their input and output messages.
- The location of the SOAP server.
- Internal connection and protocol details (transport layer, encoding, namespaces, and so on).

A WSDL description is sufficient to get all the information required to communicate with the SOAP server.

Genero Web Services Extension provides a tool, fglwsdl, that enables Genero client applications to obtain the WSDL description of a Web Service.

HTTP

HTTP (Hypertext Transfer Protocol) is the set of rules for exchanging files (text, graphic images, sound, video, and other multimedia files) on the World Wide Web.

Web Service Styles

The following Styles provided by GWS for a Web Service are WS-I compliant (Web Services Interoperability organization):

- **RPC Style Service (RPC/Literal)** is generally used to execute a function, such as a service that returns a stock option
- **Document Style Service** (**Doc/Literal**) is generally used for more sophisticated operations that exchange complex data structures, such as a service that sends an invoice to an application, or exchanges a Word document.

In addition, the **RPC Style Service (RPC/Encoded)** is provided for backwards compatibility.

New Features

- **Product line 2.1x**
  - FGLGWS 2.11.04
  - Version 2.11
  - Version 2.10
- **Product line 2.0x**
  - Version 2.00

FGLGWS 2.11.04

- One-Way RPC and Document style services
- New options in fglwsdl tool
- Support of WSDL with circular references

One-Way RPC and Document style services

The Genero Web Services library provides two new methods in the WebOperation class to create One-Way operations in services. A One-Way operation means that the server accepts an incoming request, but doesn't return any response back to the client. There is one method called **CreateOneWayRPCStyle** to create an RPC Style operation, and another one called **CreateOneWayDOCStyle** to create a Document Style operation. *A One-Way operation can be used as a logger service for instance, where a client sends a message to the server, but doesn't care about what the server is doing with it.*

See WebOperation for information about these two methods.

New options in fglwsdl tool

The fglwsdl tool was enhanced with the following new options :

- **-b** : Generate code from a WSDL using the binding section instead of the service section
- **-autoNsPrefix** : Determine the prefix for variables and types according to the XML namespace they belong to
- **-nsPrefix** : Set the prefix for a variable or a type belonging to the given XML namespace

The following options have been changed as described below :

- **-o** : If there are several services in one WSDL, they will be generated in the same file with the given base name instead of returning an error

- **-disk** : Retrieves and displays all dependencies to the current directory but there are no sub directories any longer.
- **-prefix** : Accepts patterns %s, %f and %p

See fglwsdl for more information.

Support of WSDL with circular references

The Genero Web Services library has been enhanced to support WSDL with circular references. Actually, the 4GL language doesn't provide a way to define variables or types that refer to themselves. However, to provide better interoperability and a way to handle such circular data, the fglwsdl tool now generates variables or types of **xml.DomDocument** type when circular references are detected during the processing of WSDL files. This gives the user the ability to manipulate the circular data by hand, using the XML DOM API.

See Genero Web Services XML Extension Library for information about the various classes and methods for handling XML documents.

Version 2.11

- Server low-level XML and TEXT over HTTP communication (com)
- XML facet constraints attributes
- Enhancement of the fglwsdl tool with three new options

Server low-level XML and TEXT over HTTP communication (com)

The Genero Web Services com library provides one more class, HTTPServiceRequest, to perform low-level XML and TEXT over HTTP communication on the server side. This allows communication at a very low-level layer, to write your own type of web services.

See HTTPServiceRequest for information about how to write an HTTP server.

XML facet constraints attributes

The Genero Web Services xml library provides 12 new XML attributes to map to simple 4GL variables. These attributes restrict the acceptable value-space for each variable in different ways such as:

- a minimum or a maximum number of XML characters or bytes.
- a strict number of XML characters or bytes.
- a minimum inclusive or exclusive value depending on the datatype.
- a maximum inclusive or exclusive value depending on the datatype,
- a enumeration of authorized values.
- a number of digits and fraction digits.
- how whitespaces have to be handled.

- a regular expression to match. (see Section F of XML Schema Part 2)

See Constraints between simple 4GL and XML datatypes for more details.

Enhancement of the fglwsdl tool with three new options

The fglwsdl tool was enhanced with the following three new options :

- **-disk** : to retrieve locally a WSDL or an XSD with all its dependencies from an URL on the disk
- **-noFacets** : to avoid the generation of the new facet constrain attributes (for compatibility)
- **-regex** : to validate a value against a regular expression as described in the XML Schema specification

See fglwsdl for more information.

Version 2.10

- Genero XML Extension library (xml)
- Low-level XML and TEXT over HTTP and TCP Client communication (com)
- Low-level and asynchronous Client stub generation from WSDL
- Generation of data types from XML schemas (XSD)

Genero XML Extension library (xml)

This library provides classes and methods to perform:

- XML manipulation with a **W3C Document Object Model (DOM) API**
- XML manipulation with a **Streaming API for XML (StAX)**
- Validation of **DOM** documents against XML Schemas
- Serialization of 4GL variables in XML
- Creation of XML Schemas corresponding to 4GL variables

See Genero Web Services XML Extension Library for information about the various classes and methods included in the **xml** library.

Low-level XML and TEXT over HTTP and TCP Client communication (com)

The Genero Web Services **com** library provides two classes, **HTTPRequest** and **HTTPResponse,** to perform low-level XML and TEXT over HTTP communications on the client side. Two more classes, **TCPRequest** and **TCPResponse,** are also provided to perform low-level XML and TEXT over TCP communications on the client side. This

allows communication between applications using the core Web technology, taking advantage of the large installed base of tools that can process XML delivered plainly over HTTP or TCP, as well as SOAP over HTTP.

Specific streaming methods are also available to improve the communication by sending XML to the network even if the serialization process is not yet finished, as well as for the deserialization process.

It is also possible to prevent asynchronous requests from being blocked when waiting for a response, and to perform specific HTTP form encoded requests as specified in HTML 4 or XForms 1.0.

See HTTPRequest and HTTPResponse for information about HTTP classes included in the **com** library.
See TCPRequest and TCPResponse for information about TCP classes included in the **com** library.

Generating low-level and asynchronous Client stubs from WSDL

The **fglwsdl** tool generates all client stubs with the low-level **HTTPRequest** and **HTTPResponse** classes of the **com** library to perform HTTP communications. The low-level generated stub also takes advantage of the streaming methods, if Document Style or RPC-Literal web services are performed. Streaming is not possible with RPC-Encoded web services, as nodes can have references to other nodes in the XML document, requiring the entire document in memory to perform serialization or deserialization.

The **fglwsdl** tool also generates two new 4GL functions for each operation of a Web service. These two functions enable you to perform asynchronous web service operation calls by first sending the request, and retrieving the corresponding response later in the application. This allows you to prevent a 4GL application from being blocked if the response of a web service operation takes a certain amount of time.

See HTTPRequest and HTTPResponse for information about classes included in the **com** library, and the fglwsdl tool for additional information about generated code and asynchronous calls.

Generation of data types from XML schemas (XSD)

Genero Web Services Extension provides an enhanced **fglwsdl** tool that is able to generate 4GL data types from a XML schema. The data types can then be used in your application to be serialized or deserialized in XML. The resulting XML is a valid instance of that XML schema, and validation with a XML validator will succeed.

See the fglwsdl tool for additional information about XSD.

Version 2.00

- Web Services Styles
- Generation of Server and Client from WSDL
- Genero Web Services Extension library (com)
- SOAP Header Management
- HTTPS support on the Client side
- Connection via Proxies on the Client side
- Multiple Web Services in a single DVM
- Service Location Repository
- Serialization of Genero Data Types
- WSHelper.42m Library file

Web Services Styles

You can now choose to use **Document Style Service (Doc/Literal)** or **RPC Literal Style Service (RPC/Literal)** with Genero Web Services Extension (GWS), for .NET compatibility and WS-I compatibility (standards defined by the Web Services Interoperability organization).

- **Document Style Service** allows you to exchange complex data structures, such as database tables or word processing documents (MS.Net default)
- **RPC Literal Style Service** is usually used to execute a function, such as a service that returns a stock option

Note: RPC/Encoded Style Service (Traditional SOAP section 5) is available for backwards compatibility.

See Web Services Styles and the GWS Web Server Tutorial for additional information.

Generation of Server and Client from WSDL

Genero Web Services Extension provides a tool, **fglwsdl**, to allow a Genero application that is accessing a Web Service to obtain the WSDL information for the desired service. It does not matter what language the Web Service is written in. The fglwsdl tool is installed in Genero as part of the Genero Web Services Extension package.

See The fglwsdl Tool for additional information about WSDL.

Genero Web Services Extension Library (com)

You no longer need to create a runner that includes the Genero Web Services Extension package. Instead, your applications import the Genero Web Services Extension library named **com.** This library provides classes and methods that allow you to perform tasks associated with creating GWS Servers and Clients, and managing the Web Services.

See Genero Web Services COM Extension Library for information about the various classes and methods included in the **com** library.

SOAP Header Management

GWS supports SOAP header management through the CreateHeader method in the Web Service class that is part of the Web Services Extension library (com).

See Web Service Class for additional information.

HTTPS support on the client side

GWS supports secure communications through the use of encryption and standard X.509 certificates. Based on the OpenSSL engine, new security features allow a Web Services client to communicate with any secured server over HTTP or HTTPS.

A new tool is provided, **fglpass**, allowing you to encrypt a password from a standard X.509 certificate, and to de-crypt a password you previously encrypted with a certificate.

Entries in the FGLPROFILE file are used to define the configuration for client security.

See FGLPROFILE Password Encryption and FGLPROFILE Configuration, as well as the other pages listed in the Security section, for additional information.

Connections via proxies on the Client Side

You can configure a GWS Client to connect via an HTTP proxy by adding an entry in the FGLPROFILE file.

See the SSL Proxy Tutorial for additional information.

Multiple Web Services in a Single DVM

You can define multiple Web Services in a single Genero DVM. When you start the Web Services engine, all registered Web Services are started.

See the Web Services Engine class that is part of the Web Services Extensions library (com).

Service Location Repository

You can remap the location of Genero Web Services using entries in the FGLPROFILE file, depending on the network configuration and the access rights management of the deployment site.

See Using Logical Names for Service Locations for additional information.

Serializing Genero Data Types

You can add optional attributes to the definition of data types. You can use these attributes to map the BDL data types in a Genero Web Services Client or Server application to their corresponding XML data types.

See XML Attributes for additional information, including a complete list of attributes that can be used.

WSHelper.42m library file

This file contains internal 4GL functions to handle SOAP requests and errors.

The file is provided in the **$FGLDIR/lib** directory of the Genero Web Services Extension package, and should be linked into every Genero Web Services Server or Client program.

Installation

Summary:

- Package Installation
- Platform-specific Notes

Package Installation

Starting with version 2.11.04, **Genero Web Services** (GWS) are provided as a bundle including GWS and BDL. The new package is named **FGLGWS**. After installation, the target directory reflects the same organization as a GWS installation on top of a BDL installation.

Prior to version 2.11.04, Genero Web Services are packaged as an add-on to the Genero Business Development Language. For versions prior to 2.11.04, you must install BDL first and then GWS:

- **The version of the Genero Business Development Language package must match (that is, the version number must be the same, to two decimal places: 2.00, 2.01, etc.).**
- **The Genero BDL package must be installed before installing the Genero Web Services Extension.** If the appropriate version of the Genero Business Development Language is not installed on your system, you MUST install it before installing the Genero Web Services Extension.

The **Genero Application Server** package is required to manage your Web Services in a deployment environment. Unless you are interested in testing deployment issues, it is not required for Web Services development.

Important: Starting with version 2.0, you no longer need to create a runner that includes the Genero Web Services Extension package. Now, your applications should include the following line at the top of each module to import the Genero Web Services Extension library named **com**:

```
import com
```

Platform-Specific Notes

IBM AIX

- The "IBM C++ Runtime Environment Components for AIX" must be installed in order to use "Genero Web Services Extension 2.0". See the IBM support center for more information about downloading the component.

  Note: If not installed, you will get the following error message:
  ```
  Could not load C extension library 'com'.
  Reason: A file or directory in the path name does not exist.
  ```

- Due to an IBM issue on 64Bits platforms, the openssl library is unable to open the system /dev/urandom device to generate a PRNG number.
  Therefore you must install the **Entropy Gathering Daemon (a.k.a EGD)** if you need security in your GWS application, and especially if you access a server in HTTPS.

Migration Notes

Summary:

Migrating GWS Server applications

Migrating GWS Server runners only

There is no need to create a special runner for Genero Web Services Extension 2.x. Instead, the GWS 2.x library is imported into your applications. If you want to migrate an existing GWS Server application to 2.x to avoid the need for a special runner, as well as to take advantage of any bug fixes, take the following steps:

1. Add the following statement at the top of any .4gl module where you have used GWS 1.3x functions:

```
import com
```

2. Then, compile and re-link your GWS Server application (.42r).

This imports the new GWS **com** library, and ensures that any GWS 1.3x functions that you have used will be compatible. Your existing Genero 1.3x Client applications, as well as third-party Client applications, will continue to work.

Migrating GWS Server runners and using new APIs

If you want to take advantage of the new features and simplify future migrations, you can migrate your GWS Server runner and also use the new GWS 2.x APIs. All the1.3x publishing functions for all the operations in your application must be replaced with 2.x publishing functions. Since this does not change the interface, all existing Genero 1.3x Client applications, as well as third-party Client applications, will continue to work.

Since 1.3x only supports RPC-Encoded style services, you must use the RPC style functions of the new 2.x APIs as the replacement functions, with **setInputEncoded** and

setOutputEncoded set to true. And, you cannot add XML attributes to the records used as Web Service function parameters.

To replace the fgl_ws_server_publishfunction() statement in an existing GWS Server application; for example:

```
       CALL fgl_ws_server_publishfunction(
          "EchoInteger",
          "http://tempuri.org/webservices/types/in", "echoInteger_in",
          "http://tempuri.org/webservices/types/out",
"echoInteger_out",
          "echoInteger")
```

1. Add the following statement at the top of each module:

   ```
        import com
   ```

2. Define variables for the WebService and WebOperation objects:

   ```
   DEFINE serv  com.WebService
   DEFINE op    com.WebOperation  -- Operation of a WebService
   ```

3. Create the GWS Server object:

   ```
   LET serv = com.WebService.CreateWebService("EchoInteger",
                             "http://tempuri.org/webservices")
   ```

4. Use the 2.x publishing functions for each operation:

   ```
   LET op = com.WebOperation.CreateRPCStyle("echoInteger",
               "EchoInteger",echoInteger_in,echoInteger_out)
   CALL op.setInputEncoded(true)
   CALL op.setOutputEncoded(true)
   CALL serv.publishOperation(op,NULL)
   ```

5. Compile and re-link your GWS Server application (.42r)

GWS 2.x also allows your Server application (.42r) to contain multiple services. If you would like 2.x and 1.3x GWS to co-exist in the same .42r executable, replace the existing publishing 1.3x functions as outlined above.

Enhance the GWS Server application to be WS-I compliant (recommended)

Warning: You must be able to change all the Client applications that access your migrated GWS Server.

If you use the Literal styles now available in GWS 2.x for your Web Service, your application will be WS-I compliant. However, the migration techniques shown above still use the RPC/Encoded style (Only RPC/Encoded was supported in GWS 1.3x.). If you can change all the client applications that access your migrated GWS Server, we recommend that you enhance the GWS Server application to be WS-I compliant.

1. Replace the publishing functions in the GWS Server application, as shown above, but omit the **setInputEncoded** and **setOutputEncoded** lines shown in the example; the resulting style will be Literal.

2. The enhanced GWS Server will have a new RPC/Literal WSDL that must be used to regenerate the client stub with the fglwsdl tool:

```
fglwsdl -o NewClientstub
http://localhost:8090/MyCalculator?WSDL
```

3. Compile that new client stub, and re-link it with the GWS Client application. This operation must be repeated for each Client application accessing that service.

4. Third party Client applications must be changed to use the new WSDL also.

Migrating GWS Client applications

Migration from version 1.3x to 2.10

If you use a Genero 2.10 runner for the GWS Client application, you **must**:

1. Regenerate the GWS Client stubs using the **-compability** option of the **fglwsdl** tool, so the function prototypes will be compatible:

```
fglwsdl -compatibility -o NewClientstub
http://localhost:8090/MyCalculator?WSDL
```

2. Compile the GWS Client stubs and re-link the Client application (.42r).

Migration from version 2.0x to 2.10

You **must** regenerate all client stubs into your application using the **fglwsdl** tool. This is **mandatoy** because the generated code is based on the low-level COM and XML APIs and is completly different from any previous versions; otherwise, you won't be able to execute the code.

com.WebServiceEngine options

In class com.WebServiceEngine, two options have been renamed and two options moved to a new class.

Renamed options

http_invoketimeout and tcp_connectiontimeout options have been respectively renamed into **readwritetimeout** and **connectiontimeout** as this is now available for either http or tcp protocol. The old option names remain for backward compatibility but it is strongly recommened to use the new option names.

Moved options

xml_ignoretimezone and **xml_usetypedefinition** options were part of the com.WebServiceEngine class. They are now moved to a new class **xml.Serializer** which gather functions on serialization.

Debugging

The Genero Web Services Extension gives you the ability to log the data your Web Service application is receiving from or sending to another application. The data is written to the standard error stream of the console; if needed, it can be redirected to a file. To turn on the debugging feature, set the **FGLWSDEBUG** environment variable before starting the application. The level of debugging depends on the value set for the **FGLWSDEBUG** variable.

You can set the **FGLWSDEBUG** environment variable using the values defined in the following table:

Value	Definition
0	No data displayed; debug turned off.
1	Display socket errors.
2	Display incoming and outgoing XML requests.
3	Display incoming and outgoing requests.

Debugging from Application Server

To debug a Web Service application managed by the Application Server, you have to modify the value of the **FGLWSDEBUG** environment variable in the **Application Server configuration file**. For more information, refer to the *Genero Application Server Manual* documentation.

Examples

Some basic examples are shipped with Genero Web Services Extension. You can find these examples in the **demo/WebServices** subdirectory of your installation directory (**FGLDIR**).

Using Logical Names for Service Locations

GWS release 2.00 provides a repository for Web Service locations using FGLPROFILE. To achieve maximum flexibility, you can map a logical reference used by your Web Services Client application to an actual URL. This is subject to the network configuration and access rights management of the deployment site.

Summary:

- FGLPROFILE entry
- Logical reference in the .4gl Client application
- Logical reference in the URL

FGLPROFILE entry

The following entry in the FGLPROFILE file maps the logical reference "myservice" to an actual URL:

```
ws.myservice.url = "http://www.MyServer.com/cgi-
bin/fglccgi.exe/ws/r/MyWebService"
```

Logical reference in the .4gl Client application

When you generate a Client stub from WSDL information using the tool **fglwsdl**, a global variable for the URL of the Web Service is contained in the **.inc** file. For example:

```
# Location of the SOAP server.
# You can reassign this value at run-time.
#
DEFINE Calculator_CalculatorPortTypeLocation STRING
```

You can assign a logical name to this global variable in your Web Services Client application:

```
LET Calculator_CalculatorPortTypeLocation = "myservice"
```

When the Client application accesses the Service, the actual location will be supplied by the entry in FGLPROFILE on the Client machine. This allows you to provide the same compiled .42r application to different customers. The entries in FGLPROFILE on each customer's machine would customize the Web Service location for that customer.

Logical reference in the URL

When you deploy a GWS Web Service with a GAS behind a Web Server, the service can be accessed by two different URLs. You can use a logical name in the URL, mapping the actual location of the Web Service in FGLPROFILE, depending on the location of the client machine. For example:

- For internal Clients: **http://zeus:6394/ws/r/myservice**
- For Clients using the Web: **http://www.myServer.com/...**

These two URLS could be mapped in the FGLPROFILE file on the Client machine, each specifying the location of the Service.

Tutorial: Writing a Client Application

The Genero Web Services Extension (GWS) allows a BDL program to access Web services found on the Internet. GWS supports the WSDL 1.1 specification of March 15, 2002. This example illustrates a client application that accesses the Add operation in the GWS Web Service **MyCalculator**. See: Tutorial: Writing a GWS Server Application for additional information about the Service.

Summary:

- Obtaining the WSDL information
- Calling the Web Service
- Setting a Time Period for a Response
- Handling Server errors
- Compiling the Client application

See also: The fglwsdl Tool

Obtaining the WSDL information

To access a remote Web service, you must get the WSDL information from the service provider. Sample services can be found through UDDI registries or on other sites such as XMethods (http://www.xmethods.net).

You can use the **fglwsdl** tool provided by the Genero Web Services Extension to obtain the necessary WSDL information. The following example obtains the WSDL information for the GWS Service MyCalculator created by the Server tutorial:

```
fglwsdl -o Example2Client http://localhost:8090/MyCalculator?WSDL
```

This generates the following files:

- **Example2Client.inc -** the globals file containing the definitions of the input and output records, and the prototypes of the operations.
- **Example2Client.4gl -** a module containing the definitions of the functions that can be used in your GWS client application to perform the requested Web Service operation, and the code that manages the Web Service request.

Note: The MyCalculator GWS Service must be running on the specified port in order to provide the WSDL information.

The following definitions were generated in the globals file, **Example2Client.inc**:

- **Input and Output records**

```
DEFINE Add RECORD ATTRIBUTE(XMLName="Add",
          XMLNamespace="http://tempuri.org/webservices")
        a INTEGER ATTRIBUTE(XMLName="a",XMLNamespace=""),
        b INTEGER ATTRIBUTE(XMLName="b",XMLNamespace="")
  END RECORD

DEFINE AddResponse RECORD ATTRIBUTE(XMLName="AddResponse",
           XMLNamespace="http://tempuri.org/webservices")
         r INTEGER ATTRIBUTE(XMLName="r",XMLNamespace="")
  END RECORD
```

Since BDL functions cannot have complex structures as parameters, the data types are defined as global or modular variables.

- Function prototypes for the Operations

  This globals file contains the prototype of two functions for the Add operation.

  The **Add** function uses input and output parameters, and returns the status and result. This function can only be used if the input and output parameters are not complex structures such as arrays or records. Using this function, developers do not access the global records directly.

  The **Add_g** function can be used with the global input and output records. Before calling this function, you must set the values in the variables of the global input record.

  ```
  Operation: Add
  #
  # FUNCTION: Add_g()
  # RETURNING: soapStatus
  # INPUT: GLOBAL Add
  # OUTPUT: GLOBAL AddResponse
  #
  # FUNCTION: Add(p_a, p_b)
  # RETURNING: soapStatus ,p_r
  ```

See Working with WSDL information for a more detailed explanation of the **fglwsdl** tool and its output, and the generated functions.

Calling the Web Service

Step 1: Import the GWS Extension library

The methods associated with creating and publishing a Web Service are contained in the classes that make up the Genero Web Services Extension Library (com). If you use any

of these methods in your client application, you must import the library. Since this example application sets the timeout period that the client will wait for the Service to respond, include the following line at the top of the module:

```
IMPORT com
```

Step 2: Specify the globals file

Use a GLOBALS statement to specify the generated globals file.

```
GLOBALS "Example2Client.inc"
```

Step 3: Write the MAIN program block

Provide values for the input and output messages of the operation, and call one of the generated functions. Since the input and output messages are simple integers, we can call the **Add** function.

```
MAIN
  DEFINE op1        INTEGER
  DEFINE op2        INTEGER
  DEFINE result     INTEGER
  DEFINE wsstatus   INTEGER
  LET op1 = 1
  LET op2 = 2
  CALL Add(op1, op2) RETURNING wsstatus, result
  IF wsstatus = 0 THEN
    DISPLAY "Result: ", result
  ELSE
   -- Use the global wsError record
   DISPLAY "Error: ", wsError.description
  END IF
END MAIN
```

Alternatively, we can use the global input and output records directly, calling the **Add_g** function:

```
MAIN
  DEFINE wsstatus INTEGER

  LET Add.a = 1
  LET Add.b = 2
  LET wsstatus = Add_g()
  IF wsstatus != 0 THEN
    -- Use the glocal wsError record
    DISPLAY "Error :", wsError.Description
  ELSE
    DISPLAY "Result: ", AddResponse.r
 END IF
END MAIN
```

These examples are very basic versions of the code. For complete examples, see the code samples provided with the package in **demo/WebServices**.

Setting a Time Period for the Response

To protect against remote server failure or unavailability, you can set a timeout value that indicates how long you are willing to wait for the server to respond to your request. Use the **SetOption()** method of the WebServiceEngine class to set the the "**http_invoketimeout**" option.

For example, to wait no more than 10 seconds:

```
CALL com.WebServiceEngine.SetOption( "http_invoketimeout", 10 )
```

A timeout value of **-1** means "wait forever". This is the default value.

Handling GWS Server Errors

When a GWS Service operation returns a status that is non-zero, you can get a more detailed error description from the global record **wsError**:

wsError.code: Short error message

wsError.codeNS: Namespace of the error code

wsError.description: Long error message

wsError.action: Internal "SOAP action"

Compiling the Client application

The library file **WSHelper.42m**, included in the $FGLDIR/lib directory of the Genero Web Services package, should be linked into every client or server program. Assuming the example client code shown above is in a module named **clientmain.4gl**, you can compile and link the client program as follows:

```
fglcomp clientmain.4gl Example2Client.4gl
fgllink -o myclient.42r clientmain.42m Example2Client.42m WSHelper.42m
```

Writing a Web Services Function

Writing Web Services in BDL is very easy with the **Genero Web Services Extension** package. You need only to create a classic BDL function, and to publish it as a Web Function (Web Services operation) using methods from the classes available in the com library. However, there are restrictions on the BDL function - neither input nor output parameters are allowed.

- Defining the Input Message
- Defining the Output Message
- Writing your BDL function
- Creating and publishing the Web Services operation

See also Tutorial: Writing a Server

Defining the Input Message

Input parameters in Genero Web Service operations are not allowed, but each Web Function can have one global variable or module variable that defines the input message of the function. This message must be a record in which each field represents one of the input parameters of the Web Function.

The name of each field corresponds to the name used in the SOAP request. These fields are filled with the contents of the SOAP request by the Web Services engine just before executing the corresponding BDL function.

Example:

```
DEFINE add_in RECORD
        a INTEGER,
        b INTEGER
        END RECORD
```

Note: Genero version 2.0 allows you to add optional attributes to the definition of data types. You can use attributes to map the BDL data types in a Genero application to their corresponding XML data types. See Attributes to Customize XML Mapping for additional information.

Defining the Output Message

Output parameters in Genero Web Functions are not allowed, but each Web Function can have one global variable or module variable that defines the output message of the

function. This message must be a record where each field represents one of the output parameters of the Web Function.

The name of each field corresponds to the name used in the SOAP request. These fields are retrieved from the Web Services engine immediately after executing the BDL function, and sent back to the client.

Example:

```
DEFINE add_out RECORD
          r INTEGER
          END RECORD
```

Note: GWS 2.0 allows you to add optional attributes to the definition of data types. You can use attributes to map the BDL data types in a Genero application to their corresponding XML data types. See Attributes to Customize XML Mapping for additional information.

Writing your BDL Function

A Web Function is a normal BDL function that uses the input and output records that you have defined.

Example:

```
FUNCTION add()
  LET add_out.r = add_in.a + add_in.b
END FUNCTION
```

Creating and publishing the Web Services operation

Methods are available in the Genero Web Services Extension library (**com**) to:

- Define the Web Service, by creating a WebService object
- Define the Web Services operation for your function, by creating a WebOperation object
- Publish the operation - associate it with the Web Service object that you defined.

The **com** library must be imported into each module of a Web Services Server application.

The following abbreviated example is from the Web Services Server tutorial:

```
IMPORT com
...
FUNCTION createservice()
  DEFINE serv  com.WebService    # A WebService
  DEFINE op    com.WebOperation  # Operation of a WebService
  --#Create WebService object
  LET serv = com.WebService.CreateWebService("MyCalculator",
                       "http://tempuri.org/webservices")
  --Create WebOperation object
  LET op = com.WebOperation.CreateRPCStyle("add", "Add", add_in,
add_out)
  --Publish the operation, associating it with the WebService object
  CALL serv.publishOperation(op,NULL)
...
END FUNCTION
```

See the Web Services Server tutorial and Choosing a Web Service Style for complete examples and explanations.

Choosing a Web Services Style

Genero Web Services Extension 2.0 allows you to create Web Services operations in the following styles:

- **RPC Style Service (RPC/Literal)** is generally used to execute a function, such as a service that returns a stock option.
- **Document Style Service** (**Doc/Literal**) is generally used for more sophisticated operations that exchange complex data structures, such as a service that sends an invoice to an application, or exchanges a Word document; this is the MS.Net default.

  Both RPC/Literal and Doc/Literal Styles are WS-I compliant (Web Services Interoperability organization).

- **RPC Style Service (RPC/Encoded)** is provided only for backwards compatibility with older versions of web services already published. Warning: This style is deprecated by the WS-I organization, and is not recommended, as most Web Service implementations won't support it in the future. See migration notes if you want your service to be WS-I compliant.

The style of service to be created is specified in the Genero application for the Web Service, using the following methods of the WebOperation class from from the Web Services Extension COM Library (**com**). The parameters are the same for both methods:

1. the name of the 4GL function that is executed to process the Web Service operation
2. the name you wish to assign to the Web Service operation
3. the input record defining the input parameters of the operation (or NULL if there is none)
4. the output record defining the output parameters of the operation (or NULL if there is none)

```
LET op = com.WebOperation.CreateRPCStyle("add","Add",add_in,add_out)
LET op =
com.WebOperation.CreateDOCStyle("checkInvoice","CheckInvoice",invoice_i
n,invoice_out)
```

Calling the appropriate function for the desired style is the only difference in your Genero code that creates the service. The remainder of the code that describes the service is the same, regardless of whether you want to create an RPC or Document style of service.

Note: Warning: Do not use the setInputEncoded() and setOutputEncoded() methods of the WebService class from the Web Services Extension COM Library (**com**), as they apply only to RPC/Encoded Style, which is not recommended.

If you add headers to your RPC Style service, choose the Literal serialization mechanism by setting the **encoded** parameter of the createHeader() method to FALSE. Example:

```
CALL serv.createHeader(var,FALSE)
```

GWS release 2.0 allows you to create RPC Style and Document Style operations in the same Web Service. However, we do not recommend this, as it is not WS-I compliant.

Tutorial: Writing a GWS Server application

This tutorial guides you through the steps to create a Server application for a Genero Web Service that can be accessed over the web by Client applications. A complete example is in the demo/WebServices subdirectory of the Genero installation directory.

You can write your Server application based on input/output records that you have defined. Or, you can use the tool **fglwsdl** to include third-party WSDL information in your Server application.

Summary:

- Including the Genero Web Services Extension library
- Example 1. Writing a GWS Server Application
  - Step 1: Define input and output records
  - Step 2: Write a BDL function for each Service operation
  - Step 3: Create the Service and Operations
  - Step 4: Register the Service
  - Step 5: Start the GWS Server and process requests
- Example 2: Writing a GWS Server using third-party WSDL (fglwsdl)
  - Step 1: Get the WSDL description and generate files
  - Step 2: Write a BDL function for your Service operation
  - Step 3: Create Service/Start Server and process requests as above
- Compiling GWS Server applications
- Testing the GWS Service in stand-alone mode
- Configuring the Genero Application Server for the GWS Application
- Making the GWS Service available

See also: Choosing a Web Services Style, The fglwsdl Tool, Deployment

---

Including the Genero Web Services Extension library

The methods associated with creating and publishing a Web Service are contained in the classes that make up the Genero Web Services Extension Library (com). Include the following line at the top of each module of your GWS Server application to import the library:

```
IMPORT com
```

Example 1: Writing the entire Server Application

You can define a Web Service in your application and write definitions for the input and output records that will be used by the Service. This example illustrates a Service that has one operation, **Add**, to provide the sum of two numbers.

Step 1: Define Input and Output Records

Based on the desired functionality of the operations that you plan for the Service, define the input and output records for each operation. BDL functions that are written to implement a Web Service operation cannot have input parameters or return values. Instead, each function's input and output message must be defined as a global or module RECORD.

The Input message

The fields of the global or module record represent each of the input parameters of the Web Function. The name of each field in the record corresponds to the name used in the SOAP request. These fields are filled with the contents of the SOAP request by the Web Services engine just before executing the corresponding BDL function.

The Output message

The fields of the global or module record represent each of the output parameters of the Web Function. The name of each field in the record corresponds to the name used in the SOAP request. These fields are retrieved from the Web Services engine immediately after executing the BDL function, and sent back to the client.

Your GWS service has one planned operation that adds two integers and returns the result. The input and output records are defined as follows:

```
GLOBALS
  DEFINE
    add_in RECORD    -- input record
      a INTEGER,
      b INTEGER
    END RECORD,
    add_out RECORD   -- output record
       r INTEGER
    END RECORD
END GLOBALS
```

Step 2: Write a BDL function for each Service operation

You will need to write a function to implement each operation, using the input and output global records.

To implement your **Add** operation:

```
#User Public Functions
FUNCTION add()
 LET add_out.r = add_in.a + add_in.b
END FUNCTION
```

Step 3: Create the Service and Operations

The Genero Web Services Extension library (**com**) provides classes and methods that allow you to use Genero BDL to configure a Web Service and its operations.

- **WebService** - this is a container for web operations.
- **WebOperation** - describes the operation.

Define variables for the WebService and WebOperation objects

```
FUNCTION createservice()
DEFINE serv  com.WebService    # A WebService
DEFINE op    com.WebOperation  # Operation of a WebService
```

Choose a Namespace

XML uses namespaces to group the element and attribute definitions, and to avoid conflicting names. In practice, a namespace must be a unique identifier (URI: Uniform Resource Identifier). If you do not know the unique identifier to use, your company's Web site domain name is guaranteed to be unique (such as "http://www.4js.com"); then, append any string.

Examples of valid namespaces for the Four J's Development Tools company:

- "**http://www.4js.com/MyServices**"
- **"http://www.4js.com/any_string"**

Another option (for testing only) is to use the temporary namespace "**http://tempuri.org/**".

Create the WebService object

Call the constructor method of the WebService class. The parameters are:

1. Service name
2. Valid namespace

This example uses the temporary namespace and creates a Service named "MyCalculator".

```
LET serv = com.WebService.CreateWebService("MyCalculator",
                          "http://tempuri.org/webservices")
```

Create the WebOperation object

A WebService object can have multiple operations. The operations can be created in RPC style or Document style by calling the corresponding constructor method of the WebOperation class. The parameters are:

1. the name of the 4GL function that is executed to process the XML operation
2. the name you wish to assign to the XML operation
3. the input record defining the input parameters of the operation (or NULL if there is none)
4. the output record defining the output parameters of the operation (or NULL if there is none)

To create the operation for the previously defined **add** function in RPC style:

```
LET op = com.WebOperation.CreateRPCStyle("add", "Add", add_in, add_out)
```

To create the operation for the previously defined **add** function in Document style:

```
LET op = com.WebOperation.CreateDOCStyle("add", "Add", add_in, add_out)
```

Mixing RPC style and Document style operations in the same service is not recommended, as it is not WS-I compliant. See Web Services Styles for additional information about styles.

The rest of the code in your application is the same, regardless of the Web Services style that you have chosen.

Publish the operation

Once an operation is defined, it must be associated with its corresponding WebService (the operation must be published). The publishOperation method of the WebService object has the following parameters:

- the WebOperation to be published
- a string to identify the operation if several operations have the same name; if this is NULL, the default value is an empty string

For example, to publish the **Add** operation of the **Calculator** service, which was defined as **op**:

```
CALL serv.publishOperation(op,NULL)
```

Step 4: Register the Service

Once the Service and operations are defined and the operations are published, the WebService and WebOperation objects have completed their work. Registering a service puts the Genero DVM in charge of the execution of all the operations of that service - dispatching the incoming message to the right service, returning the correct output, etc. The same service may be registered at different locations on the Web.

The WebServiceEngine is a global built-in object that manages the Server part of the Genero DVM. Use the **RegisterService** class method of the WebServiceEngine class. The parameter is:

1. The name of the WebService object

To register the Calculator service example previous created:

```
CALL com.WebServiceEngine.RegisterService(serv)
END FUNCTION
```

Note: If you wanted to create a single GWS Server DVM containing multiple Web Services, you could define additional input and output records and repeat steps 2 through 6 for each Web Service. In Step 5, a GWS Server DVM is started, containing as many Web Services as you have defined. See Deployment for additional discussion of GWS Services and GWS Servers.

Step 5: Start the GWS Server and process requests

Once you have registered the Web Service(s), you are ready to start the GWS Server and process the incoming SOAP requests. The GWS Server is located on the same physical machine where the application is being executed (where **fglrun** is executing).

This is the MAIN program block of your application.

- Define a variable for status
- Define a variable to hold the returned status of the request:

```
MAIN
DEFINE ret INTEGER
```

Call the function that you created above, which defined and registered the service and its operations:

```
CALL createservice()
```

Start the GWS Server

Use the **Start** class method of the WebServiceEngine class to start the server.

```
CALL com.WebServiceEngine.Start()
```

Process the requests

The following example uses the **Process Services** method of the WebServiceEngine class to process each incoming request, returning an integer representing the status. The parameter specifies the timeout period in number of seconds for which the method should wait to process a service; the value -1 specifies an infinite waiting time.

```
WHILE TRUE
  # Process each incoming requests (infinite loop)
  LET ret = com.WebServiceEngine.ProcessServices(-1)
  CASE ret
   WHEN 0
     DISPLAY "Request processed."
   WHEN -1
     DISPLAY "Timeout reached."
   WHEN -2
     DISPLAY "Disconnected from application server."
     EXIT PROGRAM
   WHEN -3
     DISPLAY "Client Connection lost."
   WHEN -4
     DISPLAY "Server interrupted with Ctrl-C."
   WHEN -10
     DISPLAY "Internal server error."
  END CASE
  IF int_flag<>0 THEN
    LET int_flag=0
    EXIT WHILE
  END IF
END WHILE
DISPLAY "Server stopped"
END MAIN
```

Note: The **Genero Application Server (GAS)** is required to manage your application in a production environment. For deployment, the GWS Server application must be added to the GAS configuration; see "Adding Applications" in the GAS manual. For testing purposes only, the GWS Server can be started in stand-alone mode.

Example 2: Writing a GWS Server using Third-Party WSDL (the fglwsdl tool)

To write a Web Service that is compatible with the specification of the input and output records defined by a third-party (for example, a vendor of manufacturing software, or a WSDL specialist in your company) you can use the **fglwsdl** tool to obtain the WSDL information and generate a part of the Server application, using the steps below. See The fglwsdl Tool for a complete description of the tool and its use.

Step 1: Get the WSDL description and generate files

This tutorial uses fglwsdl and the Calculator Service defined in Example1 to obtain the WSDL information and generate two corresponding BDL files:

- the globals file, containing declarations of global variables that can be used as input or output to functions accessing the Web Service operations.
- a .4gl file containing a function that creates the service described in the WSDL, publishes the operations of the service, and registers the service.

```
fglwsdl -s -o example1 http://localhost:8090/MyCalculator?WSDL
```

Note: the MyCalculator GWS Service created in Example1 must be running in order to obtain the WSDL information.

The generated globals file

The globals file **example1Service.inc** provides the definition of the global input and output records as described in the Step 1 of the Example 1 GWS Server program. The names of the input and output records have been assigned by **fglwsdl**, in accordance with the Style of the Web Service MyCalculator (created as RPCStyle in the Example1 program).Do not modify this file.

Input and output records:

```
# VARIABLE : Add
DEFINE Add RECORD ATTRIBUTE(XMLName="Add",
                            XMLNamespace="http://tempuri.org/webservices")
         a INTEGER ATTRIBUTE(XMLName="a",XMLNamespace=""),
         b INTEGER ATTRIBUTE(XMLName="b",XMLNamespace="")
       END RECORD
# VARIABLE : AddResponse
DEFINE AddResponse RECORD ATTRIBUTE(XMLName="AddResponse",

XMLNamespace="http://tempuri.org/webservices")
         r INTEGER ATTRIBUTE(XMLName="r",XMLNamespace="")
       END RECORD
```

The generated .4gl file

The **example1Service.4gl** file contains a single function that creates the service, publishes the operation, and registers the Service. The Web Service Style that is created is determined by the style specified in the WSDL information. The functions in this file accomplish the same tasks as Step 3 and Step 4 of Example 1. Do not modify this file.

```
# example1Service.4gl - generated file containing the function
#                           Createexample1Service
IMPORT com
GLOBALS "example1Service.inc"
```

```
# FUNCTION Createexample1Service
#   RETURNING soapstatus
FUNCTION Createexample1Service()
  DEFINE service      com.WebService
  DEFINE operation    com.WebOperation
  # Set ERROR handler
  WHENEVER ANY ERROR GOTO error
# Create Web Service
  LET service =
com.WebService.CreateWebService("MyCalculator","http://tempuri.org/webs
ervices")
  # Operation: Add
  # Publish Operation : Add
  LET operation =
com.WebOperation.CreateRPCStyle("Add","Add",Add,AddResponse)
  CALL service.publishOperation(operation,"")
  # Register Service
  CALL com.WebServiceEngine.RegisterService(service)
  RETURN 0
  # ERROR handler
  LABEL error:
  RETURN STATUS
  # Unset ERROR handler
  WHENEVER ANY ERROR STOP
END FUNCTION
```

Step 2: Write a BDL function for your Service operation

Using the information from these generated files, the Add operation from Example1 is re-written to have different functionality but to still be compatible with the WSDL description of the operation. This step accomplishes the same thing as Step 2 in Example 1. In this version of the add operation, the sum of the two numbers in the input record is increased by 100.

```
# my_function.4gl  -- file containing the function definition
IMPORT com         -- import the Web Services Extension library
GLOBALS "example1Service.inc"   -- use the generated globals file
#User Public Functions
FUNCTION add()                                  -- new version of the add
function
 LET AddResponse.r = (Add.a + Add.b)+ 100       -- the global input and output
records are used
END FUNCTION
```

Step 3: Create Service/Start Server and process requests

Create your own Main module that calls the function from the generated .4gl file to create the service, and then starts the GWS Server and manages requests as in Step 5 of Example 1.

```
#example2main.4gl file -- contains the MAIN program block
IMPORT com
GLOBALS "example1Service.inc"
MAIN
  DEFINE create_status INTEGER

  DEFER INTERRUPT

  CALL Createexample1Service()  -- call the function generated in
                                -- example1Service.4gl
        RETURNING create_status
  IF create_status <> 0 THEN
     DISPLAY "error"
  ELSE
    # Start the server and manage requests
     CALL ManageService()
  END IF

END MAIN

FUNCTION ManageService()
  DEFINE ret INTEGER
  CALL com.WebServiceEngine.start()
  WHILE TRUE
  # continue as in Step 5 of Example 1
  ...
END FUNCTION
```

Compiling GWS Server Applications

The library file **WSHelper.42m**, included in the $FGLDIR/lib directory of the Genero Web Services package, should be linked into every GWS Server application.

If your application uses the **fglwsdl** tool to generate information, link the .4gl generated file into the application.

Examples:

- Compiling the Example1 program

```
fglcomp example1.4gl
fgllink -o example1.42r example1.42m WSHelper.42m
```

- Compiling the Example2 program

```
fglcomp example2main.4gl my_function.4gl
example1Service.4gl
fgllink - o example2.42r example2main.42m my function.42m
                      example1Service.42m WSHelper.42m
```

Testing the GWS Service in stand-alone mode

For testing and development purposes only, the GWS Server application can be executed directly, without using the Genero Application Server (GAS).

1. Use the Genero fglrun command to execute the GWS Server application, which must reside on the same machine:

   ```
   fglrun <gws application>
   ```

   This will start the GWS Server on the port specified by the **FGLAPPSERVER** environment variable. If this environment variable is not set for the user, port number 80 is used. For example, if **FGLAPPSERVER** is set to 8090, the server will be started on that port.

   Note: The user must not set the FGLAPPSERVER variable in production environments, since the port number is selected by the Genero Application Server (GAS).

2. Obtain the WSDL information for your Service and write a test Client application. If the GWS Server in step 1 was started on your local machine, for example, the command to get the WSDL information would be:

   ```
   fglwsdl -o <test-client>
   http://localhost:8090/<service-name>?WSDL
   ```

Configuring the Genero Application Server for the GWS Application

The final step is to configure the Genero Application Server (GAS) to handle the GWS application. In a production environment, Genero Web Services becomes a part of a global application architecture handled by the application server of the **GAS** package**.** See Deployment, as well as *Adding Applications* in the GAS manual.

Making the GWS Service available

Once you compile and deploy your GWS Server application (see Deployment), it can be used by others to obtain the WSDL information and write a client application that accesses your Genero Web Service. See Tutorial: Writing a Client Application.

Your company can provide the location of the GWS Server to potential users of your Web Service in various ways; for example:

- Provide the location on a company web site
- Register the Web Service with UDDI (Universal Description, Discovery, and Integration) - the XML-based registry providing Internet listings for companies worldwide
- Communicate directly with your potential users

Deployment

In a production environment, Genero Web Services becomes a part of a global application architecture handled by the **Genero Application Server (GAS)**. The GWS DVMs are managed by the GAS.

This architecture takes care of:

- Security issues
- Scalability
  - Load management
  - Balancing of the SOAP requests amongst the available virtual machines
- Runtime monitoring

For deployment, the GWS Server application must be added to the GAS configuration. See *Adding Applications* in the GAS manual.

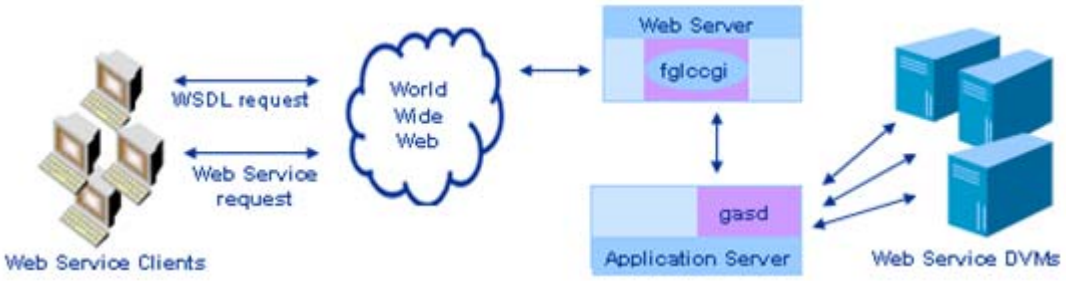
You can:

- **Create a GWS Server application that contains a single Web Service**. This application could be deployed on various physical machines. A Genero Web Services DVM is created on each machine where the GWS Server application is executed, to manage the requests for a service. A single GAS can communication with multiple GWS DVMs - the GAS daemon (**gasd**) handles load balancing among the GWS DVMs. Or each GWS DVM can have its own **gasd**.
- **Create a GWS Server application that contains multiple Web Services**. The GWS DVM would manage the client requests, dispatching the request to the appropriate service. Once again, there could be one or many GAS daemons communicating with the GWS DVMs.

Note: A Web Service Server must be stateless; several instances of the same Service can be created to support load balancing.

Deployment example

The basic deployment strategy can be implemented in varying permutations, depending on your business needs and the volume of requests.



- Using the World Wide Web, a Web Service client requests WSDL information for a particular Web Service from the Web Server.
- The Web Service client uses this information to make a Web Service request from the Web Server.
- The Web server spawns and communicates with the client CGI Connector, fglccgi. The Connector configuration is specified in the file connector.xcf.
- The Connector handles communication with the Genero Application Server, **gasd**. The **gasd** configuration is set in the file as.xcf (default) or a user-specified configuration file. The gasd must be started and listening for requests from the Connector.
- The **gasd** communicates with a Web Service DVM to fulfill the Web Service request.
- Communication is bi-directional, returning the requested information to the Web Service client.

Example

Accessing a Genero Web Service over the internet:

```
http://server_host_name/cgi-bin/fglccgi/ws/r/app_id/service_name
```

Encryption and Authentication

Summary:

- Secured communications
- Certificates
- Certificate Authorities
- Certificates and private keys storage

Secured communications

Secured communications are important. If an application wants to send or receive messages from a financial, business, or personnel application on the web, it must be able to authenticate the origin of the message, ensure that no malicious application has altered the original message, and ensure that no third party application can intercept the message.

Suppose that a person named Georges wants to send a message to his bank to transfer some money on the Internet. In this scenario, he faces the following concerns:

1. The **privacy** of the message, since it includes his account number and the transfer amount.
2. The **integrity** of the message, since someone might try to modify the original message or substitute a different message in order to transfer the money to another account.
3. The **authentication** of the message, since the bank must ensure that the message was sent from the right person.

Message privacy

To keep a message private, use a cryptographic algorithm - a technique that transforms a message into an encrypted form unreadable except by those it is intended for. Once it is in this form, the message may only be interpreted through the use of a secret key. There are two kinds of cryptography algorithms: *symmetric* and *asymmetric*.

Symmetric means the sender and the receiver of a message have to share the same key used to encrypt a clear message into an encrypted form, and then to decrypt it back into the original message. If that key is kept secret, nobody other than the sender and the receiver can read the message. However, the task of choosing a private key before communicating can be problematic.

Asymmetric means that there are two different keys working as a key-pair. One key is used to encrypt a message, and the second one is used to decrypt the encrypted message back into its original form. This solves the problem of key sharing in the symmetric

cryptography algorithm, and makes it possible to receive secure messages, simply by publishing the key used to encrypt messages (**the public key**), and keeping secret the key used to decrypt messages (**the private key**). Anyone can encrypt a message using the public key, but only the owner of the private key can read it.

The use of an asymmetric key-pair (public and private key), allows Georges to send private messages to his bank, simply by using the bank's public key to encrypt a message. Then, only the owner of the corresponding private key (the bank in this scenario) is able to read it.

Message integrity

To guarantee the integrity of a message, send a concise summary of the original message. The receiver of the message can create its own summary and compare it to the sender's summary. If they are similar, the message is considered intact, meaning that no third party has modified the original message.

Such a summary is called a **message digest** and is based on **hash** algorithms that produce a fixed-length representation of variable-length messages. Notice that message digests are designed to make it very difficult (if not impossible) to determine the original message from a summary.

However, the message digest must be sent to the receiver in a secure way to assure the message integrity. This is achieved with a digital signature authenticating the sender and containing the sender's message digest. (See Message authentication below.)

The use of message digests allows Georges' bank to verify that no one has modified the original message he sent.

Message authentication

To authenticate a message, add a digital signature to that message.

A **digital signature** is another message, created by encrypting the message digest, along with some other information, with the sender's private key. Anyone with the corresponding public key can decrypt the digital signature. If an application is able to decrypt it, it means the owner of the private key was able to encrypt it, proving that the message comes from this sender and not from someone else.

Once the sender has been authenticated, the receiver can compare the message digest integrated into the digital signature to the one it created from the message it receives, in order to check the message integrity.

The use of digital signatures allows Georges' bank to verify that the message really comes from him.

Certificates

Now that Georges is able to send a secured message to his bank, there is still a problem. How can Georges be sure that the server he is connected to is really the bank's server and not a malicious server?

Georges must be sure that the public key he is using to encrypt his message corresponds to the bank's private key. Similarly, the bank needs to verify that the message signature it receives corresponds to Georges' signature.

To identify a remote peer, use a **certificate -** a kind of digital identity card that associates the public key with a unique digital thumbprint identifying an individual, a server, or any other entity known as the **subject**. It also includes the identification and signature of the Certificate Authority that issued the certificate, and the period of time during which the certificate is valid. It may have additional information (or extensions) as well as administrative information for the Certificate Authority's use, such as a serial number.

A standard X.509 certificate contains the following standard fields:

- Certificate version
- Serial number of the certificate
- The distinguished name of the certificate issuer
- The distinguished name of the certificate owner
- The validity period of the certificate
- The public key
- The digital signature of the issuer
- Signature algorithm used
- Zero or more certificate extensions

Notes:

1. An example of a distinguished name is: **CN=Georges,E=georges@4js.com,OU=Sales,O=Four J's development tools,C=FR,S=France**
2. The CN (Common Name) of the distinguished name of the certificate owner corresponds to the certificate subject, and identifies the owner of that certificate.

Certificate Authorities

Each time Georges sends a message to his bank, he will present his own certificate to the bank, and will get the bank's certificate back. But as every one can create a certificate in the name of Georges, a higher authority that confirms the validity of a certificate is necessary. The bank must be sure it is Georges' certificate, and that no one else has taken his identity. Similarly, Georges needs an authority that confirms that the certificate coming from the server is really the bank's certificate.

The solution to validating a certificate is to sign it with a trusted certificate called **certificate authority**. This is a certificate in which an application creates total confidence concerning the validity of the certificates it has signed. Before signing a certificate, a certificate authority must proceed with a strict identification of the owner of that certificate.

Note: The private key associated to a Certificate Authority must be managed with care, as it is the entity in charge of the validity of all other certificates it has signed.

There are several companies (such as *VeriSign*, *GlobalSign* or *RSA Security*) that have established themselves as certificate authorities and provide the following services over the Internet:

- Verifying certificate requests
- Processing certificate requests
- Issuing and managing certificates

Note: It is also possible to create your own Certificate Authority, but it is up to you to manage it securely.

Root Certificate Authority

A Certificate Authority signed by itself is called a Root Certificate Authority, meaning that the certificate issuer is the same as the certificate subject. Most of the time, such a certificate belongs to a company established as a Certificate Authority, and is used to sign certificate requests coming from different companies that want their own Certificate Authority. If a client certificate is signed by a Certificate Authority previously signed by a Root Certificate Authority, the client certificate can be validated by the Root Certificate Authority even if the Certificate Authority is not present.

For example, if a company wants to buy a Certificate Authority from VeriSign, VeriSign signs that Certificate Authority with its own Root Certificate Authority. The company can then create certificates with the Certificate Authority provided by VeriSign and connect to secure servers without providing them their own Certificate Authority. The secure server, of course, has to know the VeriSign Root Certificate Authority.

Certificate chains

A certificate authority may issue a certificate for another certificate authority. This means that when an application wants to examine the certificate of the issuer, it must check all parent certificates of that issuer until it reaches one it which it has confidence.

The certificate chain corresponds to the number of parent certificate authorities allowed to validate a certificate.

Certificate Authority List

A Certificate Authority List is a list of all certificate authorities considered as trusted by one application, classified by order of importance. Each of these certificates allows the authentication of a certificate presented to that application from a remote peer.

Note: With most applications, the Certificate Authority List is a concatenated file of all certificate authorities.

Certificates and private keys storage

The entire concept of security is based on the publication of the public key, and the privacy of the associated private key. For maximum security, it is critical to restrict the access of the private key to the owner of the certificate and associated private key.

Note: Some companies provide systems to manage certificates and private keys in complete security.

Unix systems

As the Unix system is already able to restrict the access of a file to only one person, simply restrict access to the private key to the owner of that key to achieve a good level of security. This provides enough security to allow a Genero Web Services client to perform secured communications in the name of the certificate and private key owner, because access to the private key file is granted only if the correct user has logged in.

Windows systems

The Windows system doesn't provide a reliable and sufficiently strong file access rights policy to secure a file. However, Windows has an integrated **key store** system to manage certificates and private keys. It allows the registration and the storage of X.509 certificate authorities, as well as personal X.509 certificates and their associated private keys

accessible only if the correct user has logged in. It is recommended that you store the certificate and associated private key in the Windows key store instead of in files on the disk. For information on how to import a certificate and its associated private key file into the Windows key store, see Importing a certificate and its private key into the Windows key store.

Certificates in Practice

Summary:

- OpenSSL tool
- Creating a Root Certificate Authority
- Creating a Certificate Authority
- Creating a Certificate
- Creating a Certificate Authority list
- Importing a certificate and its private key into the Windows key store
- Importing a certificate authority into the Windows key store

---

OpenSSL tool

The **openssl** command line tool creates certificates for the configuration of secured communications. It requires a configuration file with the default parameters such as the key size or the private key name. **OpenSSL** is provided with a default configuration file called **openssl.cnf**.

Note: openssl looks for the **openssl.cnf** file in the directory where it is executed; it stops if the file is not present. To use the **openssl** tool from any directory, set the **OPENSSL_CONF** environment variable to specify the location of the configuration file.

For information on how the **openssl** tool works, refer to the **openssl** documentation at http://www.openssl.org/docs/apps/openssl.html.

Creating a Root Certificate Authority

- Create a CSR (Certificate Signing Request):
  ```
  $ openssl req -new -out MyRootCA.csr
  ```

  Note: This creates a **privkey.pem** file containing the RSA private key of that certificate and protected by a password.
- Remove the password of the private key (Optional):
  ```
  $ openssl rsa -in privkey.pem -out MyRootCA.pem
  ```

  Note: Removing the password of a certificate authority's private key is not recommended.
- Create a self-signed certificate from the Certificate Signing Request for a validity period of 365 days:

```
$ openssl x509 -trustout -in MyRootCA.csr -out MyRootCA.crt -req
-signkey MyRootCA.pem -days 365
```

Notes:
If you want an official Root Certificate Authority, you must send the CSR file to one of the self-established Certificate Authority companies on the Internet (instead of creating it with **openssl**).
Creation of a X509 certificate requires a serial number provided in OpenSSL configuration file; if not set, you must specify it with option -set_serial.

Creating a Certificate Authority

- Create a CSR (Certificate Signing Request):
  ```
  $ openssl req -new -out MyCA.csr
  ```

  Note: This creates a **privkey.pem** file containing to the RSA private key of that certificate and protected by a password.
- Remove the private key password (Optional):
  ```
  $ openssl rsa -in privkey.pem -out MyCA.pem
  ```

  Note: Removing the password of a certificate authority's private key is not recommended.
- Create a certificate from the Certificate Signing Request and trusted by the Root Certificate Authority:
  ```
  $ openssl x509 -in MyCA.csr -out MyCA.crt -req -signkey MyCA.pem
  -CA MyRootCA.crt -CAkey MyRootCA.pem
  ```

  Notes:
  If you want an official Certificate Authority, you must send the CSR file to one of the self-established Certificate Authority companies on the Internet (instead of creating it with **openssl**).
  Creation of a X509 certificate requires a serial number provided in OpenSSL configuration file; if not set, you must specify it with the following option: **-set_serial**.

Creating a Certificate

- Create a CSR (Certificate Signing Request):
  ```
  $ openssl req -new -out MyCert.csr
  ```

Note: This command creates a **privkey.pem** file containing the RSA private key of that certificate and protected by a password.

- Remove the private key password (Optional):
```
$ openssl rsa -in privkey.pem -out MyCert.pem
```
- Create a certificate from the Certificate Signing Request and trusted by the Certificate Authority:
```
$ openssl x509 -in MyCert.csr -out MyCert.crt -req -signkey
MyCert.pem -CA MyCA.crt -CAkey MyCA.pem
```

Notes:
If you want an official Certificate, you must send the CSR file to one of the self-established Certificate Authority companies on the Internet (instead of creating it with **openssl**).
Creation of a X509 certificate requires a serial number provided in OpenSSL configuration file, but if not set, you must specify it with option -set_serial.

Creating a Certificate Authority list

- Concatenate all certificate authorities by order of importance, listing the most important first:
```
$ openssl x509 -in MyCA1.crt -text >> CAList.pem
$ openssl x509 -in MyCA2.crt -text >> CAList.pem
$ openssl x509 -in MyCA3.crt -text >> CAList.pem
```

Importing a certificate and its private key into the Windows key store

- Create a certificate as described above.
- Create a specific PKCS12 file containing the certificate and its private key in one file:
```
$ openssl pkcs12 -export -inkey MyCert.pem -in MyCert.crt -out
MyCert.p12
```

Note: The .p12 generated file is protected by a password and can then be transported without any risk.
- On a Windows system, open this .p12 file and follow the instructions provided.

Note: If you select strong verification during the importation process, a pop-up displays each time an application accesses the private key asking the user whether the application is allowed to use it.

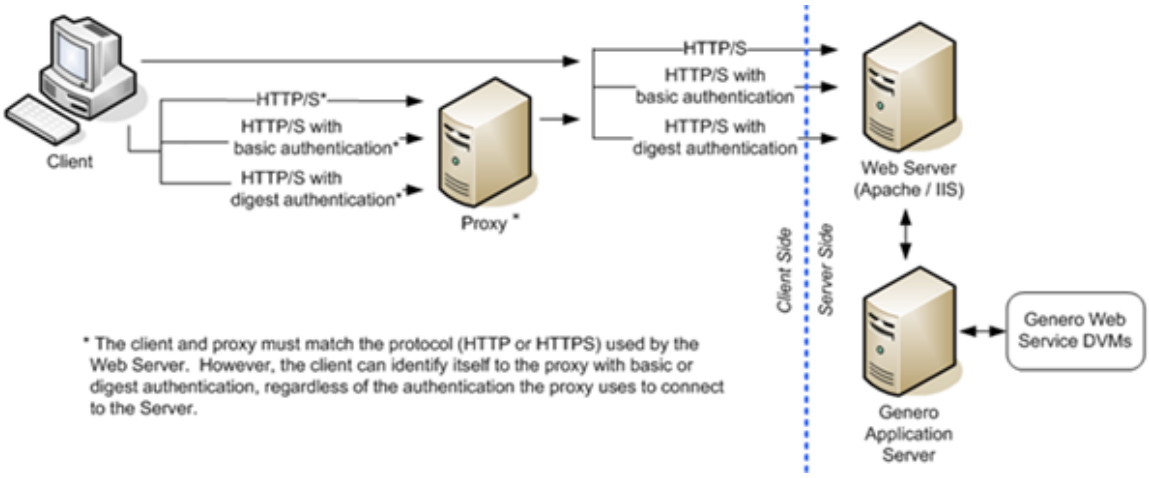
Importing a certificate authority into the Windows key store

- Create a certificate authority as described above.
- Open the .crt certificate file
- Click **Install Certificate** and follow the instructions provided.

  Note: Windows automatically places the certificate in the certificate authority list of the key store.

Accessing Secured Services

As security and authentication are more and more important, GWS provides various communications options for a client to connect to a Web Service:



Communication options

- **HTTP** : Client connects to a Web Server (or a Web Service) using HTTP as the communication protocol. (**No security, No authentication**)
- **HTTP with Basic Authentication** : Client connects to a Web Server using HTTP as the communication protocol, but a valid login and password are required from the Web Server to grant access to the Web Service. (**No security, Weak Authentication).** The login and password are sent in clear text on the communication layer.
- **HTTP with Digest Authentication**: Client connects to the Web Server using HTTP as the communication protocol, but a valid login and password are required from the Web Server to grant access to the Web Service. (**No security, Authentication).** The login and password are encoded using a digest algorithm, requiring additional information from the Web Server. This means that the first connection will always fail, but it is necessary in order to return Web Server additional information back to the client.

- **HTTPS**: Client connects to a Web Server using HTTPS as the communication protocol. (**Security, No authentication**). The communication channel is encrypted by SSL.
- **HTTPS with Basic Authentication**: Client connects to a Web Server using HTTPS as the communication protocol, but a valid login and password are required from the Web Server to grant access to the Web Service. (**Security, Weak Authentication**). The login and password are sent in clear text on the communication layer, but the communication channel is encrypted by SSL.
- **HTTPS with Digest Authentication**: Client connects to the Web Server using HTTPS as the communication protocol, but a valid login and password are

required from the Web Server to grant access to the Web Service. (**Security, Authentication)**. The login and password are encoded using a digest algorithm, requiring additional information from the Web Server. This means that the first connection will always fail, but it is necessary in order to return Web Server additional information back to the client. The communication channel is encrypted by SSL.

To improve communication speed with the cache mechanism, or to restrict internet access to specific clients, GWS allows a client to connect via proxies. The proxy is in charge of dispatching the client request to the server, and uses the same protocol as that used by the server. So, when a client connects via a proxy to access a HTTP server, the configuration of the HTTP proxy is used, and when the client communicates in HTTPS, the HTTPS proxy configuration is used.

- **HTTP proxy**: Client connects via a proxy using HTTP as the communication protocol.
- **HTTP proxy with Basic Authentication**: Client connects via a proxy using HTTP as the communication protocol, but a valid login and password are required from the proxy to dispatch the request to the Web Service. The login and password are sent in clear text on the communication layer between client and proxy.
- **HTTP proxy with Digest Authentication**: Client connects via a proxy using HTTP as the communication protocol, but a valid login and password are required from the proxy to dispatch the request to the Web Service. The login and password are encoded using a digest algorithm, requiring additional information from the proxy. This means that the first connection will always fail, but it is necessary in order to return proxy additional information back to the client.

- **HTTPS proxy**: Client connects via a proxy using HTTPS as the communication protocol. The communication channel is encrypted by SSL.
- **HTTPS proxy with Basic Authentication**: Client connects via a proxy using HTTPS as the communication protocol, but a valid login and password are required from the proxy to dispatch the request to the Web Service. The login and password are sent in clear text on the communication layer between client and proxy, but the communication channel is encrypted by SSL.
- **HTTPS proxy with Digest Authentication**: Client connects via a proxy using HTTPS as the communication protocol, but a valid login and password are required from the proxy to dispatch the request to the Web Service. The login and password are encoded using a digest algorithm, requiring additional information from the proxy. This means that the first connection will always fail, but it is necessary in order to return proxy additional information back to the client. The communication channel between client and proxy is encrypted by SSL.

FGLPROFILE password encryption

For security reasons, it is recommended that you avoid storing clear passwords in a file. The Genero Web Services Extension enables the password encryption of a HTTP Authenticate entry in the FGLPROFILE file. The encrypted password is decrypted by the Genero Web Services engine when required.

Summary:

- FglPass tool
- Encrypting a HTTP Authenticate password
- Encrypting a HTTP Authenticate password using a certificate in the Windows key store

---

FglPass tool

The Genero Web Services package provides a command line tool called **fglpass**. The **fglpass** tool can encrypt a password from a X.509 certificate. The encrypted password is displayed on the console in a Base64 form, composed only of alphanumeric characters, and therefore easily usable in any text file.

Notes:

1. The **fglpass** tool can also decrypt a password it previously encrypted with a certificate; however the private key associated to that certificate must be accessible.
   (Type fglpass -h for more details)

Encrypting a HTTP Authenticate password

- Find the HTTP Authenticate entry with the password you want to encrypt:

```
authenticate.myentry.login    = "fourjs"
authenticate.myentry.password = "mypassword"
```

- Add the certificate and its private key in the FGLPROFILE file as follows:

```
security.mykey.certificate = "MyCertificate.crt"
security.mykey.privatekey  = "MyPrivateKey.pem"
```

- Encrypt the password with **fglpass**:

```
$ fglpass -c MyCertificate.crt
Enter password :mypassword
```

The **fglpass** output looks like the following:

```
BASE64 BEGIN
dBy3E5JCVxuoxsR+aOBVfp1j0SwQPt+hdjpMKriWvO2xMd5rFnFEwv+sPPd4w/onW
viG0M5mqubBeS7QUlt/ZK0D1aO9/R5RVa5wylQu
//6vxfyd8NG/SFJmlVH63kuyXfiVfq6bHo5+nlQZpVjSHfF2msET3S9HTpZUt4Nbl
P4=
BASE64 END
```

Note: The encrypted password corresponds to the big suite of alphanumeric characters between BASE64 BEGIN and BASE64 END. The long line of text is wrapped for display purposes only.

- Replace the clear password with the encrypted one, and specify the key used to encrypt it (*mykey* in our case):

```
authenticate.myentry.login          = "fourjs"
authenticate.myentry.password.mykey = "dBy3E5JCVxuoxsR+aOBVfp1j0S
wQPt+hdjpMKriWvO2xMd5rFnFEwv+sPPd4w
/onWviG0M5mqubBeS7QUlt/ZK0D1aO9/R5RVa5wylQu//6vxfyd8NG/SFJmlVH63k
uyXfiVfq6bHo5+nlQZpVjSHfF2msET3S9HTpZUt4NblP4="
```

Note: Do not forget to put quotes around the base64 form; otherwise the '=' character is interpreted during the loading of FGLPROFILE. The long line of text is wrapped for display purposes only.

Encrypting a HTTP Authenticate password using a certificate in the Windows key store

- Find the HTTP Authenticate entry with the password you want to encrypt:

```
authenticate.myentry.login    = "fourjs"
authenticate.myentry.password = "mypassword"
```

- Add the subject of the certificate registered in the Windows key store:

```
security.mykey.subject = "Georges"
```

- Encrypt the password with **fglpass**:

```
$ fglpass -s Georges
Enter password :mypassword
```

The **fglpass** output looks like this:

```
BASE64 BEGIN
dBy3E5JCVxuoxsR+aOBVfp1j0SwQPt+hdjpMKriWvO2xMd5rFnFEwv+sPPd4w/onW
```

```
viG0M5mqubBeS7QUlt/ZK0D1aO9/R5RVa5wylQu//6vxfyd8NG
/SFJmlVH63kuyXfiVfq6bHo5+nlQZpVjSHfF2msET3S9HTpZUt4NblP4=
BASE64 END
```

Note: The encrypted password corresponds to the big suite of alphanumeric characters between BASE64 BEGIN and BASE64 END. The long line of text is wrapped for display purposes only.

- Replace the clear password with the encrypted one, and specify the key used to encrypt it (*mykey* in our case):

```
authenticate.myentry.login          = "fourjs"
authenticate.myentry.password.mykey = "dBy3E5JCVxuoxsR+aOBVfp1j0S
wQPt+hdjpMKriWvO2xMd5rFnFEwv+sPPd4w
/onWviG0M5mqubBeS7QUlt/ZK0D1aO9/R5RVa5wylQu//6vxfyd8NG/SFJmlVH63k
uyXfiVfq6bHo5+nlQZpVjSHfF2msET3S9HTpZUt4NblP4="
```

Note: Do not forget to put quotes around the base64 form; otherwise the '=' character is interpreted during the loading of FGLPROFILE. The long line of text is wrapped for display purposes only.

FGLPROFILE Configuration

The Genero Web Services secured communication is based on the OpenSSL engine, and allows a 4GL Web Services client, or a 4GL application using the com or xml API, to communicate with any secured server over **HTTP** or **HTTPS**. The configuration is defined from entries in the FGLPROFILE file. This is useful for deployment purposes, as no additional code modification is necessary, even if the location of the different servers changes.

Note: When using 4GL Web Services on server side, it is the Web Server that is in charge of the 4GL Web Services server security, not the 4GL server application itself. You must refer to your Web Server manual to secure the server part of the Web Services.

Configuration categories:

- Security Configuration
- HTTP basic or digest authentication
- Proxy
- Server
- Examples

Security Configuration

The following table lists the FGLPROFILE entries specifying the security certificates and algorithms the Web Services client uses for HTTPS and password encryption. Notice that the entries specify also the way an application using the low-level com or xml API performs secured communications.

Entry	Description
`security.global.script`	Filename of a script executed each time a password of a private key is required by the client. The security script accepts one argument corresponding to the filename of the private key for which the password is required, and must return the correct password or the client stops. Refer to Windows Password Script Example or Unix Password Script Example (below) for script examples.

`security.global.ca`	Filename of the Certificate Authority list, with the concatenated PEM-encoded third party X.509 certificates considered as trusted, and in order of preference.
`security.global.windowsca`	If set to TRUE, build the Certificate Authority list from the Certificate Authorities stored in the Windows key store. **Note:** This entry is valid only on Windows systems, and if *security.global.ca* is not set.
`security.global.cipher`	The list of encryption, digest, and key exchange algorithms the client is allowed to use during a secured communication. If this entry is omitted, all algorithms are supported. For more details about cipher, refer to www.openssl.org.
`security.`***`ident`***`.certificate`	Filename of the PEM-encoded client X.509 certificate.
`security.`***`ident`***`.privatekey`	Filename of the PEM-encoded private key associated to the above X509 certificate.
`security.`***`ident`***`.keysubject`	The subject string of a X.509 certificate and its associated private key registered in the Windows key store. **Note:** This entry is valid only on Windows systems.

Notes:

1. The **ident** key-word must be replaced with your own identifier, and all necessary entries must be set. See FGLPROFILE setting (below) for an example.
2. If an entry is defined more that once, only the last occurrence is taken into account.

HTTP basic or digest Authentication

The following table lists the FGLPROFILE entries that specify the login and password to use in the case of HTTP authentication to a server or a proxy. Notice that the entries specify also the login and password to use in an application using the low-level com or xml API.

Entry	Description
`authenticate.`***`ident`***`.login`	The login identifying the client to a server during HTTP Authentication.
`authenticate.`***`ident`***`.password`	The password validating the login of a client to a server during HTTP Authentication. **Note:** As passwords should never be in clear text, it is recommended that you encrypt them with the **fglpass** tool. For more information, see FGLPROFILE password encryption.
`authenticate.`***`ident`***`.realm`	The string identifying the server to the client during HTTP Authentication. If that string doesn't match the server's string, authentication fails. **Note:** This parameter is optional, but it is recommended that you check the server identity, especially if the server's location is suspicious.
`authenticate.`***`ident`***`.scheme`	One of the following strings representing the different HTTP Authentication mechanisms. • **Anonymous** (Default value) The client doesn't know anything about the server, and performs a first request to retrieve the server authentication mechanism. Then it uses the login and password to authenticate to the server using the Basic or Digest mechanism,

depending on the server returned value.

- **Basic**
The client authenticates itself to the server at first request, by sending the login and the password using the Basic authentication mechanism.

- **Digest**
The client performs a first request without any login and password, to retrieve the server information before authenticating itself to the server in a second request using the Digest mechanism.

Notes:

1. The **ident** key-word must be replaced with your own identifier and both entries must be set. See FGLPROFILE setting (below) for an example.
2. If an entry is defined more that once, only the last occurrence is taken into account.

Proxy Configuration

The following table lists the FGLPROFILE entries that specify how the Web Services client communicates with a proxy. Notice that the entries specify also the way an application using the low-level com or xml API communicates with a proxy.

Entry	Description
`proxy.http.location`	Location of the HTTP proxy defined as **host:port** or **ip:port** **Note:** if the port is omitted, the port 80 is used
`proxy.http.list`	The list of beginning host names, separated with semicolons, for which the Web Services client doesn't go via the HTTP proxy.
`proxy.http.authenticate`	The HTTP Authenticate identifier

	the Web Services client uses to authenticate itself to the HTTP proxy.
`proxy.https.location`	Location of the HTTPS proxy defined as **host:port** or **ip:port** **Note:** if the port is omitted, the port 443 is used
`proxy.https.list`	The list of beginning host names, separated with semicolons, for which the Web Services client doesn't go via this HTTPS proxy.
`proxy.https.authenticate`	The HTTP Authenticate identifier the Web Services client uses to authenticate itself to the HTTPS proxy.

Notes:

1. If an entry is defined more that once, only the last occurrence is taken into account.

Server Configuration

The following table lists the FGLPROFILE entries that specify the correct way a Web Services client connects to an end point (usually a server). Notice that the entries specify also the way an application using the low-level com or xml API connects to an end point.

Entry	Description
`ws.`***`ident`***`.url`	The endpoint URL of the server.
`ws.`***`ident`***`.cipher`	The list of encryption, digest and key exchange algorithms, the client is allowed to use during a secured communication to that server. It overwrites the global definition.
`ws.`***`ident`***`.verifyserver`	If set to TRUE, the client performs a strict server identity validation. If not fulfilled, it stops the communication; otherwise no

	server identity verification is performed. The default value is TRUE.
`ws.ident.security`	The Security identifier the client uses to perform an HTTPS communication to the server.
`ws.ident.authenticate`	The HTTP authenticate identifier the client uses to authenticate itself to the server.

Notes:

1. The **ident** key-word must be replaced with your own identifier. All necessary entries, depending on the remote server's configuration, must be set. For more information, see FGLPROFILE sample (below).
2. You can use the unique identifier in the 4GL code instead of the server URL, with the **alias://** prefix as for instance **alias://ident**.
3. If an entry is defined more that once, only the last occurrence is taken into account.

Examples

Windows Password Script Example

```
@echo off
REM -- Windows password script

IF "%1" == "Cert/MyPrivateKeyA.pem" GOTO KeyA
IF "%1" == "Cert/MyPrivateKeyB.pem" GOTO KeyB
GOTO end
:KeyA
ECHO PasswordA
GOTO end
:KeyB
ECHO PasswordB
GOTO end
:end
GOTO :EOF
```

Unix Password Script Example

```
  # Unix password script

  if [ "$1" == "Cert/MyPrivateKeyA.pem" ]
   then
    echo PasswordA
  fi
  if [ "$1" == "Cert/MyPrivateKeyB.pem" ]
   then
    echo PasswordB
  fi
```

FGLPROFILE sample

The following is an FGLPROFILE sample, configured for a connection to a HTTPS server via a proxy, and with HTTP and Proxy Authentication.

```
  ############################
  ## Security configuration ##
  ############################
  security.global.script     =  "Cert/password.sh"
  security.global.ca         =  "Cert/CAList.pem"
  security.global.cipher     =  "HIGH" # Use only HIGH encryption
ciphers
  security.mykey.certificate =  "Cert/MyCertificateA.crt"
  security.mykey.privatekey  =  "Cert/MyPrivateKeyA.pem"

  ##############################
  ## Proxy HTTP Authentication ##
  ##############################
  authenticate.proxyauth.login    =  "myapplication"
  authenticate.proxyauth.password =  "fourjs"
  authenticate.proxyauth.scheme   =  "Basic"

  ##############################
  ## HTTPS Proxy configuration ##
  ##############################
  proxy.https.location     =  "10.0.0.170"
  proxy.https.list         =  "www.4js.com;www.4js1.com"
  proxy.https.authenticate =  "proxyauth"

  ##############################
  ## Server HTTP Authentication ##
  ##############################
  authenticate.serverauth.login    =  "fourjs"
  authenticate.serverauth.password =  "password"

  ########################
  ## Server configuration ##
  ########################
  ws.myserver.url          =  "https://www.MyMachine.com/cgi-
bin/fglccgi.exe/ws/r/MyWebService"
  ws.myserver.authenticate =  "serverauth"
  ws.myserver.security     =  "mykey"
```

Tutorial: Configuring a Client to access an HTTPS Server

Configuration steps to access a server in HTTPS

Step 1: Create the client's X.509 certificate and private key

Step 2: Create the client's Certificate Authority list

Step 3: Add the client's security configuration to FGLPROFILE

Step 4: Set the global certificate authority list in FGLPROFILE

Step 5: Add configuration entries for the server to FGLPROFILE

Step 1: Create the client's X.509 certificate and private key

- Create the Certificate Signing Request and private key:
  ```
  $ openssl req -new -out MyClient.csr
  ```
  Note: by default, **openssl** outputs the private key in the **privkey**.**pem** file.
- Remove the password from the RSA private key:
  ```
  $ openssl rsa -in privkey.pem -out MyClient.pem
  ```
  Note: the key is also renamed in **MyClient**.**pem**.
- Create the self-signed X.509 certificate valid for a period of 1 year:
  ```
  $ openssl x509 -in MyClient.csr -out MyClient.crt -req -signkey
  MyClient.pem -days 365
  ```

Notes:

1. Most servers do not check the identity of the clients. For these servers, the client's certificate does not necessary need to be trusted; it is only used for data encryption purpose. If, however, the server performs client identification, you must trust a Certificate Authority in which it has total confidence concerning the validity of the client's certificates.

Step 2: Create the client's certificate authority list

- Retrieve the certificate of the HTTPS server:
  Type the server's URL in your Internet browser. When prompted, save the certificate to disk.
- Create the client's Certificate Authority List from the certificate that you saved to disk in the previous step:

```
$ openssl x509 -in ServerCertificate.crt -text >>
ClientCAList.pem
```

Step 3: Add the client's security configuration to FGLPROFILE

The client security entry defines the certificate and the associated private key used by the Genero Web Services client during a HTTPS communication. The security entry must be defined with an unique identifier (**id1** for example).

```
security.id1.certificate = "MyClient.crt"
security.id1.privatekey  = "MyClient.pem"
```

Step 4: Set the global certificate authority list in FGLPROFILE

The global certificate authority list entry defines the file containing the certificate authority list used by the Genero Web Services client to validate all certificates coming from the different servers.

```
security.global.ca = "ClientCAList.pem"
```

Step 5: Add configuration entries for the server to FGLPROFILE

The Genero Web Services client needs a set of configuration entries that specify how to communicate with the server. The following entries must be defined with an unique identifier (such as **myserver**):

```
ws.myserver.url          = "https://www.MyServer.com/cgi-
bin/fglccgi.exe/ws/r/MyWebService"
ws.myserver.security     = "id1"
```

Notes:

1. The unique identifier **myserver** can be used in the 4GL client code in place of the actual URL.
2. The security entry value must match the unique identifier defined by the client security entry created in Step 3.

Tutorial: Configuring a Client to connect via a Proxy

Configuration steps to connect via a HTTP proxy

Step 1: Add the location of the proxy into FGLPROFILE

Step 2: Define the list of host names the client will not have to connect via a proxy

---

Step 1 : Add the location of the proxy into FGLPROFILE

To configure a client to connect via a HTTP proxy located at IP **10.0.0.170** and listening on port number **8080**, add the following to the FGLPROFILE file:

```
proxy.http.location = "10.0.0.170:8080"
```

Note:

To configure the client to connect via a HTTPS proxy, replace *http* with *https*.

Step 2 : Define the list of host names the client will not have to connect to via a proxy

To not connect via a proxy for each host beginning with **www.4js.com** or **www.google.**, do the following:

```
proxy.http.list = "www.4js.com;www.google."
```

Note:

The same operation is available for a HTTPS proxy; replace *http* with *https*.

Tutorial: Configuring a client for HTTP and Proxy Authentication

Configuration steps to authenticate the client to a server	Configuration steps to authenticate the client to a proxy
Step 1: Add a HTTP Authenticate entry to FGLPROFILE	Step 1: Add an HTTP Authenticate entry to FGLPROFILE
Step 2: Configure the client to authenticate to a server	Step 2: Configure the client to authenticate to a proxy

---

Step 1 (HTTP Authentication): Add a HTTP Authenticate entry to FGLPROFILE

To connect to a server with HTTP Authentication (See RFC 2617 for more details), it is necessary to define the client login and password as registered on the server side.
The following two entries must be defined with an unique identifier (**httpauth** for our example) to define a HTTP Authentication with **fourjs** as login and **passphrase** as password:

```
authenticate.httpauth.login     = "fourjs"
authenticate.httpauth.password  = "passphrase"
```

Step 2 (HTTP Authentication): Configure the client to authenticate to a server

As a client is able to connect to different servers that do not know the client with the same login and password, it is necessary to specify the login and password that correspond to each server.
To authenticate the client known as **fourjs** and with the password **passphrase** by the server **myserver**, add the following entry:

```
ws.myserver.authenticate = "httpauth"
```

Step 1 (Proxy Authentication): Add a HTTP Authenticate entry to FGLPROFILE

To connect via a proxy with HTTP Proxy Authentication (See RFC 2617 for more details), it is necessary to define the client login and password as registered on the HTTP proxy.
The following two entries must be defined with an unique identifier (**proxyauth** for our example) to define a HTTP Proxy Authentication with **myapplication** as login and **mypassword** as password:

```
authenticate.proxyauth.login     = "myapplication"
authenticate.proxyauth.password  = "mypassword"
```

Step 2 (Proxy Authentication): Configure the client to authenticate to a Proxy

To authenticate a client known as **myapplication** and with **mypassword** as password by the HTTP Proxy, add the following entry to the HTTP proxy configuration:

```
proxy.http.authenticate = "proxyauth"
```

Notes:

1. To authenticate the client to a HTTPS proxy, replace *http* with *https*.

Deploying a Client and a Server for HTTPS

Creation of all X.509 certificates	Client Configuration	Server Configuration
Step 1: Create the Root Certificate Authority	Step 6: Define the global certificate authority list	Step 11: Register the server as a Web Service in the GAS
Step 2: Create the server's certificate and private key	Step 7: Define the client security configuration	Step 12: Configure apache for HTTPS
Step 3: Create the client's certificate and private key	Step 8: Define the HTTP authentication configuration	Step 13: Configure apache for HTTP basic authentication
Step 4: Create the server's certificate authority list	Step 9: Encrypt the HTTP authentication password	
Step 5: Create the client's certificate authority list	Step 10: Configure the client to access the server	

Step 1 : Create the Root Certificate Authority

- Create the Root Authority's Certificate Signing Request and private key:
```
$ openssl req -new -out MyCompanyCA.csr -keyout MyCompanyCA.pem
```
- Create the Root Certificate Authority for a period of validity of 2 years:
```
$ openssl x509 -trustout -in MyCompanyCA.csr -out MyCompanyCA.crt
-req -signkey MyCompanyCA.pem -days 730
```

Notes:

1. The private key file (**MyCompanyCA.pem**) of a Root Certificate Authority must be handled with care. This file is responsible for the validity of all other certificates it has signed. As a result, it must not be accessible by other users.

Step 2 : Create the server's certificate and private key

- Create the server's Certificate Signing Request and private key:
  ```
  $ openssl req -new -out MyServer.csr
  ```
  Note: By default, **openssl** outputs the private key in the **privkey**.**pem** file.
- Remove the password from the private key:
  ```
  $ openssl rsa -in privkey.pem -out MyServer.pem
  ```
  Note: The key is also renamed in **MyServer**.**pem**.
- Create the server's Certificate trusted by the Root Certificate Authority:
  ```
  $ openssl x509 -in MyServer.csr -out MyServer.crt -req -signkey
  MyServer.pem -CA MyCompanyCA.crt -CAkey MyCompanyCA.pem
  ```

Notes:

1. The purpose of the server's Certificate is to identify the server to any client that connects to it. Therefore, **the subject of that server's certificate must match the hostname of the server as it is known on the network**; otherwise the client will be suspicious about the server's identity and stop the communication. For instance, if the URL of the server is *https:///www.MyServer.com/cgi-bin/fglccgi.exe/ws/r/MyWebService*, the subject must be *www.MyServer.com*.

Step 3 : Create the client's certificate and private key

- Create the client's Certificate Signing Request and private key:
  ```
  $ openssl req -new -out MyClient.csr
  ```
  Note: openssl by default, outputs the private key in the **privkey**.**pem** file.
- Remove the password from the private key:
  ```
  $ openssl rsa -in privkey.pem -out MyClient.pem
  ```
  Note: The key is also renamed in **MyClient**.**pem**.
- Create the client's Certificate trusted by the Root Certificate Authority:
  ```
  $ openssl x509 -in MyClient.csr -out MyClient.crt -req -signkey
  MyClient.pem -CA MyCompanyCA.crt -CAkey MyCompanyCA.pem
  ```

Notes:

1. The purpose of the client's Certificate is to identify the client to any server; therefore the subject of the certificate must correspond to the client's identity as it is known by the servers.

Step 4 : Create the server's certificate authority list

- Create the server's Certificate Authority List:
```
$ openssl x509 -in MyCompanyCA.crt -text >> ServerCAList.pem
```

Notes:

1. As the server trusts only the Root Certificate Authority, the list contains only that one certificate authority; all other certificates that were trusted by the Root Certificate Authority will also be considered as trusted by the server.

Step 5 : Create the client's certificate authority list

- Create the client's Certificate Authority List:
```
$ openssl x509 -in MyCompanyCA.crt -text >> ClientCAList.pem
```

Notes:

1. As the client trusts only the Root Certificate Authority, the list contains only that one certificate authority; all other certificates that were trusted by the Root Certificate Authority will also be considered as trusted by the client.

Step 6 : Define the global certificate authority list

The global certificate authority list entry defines the file containing the certificate authority list that the Genero Web Services client will use to validate all certificates coming from the different servers it will connect to. The certificate authority list entry must be defined as follows:

```
security.global.ca = "ClientCAList.pem"
```

Step 7 : Define the client security configuration

The client security entry defines the certificate and the associated private key that the Genero Web Services client will use during a communication with a HTTPS server. The security entry must be defined with an unique identifier (**id1** in our case).

```
security.id1.certificate = "MyClient.crt"
security.id1.privatekey  = "MyClient.pem"
```

Notes:

1. If the private key is protected with a password, you must remove it or create a script that returns the password on demand.

Step 8 : Define the HTTP authentication configuration

As our server supports HTTP authentication (See RFC 2617 for more details) , it is necessary to define the client login and password with the same value as registered on the server side. The following two entries must be defined with an unique identifier (**id2** in our case).

```
authenticate.id2.login     = "fourjs"
authenticate.id2.password  = "mypassword"
```

Step 9 : Encrypt the HTTP authentication password

Due to security leaks, it is not recommended that you have a password in clear text. The Genero Web Services package provides the tool **fglpass**. This tool encrypts a password with a certificate that is readable only with the associated private key. To encrypt the HTTP authentication password, do the following:

- Encrypt the clear text password with **fglpass** using the client certificate:
  ```
  $ fglpass -c MyClient.crt
  Enter password :mypassword
  ```
  Note: fglpass outputs the encrypted password on the console but can be redirected to a file.

- Modify the HTTP authentication password entry by specifying the security configuration to use to decrypt it (id1 in our case)

```
authenticate.id2.password.id1="HWTFu8QE2t3e5D4joy7js8mB95oOGTzLmcAor9j5
DS+CloiliGCwZvZ9eWpfmIWSON9IwoiJheYxfnu20uaGGmmiUGiHxT6341ePXNSicu32Ntl
Vp9t6RcS0wN/p9a6D4XtiD9iHW7iQvXhqC9uamd3gI9Q3GhHwXOMMlY//c8Y="
```

Notes:

1. The size of the encrypted password depends on the size of the public key, and can change according to the certificate used to encrypt it.

Step 10 : Configure the client to access the server

The Genero Web Services client needs a set of configuration entries to specify the security configuration and the HTTP authentication (id1 and id2, respectively) to use when accessing our server. The following entries must be defined with a unique identifier (**myserver** in our case):

```
ws.myserver.url          = "https://www.MyServer.com/cgi-
bin/fglccgi.exe/ws/r/MyWebService"
ws.myserver.security     = "id1"
ws.myserver.authenticate = "id2"
```

Notes:

1. The unique identifier **myserver** can be used in the 4GL client code instead of the real URL.

Step 11 : Register the server as a Web Service in the GAS

As the Web Server is in charge of the complete HTTPS protocol with all the clients, there is no additional GAS configuration needed to add security. Simply register the 4GL server to the list of Web Services of the GAS. For more information, refer to the *Genero Application Server Manual* documentation.

Step 12 : Configure apache for HTTPS

You must configure Apache to support HTTPS by adding the required modules. Please refer to the Apache Web server documentation for more information.

- For the Apache 1.3 manual, go to http://httpd.apache.org/docs/1.3.
- For the Apache 2.0 manual, go to http://httpd.apache.org/docs/2.0/.

Once the Apache Web server supports HTTPS, you must change or add the following directives to the apache configuration file:

- Set the Apache Web server Certificate Authority List directive created in Step 4:
  SSLCACertificateFile D:/Apache-Server/conf/ssl/ServerCAList.pem
- Set the Apache Web server Certificate and associated private key directives created in Step 2:
  SSLCertificateFile D:/Apache-Server/conf/ssl/MyServer.crt
  SSLCertificateKeyFile D:/Apache-Server/conf/ssl/MyServer.pem
- Require the Apache Web server to verify the validity of all client certificates:
  SSLVerifyClient require

Notes:

1. The Apache Web server must be started on a machine where the host is the same as the one defined in the subject of the server's certificate (*www.MyServer.com* in our case).

Step 13 : Configure apache for HTTP basic authentication

You must configure Apache to support HTTP basic authentication by adding the required modules.

Please refer to the Apache Web server documentation for more information.

- For the Apache 1.3 manual, go to http://httpd.apache.org/docs/1.3.
- For the Apache 2.0 manual, go to http://httpd.apache.org/docs/2.0/.

Once the Apache Web server supports HTTP basic authentication, you must:

1. Add an user to the Apache Web server basic authentication file with the same login and password as defined in Step 8.

Apache provides the tool **htpasswd** that you can use to create the file and add the user. To add the user **fourjs** with the password **mypassword** to a new file called **myusers**:

```
$ htpasswd -c myusers fourjs mypassword
```

Note: to add additional users, remove the option '`-c`'.

2. Add an Apache Web server location directive that enables you to group several directives for one URL. (In our case, the URL is /cgi-bin/fglccgi.exe/ws/r/MyWebService).
   The following example (based on Apache 2.0) defines the HTTP authentication type (Basic), with a user file (user-basic) containing the login and password of those who are allowed to access the service.

```
<Location /cgi-bin/fglccgi.exe/ws/r/MyWebService>
    AllowOverride None
    Order allow,deny
    Allow from all
    #
    # Basic HTTP authenticate configuration
    #
    AuthName "Top secret"
    AuthType Basic
    AuthUserFile "D:/Apache-Server/conf/authenticate/myusers"
    Require valid-user # Means any user in the password file
</Location>
```

3. For more information about Apache Web server directives, refer to the Apache Web Server manual.

How to Call Java APIs from Genero

Overview

This tutorial explains how to call a Java library from Genero, using Genero and Java Web services. This can easily be done using the Java JAX-WS framework and without any strong linkage between Genero and Java. We will use a Java codebar creation library to build a picture from a code.

Note: Accessing a .NET library could be done in the same manner.

Prerequisites

- A JRE 1.5 or above
- The Java codebar library is available here; you must add the following JARs to the java CLASSPATH : **barcode.jar** and **BarcodeReader.jar**
    - **Note:** the trial version has some functions partially implemented.
- Download the JAX-WS framework from the Sun metro project here; add the following JAR to the java CLASSPATH: **webservices-tools.jar**
    - **Note:** this .jar is only necessary to generate the WSDL at runtime.

Using the barcode library

The **barcode library** is composed of two libraries - one for building a barcode image from a numeric code, and one for reading a barcode image to return the numeric code. This section depends on the library you want to use in Genero.

In our tutorial, we create two functions called **buildImage** and **readImage;** the Java implementation is below:

- buildImage(type : String, code : String) : byte[ ]

```
try {
  Barcode builder=new Barcode();
  builder.setType(GetBarcodeBuilderType(type));
  builder.setData(data);
  builder.setAddCheckSum(true);
  ByteArrayOutputStream out=new ByteArrayOutputStream();
  if (builder.createBarcodeImage(out)) {
    byte[] ret = out.toByteArray();
    return ret;
  } else {
    return null;
  }
} catch(Exception e) {
  return null;
}
```

- readImage(type : String, img : byte[ ]) : String

```
try {
  File f=new File("tmp.jpg");
  FileOutputStream stream=new FileOutputStream(f);
  stream.write(img);
  stream.close();
  String[] datas = BarcodeReader.read(f, GetBarcodeReaderType(type));
  if (datas==null) {
    return null;
  } else {
    String ret = datas[0];
    return ret;
  }
} catch (Exception e) {
  return null;
}
```

The following two functions convert the type of a code bar to the type expected by the library:

```
private int GetBarcodeBuilderType(String str) {
    if (str.equals("CODABAR")) {
      return Barcode.CODABAR;
    } else if (str.equals("CODE11")) {
      return Barcode.CODE11;
    } else if (str.equals("CODE128")) {
      return Barcode.CODE128;
    } else if (str.equals("CODE128A")) {
      return Barcode.CODE128A;
    } else if (str.equals("CODE128B")) {
      return Barcode.CODE128B;
    } else if (str.equals("CODE128C")) {
```

```
      return Barcode.CODE128C;
    } else if (str.equals("CODE2OF5")) {
      return Barcode.CODE2OF5;
    } else if (str.equals("CODE39")) {
      return Barcode.CODE39;
    } else if (str.equals("CODE39EX")) {
      return Barcode.CODE39EX;
    } else if (str.equals("CODE93")) {
      return Barcode.CODE93;
    } else if (str.equals("CODE93EX")) {
      return Barcode.CODE93EX;
    } else if (str.equals("EAN13")) {
      return Barcode.EAN13;
    } else if (str.equals("EAN13_2")) {
      return Barcode.EAN13_2;
    } else if (str.equals("EAN13_5")) {
      return Barcode.EAN13_5;
    } else if (str.equals("EAN8")) {
      return Barcode.EAN8;
    } else if (str.equals("EAN8_2")) {
      return Barcode.EAN8_2;
    } else if (str.equals("EAN8_5")) {
      return Barcode.EAN8_5;
    } else if (str.equals("INTERLEAVED25")) {
      return Barcode.INTERLEAVED25;
    } else if (str.equals("ITF14")) {
      return Barcode.ITF14;
    } else if (str.equals("ONECODE")) {
      return Barcode.ONECODE;
    } else if (str.equals("PLANET")) {
      return Barcode.PLANET;
    } else if (str.equals("POSTNET")) {
      return Barcode.POSTNET;
    } else if (str.equals("RM4SCC")) {
      return Barcode.RM4SCC;
    } else if (str.equals("UPCA")) {
      return Barcode.UPCA;
    } else if (str.equals("UPCE")) {
      return Barcode.UPCE;
    } else {
      return -1;
    }
  }



private int GetBarcodeReaderType(String str) {
    if (str.equals("CODABAR")) {
      return BarcodeReader.CODABAR;
    } else if (str.equals("CODE11")) {
      return BarcodeReader.CODE11;
    } else if (str.equals("CODE128")) {
      return BarcodeReader.CODE128;
    } else if (str.equals("CODE39")) {
      return BarcodeReader.CODE39;
    } else if (str.equals("CODE39EX")) {
```

```
      return BarcodeReader.CODE39EX;
    } else if (str.equals("CODE93")) {
      return BarcodeReader.CODE93;
    } else if (str.equals("DATAMATRIX")) {
      return BarcodeReader.DATAMATRIX;
    } else if (str.equals("EAN13")) {
      return BarcodeReader.EAN13;
    } else if (str.equals("EAN8")) {
      return BarcodeReader.EAN8;
    } else if (str.equals("INTERLEAVED25")) {
      return BarcodeReader.INTERLEAVED25;
    } else if (str.equals("ITF14")) {
      return BarcodeReader.ITF14;
    } else if (str.equals("ONECODE")) {
      return BarcodeReader.ONECODE;
    } else if (str.equals("PLANET")) {
      return BarcodeReader.PLANET;
    } else if (str.equals("POSTNET")) {
      return BarcodeReader.POSTNET;
    } else if (str.equals("QRCODE")) {
      return BarcodeReader.QRCODE;
    } else if (str.equals("RM4SCC")) {
      return BarcodeReader.RM4SCC;
    } else if (str.equals("RSS14")) {
      return BarcodeReader.RSS14;
    } else if (str.equals("RSSLIMITED")) {
      return BarcodeReader.RSSLIMITED;
    } else if (str.equals("UPCA")) {
      return BarcodeReader.UPCA;
    } else if (str.equals("UPCE")) {
      return BarcodeReader.UPCE;
    } else {
      return -1;
    }
  }
```

Calling Java from Genero

The integration of one or several Java libraries with multiple methods in a Genero application can be performed as described below:

Step 1: Write a new java class

Instead of writing the functions in 4GL, you simply need to write them in a Java class with the methods you want to use in 4GL as described below. In our example, the two functions are **buildImage** and **readImage**. And of course, don't forget to import the necessary Java import instructions.

```
import com.barcodelib.barcodereader.BarcodeReader;
import com.barcodelib.barcode.Barcode;
import java.io.*;
```

```
import javax.jws.*;
import javax.jws.soap.SOAPBinding;
import javax.xml.ws.Endpoint;

public class BarcodeService {
  public byte[] buildImage(String type,String data)
  {
    /*BUILDIMAGE IMPLEMENTATION CODE DESCRIBED ABOVE*/
  }
  public String readImage(String type,byte[] img)
  {
    /*READIMAGE IMPLEMENTATION CODE DESCRIBED ABOVE*/
  }
}
```

Notice that if you want the service to run standalone, you must also add following the main method to tell the system the port number on which the service will run:

```
public static void main(String[] args)
{
 String endpointUri = "http://localhost:9090/";
 Endpoint.publish(endpointUri, new BarcodeService ());
 System.out.println("BarcodeService started at " + endpointUri);
}
```

Step 2 : Transform the Java class in a Web Service

To transform the previous java class in a Web Service, simply add a WebService annotation:

```
@WebService(targetNamespace = "http://www.4js.com/barcode",
  name="Barcode",
  serviceName="BarcodeService")
public class BarcodeService {
   ...
}
```

This defines all public and non static methods of the class as operations of the **BarcodeService** Web Service.

Step 3 : Start the service

Compile the previously created java class, and run it.

Commands to compile and execute the service in standalone mode:

```
$ javac BarcodeService.java
```

```
$ java BarcodeService
```

Once the service is started, it is ready to accept requests and you can also retrieve its WSDL at following URL:

http://localhost/9090/BarcodeService?WSDL

Note: If you want the service to be started on a web server, you must deploy it first using Eclipse or the Web Server deployment tools.

Step 4 : Generate 4GL stub to access the Java library

Use the **fglwsdl** tool to generate the client stub to access the BarcodeService:

```
$ fglwsdl http://localhost:9090/BarcodeService?WSDL
```

This will create two 4GL files that must be compiled and linked into your 4GL application in order to call the Java barcode library functions. These files contain the 4GL interface to access the Java library where you will find the two functions, **readImage** and **buildImage,** defined in 4GL.

Step 5 : Modify your 4GL application

The last step is to modify the existing application where you want to use the Java library, by calling the 4GL functions generated in the stub. Then compile your application and the previously generated stubs, and link everything together.

Your application is now ready to use the different features of your Java library.

Example 4GL program

This program calls the **buildImage** function of the Barcode Java library.

```
GLOBALS "BarcodeService_BarcodePort.inc"

MAIN
  DEFINE wsstatus INTEGER

  IF num_args() != 3 THEN
    CALL ExitHelp()
```

```
  END IF

  LET ns1buildImage.arg0 = arg_val(1)
  LET ns1buildImage.arg1 = arg_val(2)
  LOCATE ns1buildImageResponse.return IN MEMORY

  LET wsstatus = buildImage_g()
  IF wsstatus <> 0 THEN
    DISPLAY "Error ("||wsError.code||") : ",wsError.description
  ELSE
    IF ns1buildImageResponse.return IS NULL THEN
      DISPLAY "Encoding failed"
    ELSE
      CALL ns1buildImageResponse.return.writeFile(arg_val(3))
    END IF
  END IF

  FREE ns1buildImageResponse.return

END MAIN


FUNCTION ExitHelp()
  DISPLAY arg_val(0)||" <type> <data> <filename>"
  DISPLAY "type     : codebar type such as EAN8 or CODE128"
  DISPLAY "data     : data to be encoded with a codebar [0-9A-D]"
  DISPLAY "filename : resulting image filename"
  DISPLAY "exemple  : createImage EAN8 12358723A mybarcode.jpg"
  EXIT PROGRAM (-1)
END FUNCTION
```

Conclusion

You call any Java library from Genero using Web Services, and without a strong dependency to a JVM. This follows SOA principles - it allows you to reuse the Java library in another 4GL application without any new development, you can update the Java part without recompiling any 4GL sources, and integrate any function available from a SOA platform.

How to Call .NET APIs from Genero

Overview

This document explains how to call a .NET library from Genero, using Genero and Web services, and IIS and Visual Studio .NET without any strong linkage between Genero and .NET. You can even call a .NET library from a non-Windows Genero platform.

For this tutorial we will use a .NET codebar creation library to build a picture from a numeric code, and C# as the development language. This will also work with any other .NET language.

Prerequisites

- IS (Internet Information Services) web server
- Visual Studio Professional Edition C#
  - **Note**: Visual Studio in only needed for development; you can deploy once the service was built on any IIS web server.
- The .NET codebar library is available at http://www.barcodelib.com/net_barcode/main.html ,
  - **Note**: The trial version has some functions partially implemented.
  - The .NET library is called BarcodeLib.Barcode.dll, and must be added to the Visual Studio Project.

Using the barcode library

This section depends on the library you want to use in Genero. In our tutorial, we create one function called **buildImage. T**he C# implementation is below :

- buildImage(type : String, code : String) : byte[]

```
Linear barcode = new Linear();
barcode.Data = code;
barcode.Type = GetBarcodeBuilderType(type);
barcode.AddCheckSum = true;
// save barcode image into your system
barcode.ShowText = true;
byte[] ret = barcode.drawBarcodeAsBytes();
if (ret != null) return ret;
else return null;
```

You will also need to convert the type of a code bar to the right type as expected by the library. Therefore, you will need the following function:

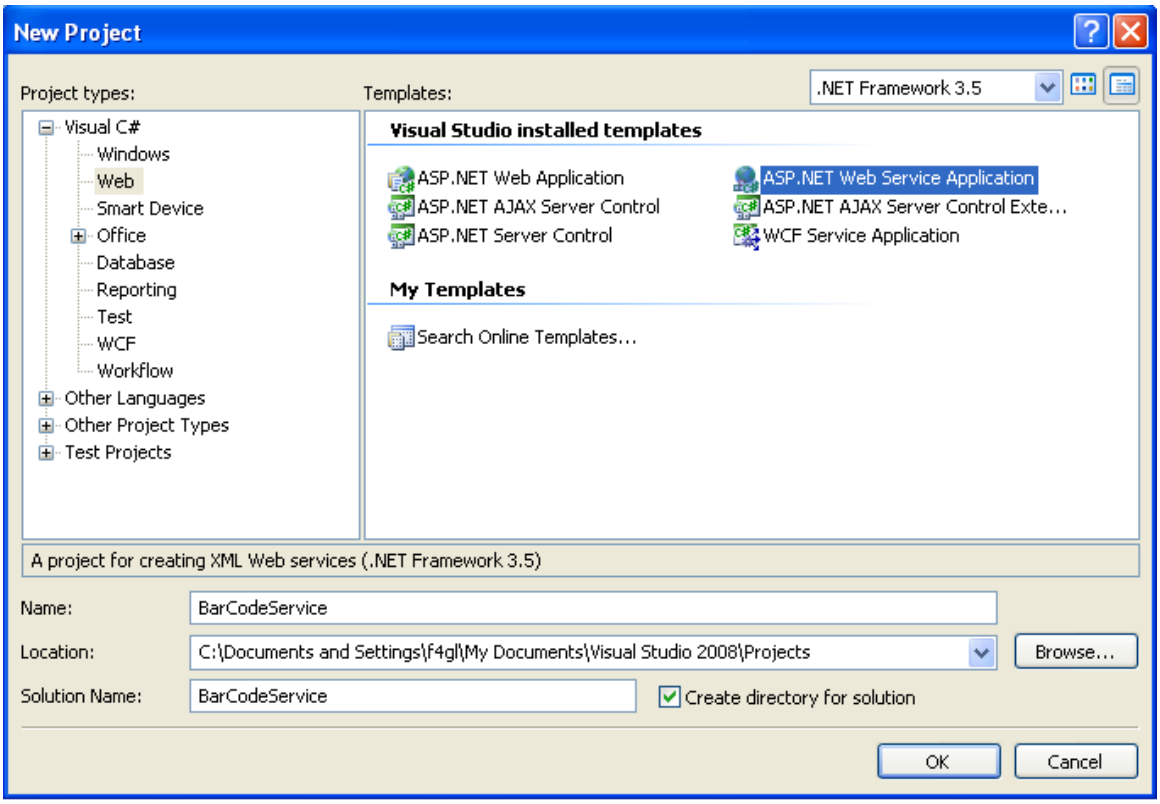
```
private BarcodeType GetBarcodeBuilderType(String str)
{
  if (str.Equals("CODABAR")) {
    return BarcodeType.CODABAR;
  } else if (str.Equals("CODE11")) {
    return BarcodeType.CODE11;
  } else if (str.Equals("CODE128")) {
    return BarcodeType.CODE128;
  } else if (str.Equals("CODE128A")) {
    return BarcodeType.CODE128A;
  } else if (str.Equals("CODE128B")) {
    return BarcodeType.CODE128B;
  } else if (str.Equals("CODE128C")) {
    return BarcodeType.CODE128C;
  } else if (str.Equals("CODE2OF5")) {
    return BarcodeType.CODE2OF5;
  } else if (str.Equals("CODE39")) {
    return BarcodeType.CODE39;
  } else if (str.Equals("CODE39EX")) {
    return BarcodeType.CODE39EX;
  } else if (str.Equals("CODE93")) {
    return BarcodeType.CODE93;
  } else if (str.Equals("EAN13")) {
    return BarcodeType.EAN13;
  } else if (str.Equals("EAN13_2")) {
    return BarcodeType.EAN13_2;
  } else if (str.Equals("EAN13_5")) {
    return BarcodeType.EAN13_5;
  } else if (str.Equals("EAN8")) {
    return BarcodeType.EAN8;
  } else if (str.Equals("EAN8_2")) {
    return BarcodeType.EAN8_2;
  } else if (str.Equals("EAN8_5")) {
    return BarcodeType.EAN8_5;
```

```
  } else if (str.Equals("INTERLEAVED25")) {
    return BarcodeType.INTERLEAVED25;
  } else if (str.Equals("ITF14")) {
    return BarcodeType.ITF14;
  } else if (str.Equals("ONECODE")) {
    return BarcodeType.ONECODE;
  } else if (str.Equals("PLANET")) {
    return BarcodeType.PLANET;
  } else if (str.Equals("POSTNET")) {
    return BarcodeType.POSTNET;
  } else if (str.Equals("RM4SCC")) {
    return BarcodeType.RM4SCC;
  } else if (str.Equals("UPCA")) {
    return BarcodeType.UPCA;
  } else if (str.Equals("UPCE")) {
    return BarcodeType.UPCE;
  } else {
    throw new Exception();
  }
}
```

Calling .Net from Genero

Step 1: Create an ASP.NET Web Service Application

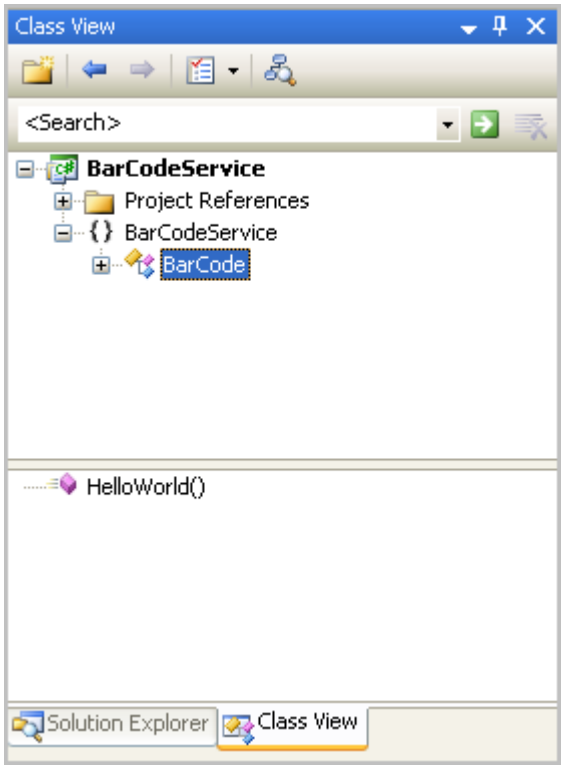
Start Visual Studio, and create a new web project with the name BarCodeService as shown in the following image:



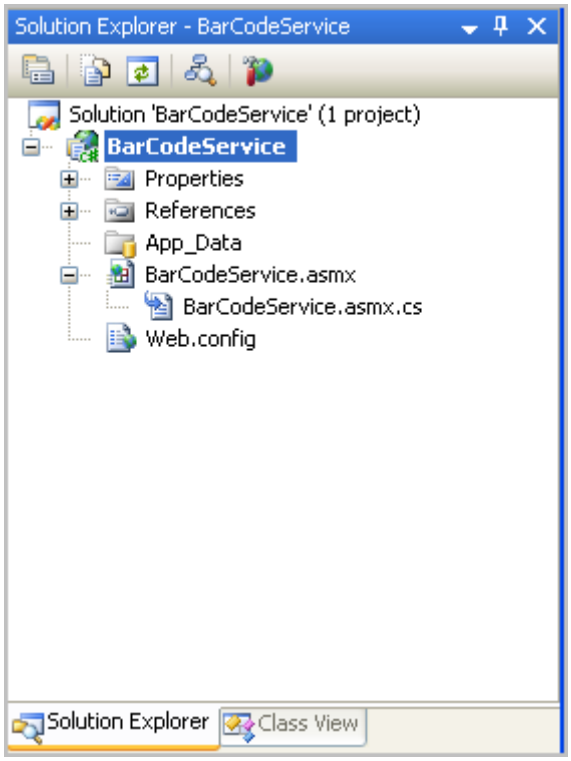
Step 2: Rename the generated files

Rename the generated class called Service1 with an appropriate name such as BarCode, and the file Service1.asmx to BarCodeService.asmxm, for instance. The .asmx file is the file that is accessible from the IIS web server once the application is deployed. The .asmx file also contains a reference to the default generated class, Service1, which must also be renamed to the new name (BarCode in our tutorial), in case Visual Studio didn't make the change automatically.

The class view after renaming the class:

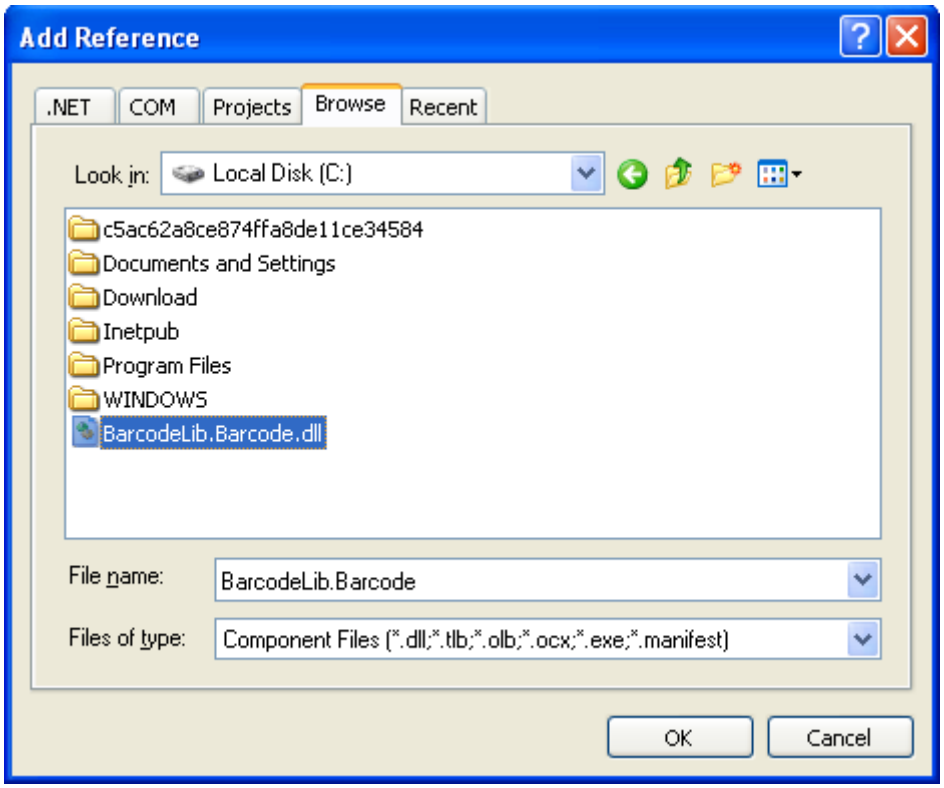


The file view after renaming the asmx file:



Step 3 : Add the barcode library as a reference

Right-click on the solution explorer, select **Add Reference** and use the **Browse** panel to enter the location of the barcode library called **BarcodeLib.Barcode.dll**:

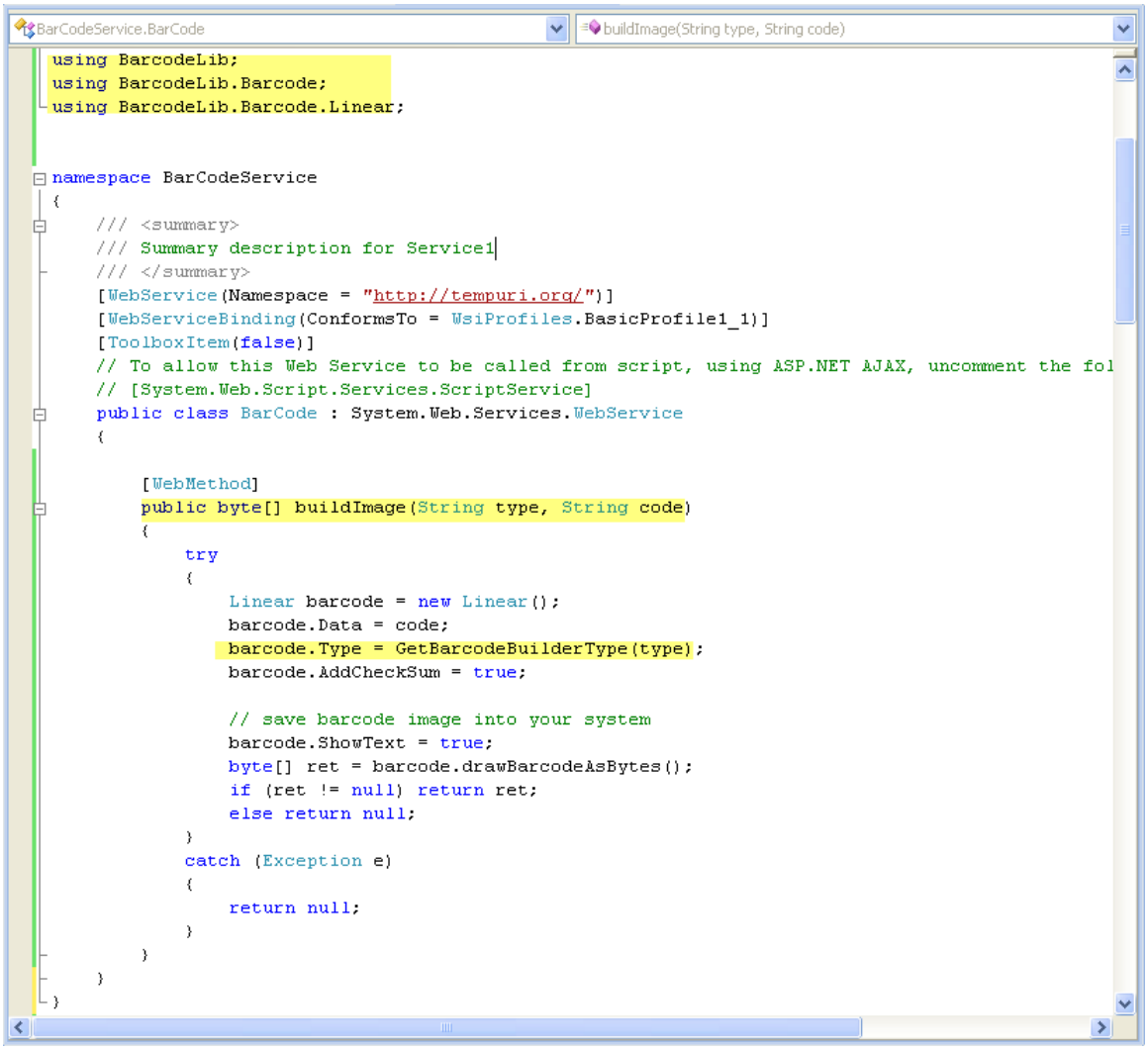


Note: By default, the barcode library will be copied to the right place when deploying on the IIS web server.

Step 4: Add the buildImage method

Remove the default generated HelloWorld method, and create the buildImage method, as shown below.

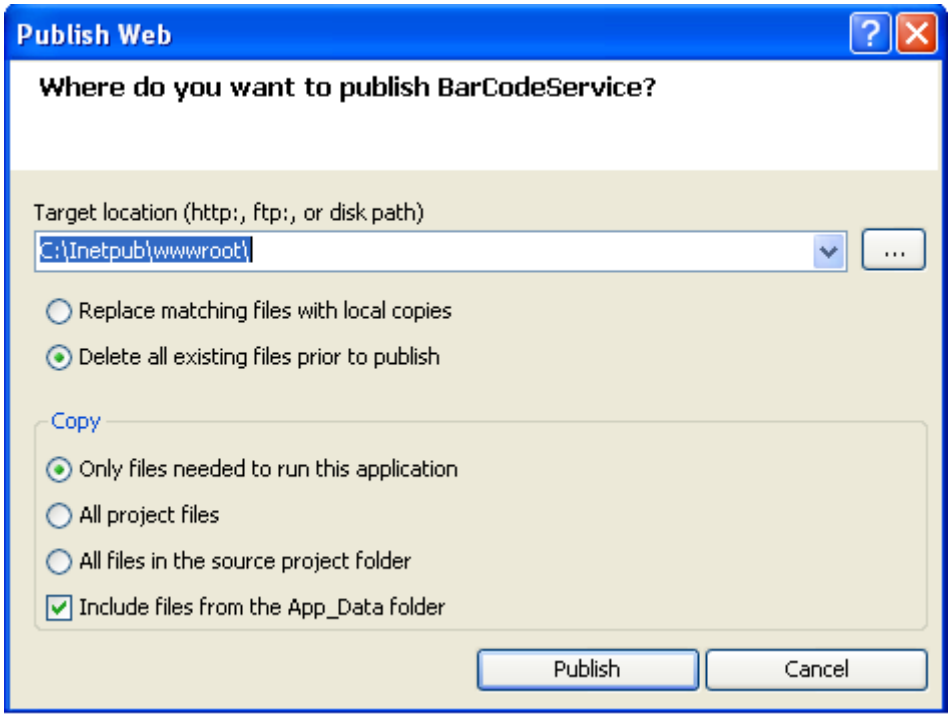
Add the three `using` instructions to import the barcode library, and to declare **buildImage** as a WebMethod. Use the **GetBarcodeBuilderType()** method to convert a string to a code as expected by the barcode library.



Step 5 : Publish the service

Build the entire application, right-click on the solution, and select the publish operation. This will copy all necessary files to your IIS web server and make your application available at an URL, depending on where you deploy it on your IIS web server.

Iin our tutorial, the service will be located at the root of the server. In other words, it will be available at **http://localhost/BarCodeService.asmx** and the WSDL at URL **http://localhost/BarCodeService.asmx?WSDL**



Step 6: Generate 4GL stub to access the .NET library

Use the fglwsdl tool to generate the client stub to access the BarcodeService, as follows:

```
$ fglwsdl http://localhost/BarCodeService.asmx?WSDL
```

This will create two 4GL files, which must be compiled and linked into your 4GL application in order to call the .NET barcode library functions. These files contain the 4GL interface to access the .NET library where you will find the f**unction buildImage**, defined in 4GL.

Step 7: Modify your 4GL application

Modify your existing application, where you want to use the .NET library, by calling the 4GL functions generated in the stub. Then compile your application and the previously generated stubs, and link everything together.

Your application is now ready to use the different features of your .NET library.

Example 4GL program

This program calls the buildImage function of the Barcode .NET library.

```
GLOBALS "BarCode_BarCodeSoap.inc"

MAIN
  DEFINE wsstatus INTEGER

  IF num_args() != 3 THEN
   CALL ExitHelp()
  END IF

  LET buildImage.type = arg_val(1)
  LET buildImage.code = arg_val(2)
  LOCATE buildImageResponse.buildImageResult IN MEMORY

  LET wsstatus = buildImage_g()
  IF wsstatus <> 0 THEN
    DISPLAY "Error ("||wsError.code||") : ",wsError.description
  ELSE
    IF buildImageResponse.buildImageResult IS NULL THEN
      DISPLAY "Encoding failed"
    ELSE
      CALL buildImageResponse.buildImageResult.writeFile(arg_val(3))
    END IF
  END IF

  FREE buildImageResponse.buildImageResult

END MAIN


FUNCTION ExitHelp()
  DISPLAY arg_val(0)||" <type> <data> <filename>"
  DISPLAY "type : codebar type such as EAN8 or CODE128"
  DISPLAY "data : data to be encoded with a codebar [0-9A-D]"
  DISPLAY "filename : resulting image filename"
  DISPLAY "exemple : createImage EAN8 12358723A mybarcode.jpg"
  EXIT PROGRAM (-1)
END FUNCTION
```

Conclusion

It is quite easy to interact with a .NET library from Genero using .NET Visual Studio and the web services. Of course you also need an IIS web server installed on your Windows system. This means that you can, in the same 4GL application, interact with .NET and Java libraries without any strong linkage between Genero and the third party libraries you want to use. This meets the SOA principles that provide more flexibility to your entire 4GL application.

You can integrate any new library from another vendor, without the risk of conflicts between different libraries that could happen if you had to link everything together in C or Java.

You can upgrade a third party library without recompiling the 4GL application, which will still work.

You can use all these third party libraries in other 4GL or other applications.

Attributes to Customize XML Serialization

- Mapping between simple 4GL and XML datatypes
- Facet constraints between simple 4GL and XML datatypes
- XML Serialization Customizing
    - Without value
    - With a mandatory value
- Default Mapping between XML and simple 4GL datatypes
- Mapping Example

Mapping between simple 4GL and XML datatypes

The following attributes cannot have values:

Attribute	Definition
XSDAnySimpleType	Map 4GL String or Varchar to XML Schema simpleType.
XSDAnyType	Map 4GL String or Varchar to XML Schema anyType.
XSDAnyURI	Map 4GL String or Varchar to XML Schema anyURI.
XSDBase64binary	Map 4GL Byte to the XML Schema base64binary.
XSDBoolean	Map 4GL Int or Smallint to XML Schema boolean.
XSDByte	Map 4GL Smallint to XML Schema byte.
XSDDate	Map 4GL Date or Datetime to XML Schema date.
XSDDateTime	Map 4GL Datetime to XML Schema dateTime.
XSDDecimal	Map 4GL Decimal to XML Schema decimal.
XSDDouble	Map 4GL Float to XML Schema double.
XSDDuration	Map 4GL Interval to XML Schema duration.
XSDEntities	Map 4GL String or VarChar to XML Schema entities.
XSDEntity	Map 4GL String or VarChar to XML Schema entity.
XSDFloat	Map 4GL Smallfloat to XML Schema float.
XSDGday	Map 4GL Datetime to XML Schema gDay.
XSDGMonth	Map 4GL Datetime to XML Schema gMonth.
XSDGMonthDay	Map 4GL Datetime to XML Schema gMonthDay.
XSDGYear	Map 4GL Datetime to XML Schema gYear.
XSDGYearMonth	Map 4GL Datetime to XML Schema gYearMonth.
XSDHexBinary	Map 4GL Byte to XML Schema hexBinary.

XSDID	Map 4GL String or VarChar to XML Schema id.
XSDIDREF	Map 4GL String or VarChar to XML Schema idRef.
XSDIDREFS	Map 4GL String or VarChar to XML Schema idRefs.
XSDInt	Map 4GL Integer to XML Schema int.
XSDInteger	Map 4GL Decimal to XML Schema integer.
XSDLanguage	Map 4GL String or VarChar to XML Schema language.
XSDLong	Map 4GL Decimal to XML Schema long..
XSDNCName	Map 4GL String or VarChar to XML Schema NCName.
XSDName	Map 4GL String or VarChar to XML Schema Name.
XSDNegativeInteger	Map 4GL Decimal to XML Schema negativeInteger.
XSDNMTOKEN	Map 4GL String or VarChar to XML Schema NMToken.
XSDNMTOKENS	Map 4GL String or VarChar to XML Schema NMTokens.
XSDNonNegativeInteger	Map 4GL Decimal to XML Schema nonNegativeInteger.
XSDNonPositiveInteger	Map 4GL Decimal to XML Schema nonPositiveInteger.
XSDNormalizedString	Map 4GL String or VarChar to XML Schema normalizedString.
XSDNotation	Not supported.
XSDPositiveInteger	Map 4GL Decimal to XML Schema positiveInteger.
XSDQName	Map 4GL String or VarChar to XML Schema QName.
XSDShort	Map 4GL Smallint to XML Schema short.
XSDString	Map 4GL String, Char, Text or VarChar to XML Schema string.
XSDTime	Map 4GL Datetime to XML Schema time.
XSDToken	Map 4GL String or VarChar to XML Schema token.
XSDUnsignedByte	Map 4GL Smallint to XML Schema unsignedByte.
XSDUnsignedInt	Map 4GL Decimal to XML Schema unsignedInt.
XSDUnsignedLong	Map 4GL Decimal to XML Schema unsignedLong.

XSDUnsignedShort	Map 4GL Integer to XML Schema unsignedShort.

Facet constraints between simple 4GL and XML datatypes

Following attributes are facet constraints depending on the XSD data type used on a simple 4GL variable to restrict the allowed value-space. (Notice that some attributes are allowed only on some XSD datatypes). Several facet constraints can be set on the same datatype, and a madatory values is expected (for example, **XSDMinLength="8"**)

Attribute	Definition
XSDLength	Define the exact number of XML character or bytes.
XSDMinLength	Define the minimum number of XML character or bytes.
XSDMaxLength	Define the maximum number of XML character or bytes.
XSDEnumeration	Define a list of allowed values separated by the character **|**.
XSDWhiteSpace	Perform a XML string manipulation before serialization or deserialization.
XSDPattern	Define the regular expression the value has to match.
XSDMinInclusive	Define the inclusive minimum value according to the datatype where it is set.
XSDMaxInclusive	Define the inclusive maximum value according to the datatype where it is set.
XSDMinExclusive	Define the exclusive minimum value according to the datatype where it is set.
XSDMaxExclusive	Define the exclusive maximum value according to the datatype where it is set.
XSDTotalDigits	Define the total number of digits.
XSDFractionDigits	Define the number of digits of the fraction part.

XML Serialization Customizing

Following attributes are used to change the default the way to serialize 4GL into XML and vice-versa, some needs a mandatory value, and some don't.

The following attributes cannot have values:

Attribute	Definition
XMLOptional	Define whether the variable can be missing.

XMLElement	Map a 4GL simple datatype to an XML Element.
XMLAttribute	Map a 4GL simple datatype to an XML Attribute.
XMLBase	Set the base type of an XML Schema simpleContent.
XMLAll	Map a 4GL Record to an XML Schema all structure.
XMLChoice	Map a 4GL Record to an XML Schema choice structure.
XMLSequence	Map a 4GL Record to an XML Schema sequence structure.
XMLSimpleContent	Map a 4GL Record to an XML Schema simpleContent structure.
XSComplexType	Map a 4GL Record type definition to an XML Schema complexType.
XMLList	Map a one-dimensional array to an XML Schema list.
XMLSelector	Define which member of an XMLChoice record is selected.

Values are mandatory for the following attributes: (for example, **XMLName="myname"**)

Attribute	Definition
XMLName	Define the XML Name of a variable in an XML document.
XMLNamespace	Define the XML Namespace of a variable in an XML document.
XMLType	Force the XML type name of a variable.
XMLTypenamespace	Force the XML type namespace of a variable.
XSTypename	Define the XML Type Name of a 4GL type definition.
XSTypenamespace	Define the XML Type Namespace of a 4GL type definition.
XMLElementNamespace	Define the default XML namespace of all children defined as XMLElement in a Record.
XMLAttributeNamespace	Define the default XML namespace of all children defined as XMLAttribute in a Record.

Default XML Mapping

By default, GWS maps BDL variables in the input or output messages of a Genero Web Services application to their corresponding XML data types, enabling values to be passed between applications and Web Services. The XML data types conform to the standard XML Schema Definition (XSD):

Data Type of BDL variable	Default XML Data Type
BYTE	xsd:base64binary
CHAR	xsd:string
DATE	xds.date
DATETIME YEAR TO FRACTION(1-5)	xsd:dateTime
DATETIME YEAR TO SECOND	xsd:dateTime
DATETIME YEAR TO HOUR	xsd:dateTime
DATETIME YEAR TO MINUTE	xsd:dateTime
DATETIME YEAR TO YEAR	xsd:gYear
DATETIME YEAR TO MONTH	xsd:gYearMonth
DATETIME YEAR TO DAY	xsd:date
DATETIME MONTH TO MONTH	xsd:gMonth
DATETIME MONTH TO DAY	xsd:gMonthDay
DATETIME DAY TO DAY	xsd:gDay
DATETIME HOUR TO HOUR	xsd:time
DATETIME HOUR TO MINUTE	xsd:time
DATETIME HOUR TO SECOND	xsd:time
DATETIME HOUR TO FRACTION(1-5)	xsd:time
DECIMAL	xsd:decimal
FLOAT	xsd:double
INTEGER	xsd:int
INTERVAL	xsd:duration
SMALLFLOAT	xsd:float

SMALLINT	xsd:short
STRING	xsd:string
TEXT	xsd:string
VARCHAR	xsd:string

In addition, the Web Service Style that you use determines what default XMLName attributes are assigned to variables.

Mapping Example

Genero version 2.0 allows you to add optional attributes to the definition of variables. These attributes can be used to map a BDL data type used in the input or output message of a Genero Web Service application to a specific XML data type, rather than using the default. For example, BDL does not have a BOOLEAN data type. If an XML Schema boolean data type is required for an application, you can use an attribute to map a BDL SMALLINT to a boolean.

The following example uses the XSDBoolean attribute to map a BDL SMALLINT variable to an XML Schema Boolean type, and assigns an uppercase name as the XMLName attribute:

```
GLOBALS
  DEFINE invoice_out RECORD
      ok SMALLINT ATTRIBUTE(XSDBoolean,XMLName="OK")
    END RECORD
END GLOBALS
```

Note: If you assign your own XMLName attributes, be sure to respect the conventions when using the RPC Service Style.

See the Writing a GWS Server application Tutorial for additional information about input and output messages.

XSDAnySimpleType

Map 4GL String or Varchar to XML Schema anySimpleType

XSDAnyType

Map 4GL String or Varchar to XML Schema anyType

XSDAnyURI

Map 4GL String or Varchar to XML Schema anyURI

XSDBase64binary

Map 4GL Byte to XML Schema base64binary

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 BYTE ATTRIBUTE(XSDBase64binary,XMLName="Val")
             END RECORD
<Root>
  <Val>F0FFC8D27FF001547FC219E1FFF009F0FFC8D27FF001547D</Val>
</Root>
```

XSDBoolean

Map 4GL Int or Smallint to XML Schema boolean

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 INTEGER ATTRIBUTE(XSDBoolean,XMLName="Val")
             END RECORD
<Root>
  <Val>true</Val>
</Root>
```

XSDByte

Map 4GL Smallint to XML Schema byte

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 SMALLINT ATTRIBUTE(XSDByte,XMLName="Val")
             END RECORD
```

```
<Root>
  <Val>-126</Val>
</Root>
```

XSDDate

Map 4GL Date or Datetime to XML Schema date

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DATE ATTRIBUTE(XSDDate,XMLName="Val")
             END RECORD
<Root>
  <Val>2006-06-29+01:00</Val>
</Root>
```

XSDDateTime

Map 4GL Datetime to XML Schema dateTime

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DATETIME ATTRIBUTE(XSDDateTime,XMLName="Val")
             END RECORD
<Root>
  <Val>2006-06-29T09:35:26.13584+01:00</Val>
</Root>
```

XSDDecimal

Map 4GL Decimal to XML Schema decimal

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DECIMAL(5,3) ATTRIBUTE(XSDDecimal,XMLName="Val")
             END RECORD
<Root>
  <Val>12.345</Val>
</Root>
```

XSDDouble

Map 4GL Float to XML Schema double

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 FLOAT ATTRIBUTE(XSDDouble,XMLName="Val")
             END RECORD
<Root>
  <Val>12.78e-2</Val>
</Root>
```

XSDDuration

Map 4GL Interval to XML Schema duration

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
               val1 INTERVAL DAY TO SECOND
ATTRIBUTE(XSDDuration,XMLName="Val")
             END RECORD
<Root>
  <Val>P3DT10H30M45S</Val>
</Root>
```

XSDEntities

Map 4GL String or VarChar to XML Schema ENTITIES

XSDEntity

Map 4GL String or VarChar to XML Schema ENTITY

XSDFloat

Map 4GL Smallfloat to XML Schema float

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 SMALLFLOAT ATTRIBUTE(XSDFloat,XMLName="Val")
             END RECORD
```

```
<Root>
  <Val>126.435</Val>
</Root>
```

XSDGday

Map 4GL Datetime to XML Schema gDay

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DATETIME DAY TO DAY ATTRIBUTE(XSDGday,XMLName="Val")
             END RECORD
```

```
<Root>
  <Val>---25</Val>
</Root>
```

XSDGMonth

Map 4GL Datetime to XML Schema gMonth

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DATETIME MONTH TO MONTH
ATTRIBUTE(XSDGMonth,XMLName="Val")
             END RECORD
```

```
<Root>
  <Val>--12</Val>
</Root>
```

XSDGMonthDay

Map 4GL Datetime to XML Schema gMonthDay

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DATETIME MONTH TO DAY
ATTRIBUTE(XSDGMonthDay,XMLName="Val")
             END RECORD
<Root>
  <Val>--12-31</Val>
</Root>
```

XSDGYear

Map 4GL Datetime to XML Schema gYear

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DATETIME YEAR TO YEAR
ATTRIBUTE(XSDGYear,XMLName="Val")
             END RECORD
<Root>
  <Val>2006</Val>
</Root>
```

XSDGYearMonth

Map 4GL Datetime to XML Schema gYearMonth

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DATETIME YEAR TO MONTH
ATTRIBUTE(XSDGYearMonth,XMLName="Val")
             END RECORD
<Root>
  <Val>2006-06</Val>
</Root>
```

XSDHexBinary

Map 4GL Byte to XML Schema hexBinary

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 BYTE ATTRIBUTE(XSDHexBinary,XMLName="Val")
             END RECORD
<Root>
  <Val>0FB6</Val>
</Root>
```

XSDID

Map 4GL String or VarChar to XML Schema ID

XSDIDREF

Map 4GL String or VarChar to XML Schema IDREF

XSDIDREFS

Map 4GL String or VarChar to XML Schema IDREFS

XSDInt

Map 4GL Integer to XML Schema int

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 INTEGER ATTRIBUTE(XSDInt,XMLName="Val")
             END RECORD
<Root>
  <Val>-1258</Val>
</Root>
```

XSDInteger

Map 4GL Decimal to XML Schema integer

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DECIMAL(32,0) ATTRIBUTE(XSDInteger,XMLName="Val")
             END RECORD
<Root>
  <Val>12678</Val>
</Root>
```

XSDLanguage

Map 4GL String or VarChar to XML Schema language

XSDLong

Map 4GL Decimal to XML Schema long

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DECIMAL(19,0) ATTRIBUTE(XSDLong,XMLName="Val")
             END RECORD
<Root>
  <Val>1267488</Val>
</Root>
```

XSDNCName

Map 4GL String or VarChar to XML Schema NCName

XSDName

Map 4GL String or VarChar to XML Schema Name

XSDNegativeInteger

Map 4GL Decimal to XML Schema negativeInteger

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DECIMAL(32,0)
ATTRIBUTE(XSDNegativeInteger,XMLName="Val")
             END RECORD
<Root>
  <Val>-4828</Val>
</Root>
```

XSDNMTOKEN

Map 4GL String or VarChar to XML Schema NMToken

XSDNMTOKENS

Map 4GL String or VarChar to XML Schema NMTokens

XSDNonNegativeInteger

Map 4GL Decimal to XML Schema nonNegativeInteger

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DECIMAL(32,0)
ATTRIBUTE(XSDNonNegativeInteger,XMLName="Val")
             END RECORD
<Root>
  <Val>1589</Val>
</Root>
```

XSDNonPositiveInteger

Map 4GL Decimal to XML Schema nonPositiveInteger

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DECIMAL(32,0)
```

```
ATTRIBUTE(XSDNonPositiveInteger,XMLName="Val")
              END RECORD
<Root>
  <Val>-8574</Val>
</Root>
```

XSDNormalizedString

Map 4GL String or VarChar to XML Schema normalizedString

XSDnotation

Not supported

XSDPositiveInteger

Map 4GL Decimal to XML Schema positiveInteger

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
               val1 DECIMAL(32,0)
ATTRIBUTE(XSDPositiveInteger,XMLName="Val")
              END RECORD
<Root>
  <Val>+41893</Val>
</Root>
```

XSDQName

Map 4GL String or VarChar to XML Schema QName

XSDShort

Map 4GL Smallint to XML Schema short

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 SMALLINT ATTRIBUTE(XSDShort,XMLName="Val")
             END RECORD
<Root>
  <Val>12678</Val>
</Root>
```

XSDString

Map 4GL String, Char, Text of VarChar to XML Schema string

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 STRING ATTRIBUTE(XSDString,XMLName="Val")
             END RECORD
<Root>
  <Val>Hello world, how are you ?</Val>
</Root>
```

XSDTime

Map 4GL Datetime to XML Schema time

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DATETIME ATTRIBUTE(XSDTime,XMLName="Val")
             END RECORD
<Root>
  <Val>23:16:03.589+01:00</Val>
</Root>
```

XSDToken

Map 4GL String or VarChar to XML Schema token

XSDUnsignedByte

Map 4GL Smallint to XML Schema unsignedByte

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 SMALLINT ATTRIBUTE(XSDUnsignedByte,XMLName="Val")
             END RECORD
<Root>
  <Val>254</Val>
</Root>
```

XSDUnsignedInt

Map 4GL Decimal to XML Schema unsignedInt

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DECIMAL(32,0)
ATTRIBUTE(XSDUnsignedInt,XMLName="Val")
             END RECORD
<Root>
  <Val>1267896754</Val>
</Root>
```

XSDUnsignedLong

Map 4GL Decimal to XML Schema unsignedLong

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 DECIMAL(32,0)
ATTRIBUTE(XSDUnsignedLong,XMLName="Val")
             END RECORD
<Root>
  <Val>12678967543233</Val>
</Root>
```

XSDUnsignedShort

Map 4GL Integer to XML Schema unsignedShort

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 INTEGER ATTRIBUTE(XSDUnsignedShort,XMLName="Val")
             END RECORD
<Root>
  <Val>65535</Val>
</Root>
```

XMLOptional

Define whether the variable can be missing or not. It specifies how a 4GL NULL value is interpreted in XML.

NOTE 1 : the attribute cannot be set on a type definition
NOTE 2 : the attribute cannot be set if the main variable is not a RECORD

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 INTEGER ATTRIBUTE(XSDint,XMLName="ValOne")
              val2 FLOAT
ATTRIBUTE(XSDdouble,XMLName="ValTwo",XMLOptional)
             END RECORD
<Root>
  <ValOne>458</ValOne>
  <ValTwo>58.48</ValTwo>
</Root>
```

```
<Root>
  <ValOne>458</ValOne>
</Root>
```

XMLElement (Optional)

Map a 4GL simple datatype to an XML Element.

NOTE : the attribute cannot be set on a type definition.

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
              val1 INTEGER
ATTRIBUTE(XMLElement,XSDunsignedShort,XMLName="Val1")
              rec RECORD ATTRIBUTE(XMLName="Rec")
                val2    FLOAT   ATTRIBUTE(XMLElement,XMLName="Val2"),
                val3    STRING  ATTRIBUTE(XMLElement,XMLName="Val3")
                  END RECORD
             END RECORD
<Root>
  <Val1>148</Val1>
```

```
  <Rec1>
    <Val2>25.8</Val2>
    <Val3>Hello world</Val3>
  </Rec1>
</Root>
```

XMLAttribute

Map a 4GL simple datatype to an XML Attribute.

NOTE 1 : the attribute cannot be set on on a type definition
NOTE 2 : the attribute can only be set on a RECORD's member

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
               val1 INTEGER
ATTRIBUTE(XMLAttribute,XSDunsignedShort,XMLName="Val1")
               rec RECORD ATTRIBUTE(XMLName="Rec")
                 val2    FLOAT   ATTRIBUTE(XMLAttribute,XMLName="Val2"),
                 val3    STRING  ATTRIBUTE(XMLElement,XMLName="Val3")
                   END RECORD
             END RECORD
```

```
<Root Val1="148">
  <Rec1 Val2="25.8">
    <Val3>Hello world</Val3>
  </Rec1>
</Root>
```

XMLBase

Define the simple 4GL variable used as the base type of an XML Schema simpleContent structure.

NOTE : the attribute can be set on one and only one member of a RECORD defined with the XMLSimpleContent attribute

---

XMLAll

Map a 4GL Record to an XML Schema all structure.
The order in which the record members appear in the XML document is not significant.

Example

```
DEFINE myall RECORD ATTRIBUTE(XMLAll,XMLName="Root")
              val1  INTEGER ATTRIBUTE(XMLName="Val1"),
              val2  FLOAT   ATTRIBUTE(XMLAttribute,XMLName="Val2"),
              val3  STRING  ATTRIBUTE(XMLName="Val3")
             END RECORD
```

```
<Root Val2="25.8">
  <Val3>Hello world</Val3>
  <Val1>148</Val1>
</Root>
```

```
<Root Val2="25.8">
  <Val1>148</Val1>
  <Val3>Hello world</Val3>
</Root>
```

XMLChoice

Map a 4GL Record to an XML Schema choice structure.
The choice of the record's member is performed at runtime, and changes dynamically according to a mandatory member. This specific member must be of type SMALLINT or INTEGER, and have an XMLSelector attribute set. The XMLChoice attribute also supports a "nested" value that removes the surrounding XML tag.

NOTE 1 : valid selector values are indexes referring to members considered as XML element nodes. All other values will raise XML runtime errors.
NOTE 2 : nested choice records cannot be defined as main variables; there must always be a surrounding variable.

Example

```
DEFINE mychoice RECORD ATTRIBUTE(XMLChoice,XMLName="Root")
                val1  INTEGER   ATTRIBUTE(XMLName="Val1")
                val2  FLOAT     ATTRIBUTE(XMLAttribute,XMLName="Val2"),
                sel   SMALLINT  ATTRIBUTE(XMLSelector),
                val3  STRING    ATTRIBUTE(XMLName="Val3")
              END RECORD
```

Case where "sel" value is 4

```
<Root Val2="25.8">
  <Val3>Hello world</Val3>
</Root>
```

Case where "sel" value is 1

```
<Root Val2="25.8">
  <Val1>148</Val1>
</Root>
```

Nested example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
                val1  INTEGER   ATTRIBUTE(XMLName="Val1")
                val2  FLOAT     ATTRIBUTE(XMLAttribute,XMLName="Val2"),
                choice RECORD   ATTRIBUTE(XMLChoice="nested")
                       choice1   INTEGER
ATTRIBUTE(XMLName="ChoiceOne"),
                       choice2   FLOAT
ATTRIBUTE(XMLName="ChoiceTwo"),
                       nestedSel SMALLINT  ATTRIBUTE(XMLSelector)
                    END RECORD,
```

```
                val3  STRING    ATTRIBUTE(XMLName="Val3")
            END RECORD
```

Case where "nestedSel" value is 1

```
<Root Val2="25.8">
  <Val1>148</Val1>
  <ChoiceOne>6584</ChoiceOne>
  <Val3>Hello world</Val3>
</Root>
```

Case where "nestedSel" value is 2

```
<Root Val2="25.8">
  <Val1>148</Val1>
  <ChoiceTwo>85.8</ChoiceTwo>
  <Val3>Hello world</Val3>
</Root>
```

XMLSequence (Optional)

Map a 4GL Record to an XML Schema sequence structure.
The order in which the record members appear in the XML document must match the order of the 4GL Record.
The XMLSequence attribute also supports a "nested" value that removes the surrounding XML tag.

NOTE : nested sequence records cannot be defined as main variables; there must always be a surrounding variable.

Example

```
DEFINE mysequence RECORD ATTRIBUTE(XMLSequence,XMLName="Root")
                val1  INTEGER   ATTRIBUTE(XMLName="Val1")
                val2  FLOAT     ATTRIBUTE(XMLAttribute,XMLName="Val2"),
                val3  STRING    ATTRIBUTE(XMLName="Val3")
            END RECORD
```

```
<Root Val2="25.8">
  <Val1>-859</Val1>
  <Val3>Hello world</Val3>
</Root>
```

Nested example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
                val1  INTEGER   ATTRIBUTE(XMLName="Val1")
                val2  FLOAT     ATTRIBUTE(XMLAttribute,XMLName="Val2"),
                sequence RECORD   ATTRIBUTE(XMLSequence="nested")
                      seq1   INTEGER   ATTRIBUTE(XMLName="SeqOne"),
                      seq2   FLOAT     ATTRIBUTE(XMLName="SeqTwo"),
                   END RECORD,
                val3  STRING    ATTRIBUTE(XMLName="Val3")
            END RECORD
```

```
<Root Val2="25.8">
  <Val1>148</Val1>
  <SeqOne>6584</SeqOne>
  <SeqTwo>85.597</SeqTwo>
  <Val3>Hello world</Val3>
```

```
</Root>
```

XMLSimpleContent

Map a 4GL Record to an XML Schema simpleContent structure.

NOTE : one member must have the XMLBase attribute; all other members must have an XMLAttribute attribute. If not, the compiler complains.

Example

```
DEFINE mysimpletype RECORD ATTRIBUTE(XMLSimpleContent,XMLName="Root")
                base  STRING  ATTRIBUTE(XMLBase),
                val1  INTEGER ATTRIBUTE(XMLAttribute,XMLName="Val1"),
                val2  FLOAT   ATTRIBUTE(XMLAttribute,XMLName="Val2")
              END RECORD
<Root Val1="148" Val2="25.8">
  Hello
</Root>
```

XSComplexType

Map a 4GL Record type definition to an XML Schema complexType.

NOTE : you can have one member as a nested sequence or choice, or as an XMLList array with a nested sequence or choice as the array's elements; all other members must have an XMLAttribute attribute. If not, the compiler complains.

Example

```
TYPE mycomplextype RECORD
ATTRIBUTE(XSComplexType,XSTypeName="MyComplexType",
XSTypeNamespace="http://tempuri.org")
                name DYNAMIC ARRAY ATTRIBUTE(XMLList) OF RECORD
ATTRIBUTE(XMLSequence="nested")
                   firstname STRING ATTRIBUTE(XMLName="FirstName"),
                   lastname  STRING ATTRIBUTE(XMLName="LastName")
                 END RECORD,
                date DATE ATTRIBUTE(XMLAttribute,XMLName="Date")
              END RECORD
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://tempuri.org" elementFormDefault="qualified" >
  <xsd:complexType name="MyComplexType">
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element name="FirstName" type="xsd:string" />
```

```
      <xsd:element name="LastName"  type="xsd:string" />
    </xsd:sequence>
    <xsd:attribute name="Date" type="xsd:date" use="required"/>
  </xsd:complexType>
</xsd:schema>
```

XMLList

Map a one dimentional array to an XML Schema element that has more than one occurrence.

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
                val1  INTEGER ATTRIBUTE(XMLName="Val1"),
                list  DYNAMIC ARRAY ATTRIBUTE(XMLList) OF STRING
ATTRIBUTE(XMLName="Element"),
                val2  FLOAT   ATTRIBUTE(XMLName="Val2")
             END RECORD
```

```
<Root>
  <Val1>148</Val1>
  <Element>hello</Element>
  <Element>how</Element>
  <Element>are</Element>
  <Element>you</Element>
  <Val2>0.58</Val2>
</Root>
```

NOTE : it is not possible to define an XMLList attribute on a main array.

XMLSelector

Define the index of the candidate among all members of an XMLChoice record that will be serialized or de-serialized at runtime.
The index starts at 1.

NOTE : the selector data type must be a SMALLINT or a INTEGER.

XMLName

Define the name of a variable in an XML document.

NOTE : the attribute cannot be set on a type definition.

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root")
                val1  INTEGER ATTRIBUTE(XMLName="Val1"),
                val2  FLOAT,
                val3  INTEGER ATTRIBUTE(XMLName="Val3"),
              END RECORD
<Root>
  <Val1>148</Val1>
  <val2>0.5</val2>
  <Val3>-18547</Val3>
</Root>
```

XMLNamespace

Define the namespace of a variable in an XML document.

NOTE 1 : if the attribute is set on a Record, by default all members defined as XMLElement of that record are in the same namespace.
NOTE 2 : if the attribute is set on an Array, by default all elements defined as XMLElement of that array are in the same namespace.
NOTE 3 : the attribute cannot be set on a type definition.

Example

```
DEFINE myVar RECORD
ATTRIBUTE(XMLName="Root",XMLNamespace="http://tempuri.org")
                attr1 INTEGER ATTRIBUTE(XMLAttribute,XMLName="Attr1"),
                val1  FLOAT
ATTRIBUTE(XMLName="Val1",XMLNamespace="http://www.4js.com"),
                val2  INTEGER ATTRIBUTE(XMLName="Val2"),
                attr2 STRING
ATTRIBUTE(XMLAttribute,XMLName="Attr2",XMLNamespace="http://anyuri.org"),
              END RECORD
<fjs1:Root xmlns:fjs1="http://tempuri.org" Attr1="158"
xmlns:fjs3="http://anyuri.org" fjs3:Attr2="Hello">
  <fjs2:Val1 xmlns:fjs2="http://www.4js.com">0.5</fjs2:Val1>
  <fjs1:Val2>-18547</fjs1:Val2>
</fjs1:Root>
```

XMLType

Force the XML type name of a variable by adding xsi:type at serialization or by checking xsi:type at deserialization.

NOTE : the attribute must be used with the XMLTypenamespace attribute; otherwise, the compiler complains.

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root",XMLNamespace="http://tempuri.org")
                val1  FLOAT   ATTRIBUTE(XMLName="Val1"),
                val2  INTEGER
ATTRIBUTE(XMLName="Val2",XMLType="MyRecord",XMLTypenamespace="http://mynamespace.org")
             END RECORD
<fjs1:Root xmlns:fjs1="http://tempuri.org">
  <fjs1:Val1>0.5</fjs1:Val1>
  <fjs1:Val2 xmlns:fjs2="http://mynamespace.org" xsi:type="fjs2:MyRecord">-
18547</fjs1:Val2>
</fjs1:Root>
```

XMLTypenamespace

Force the XML type namespace of a variable by adding xsi:type at serialization or by checking xsi:type at de-serialization.

NOTE : the attribute must be used with the XMLType attribute; otherwise the compiler complains.

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root",XMLNamespace="http://tempuri.org")
                val1  FLOAT   ATTRIBUTE(XMLName="Val1"),
                val2  INTEGER
ATTRIBUTE(XMLName="Val2",XMLType="MyRecord",XMLTypenamespace="http://mynamespace.org")
             END RECORD
<fjs1:Root xmlns:fjs1="http://tempuri.org">
  <fjs1:Val1>0.5</fjs1:Val1>
  <fjs1:Val2 xmlns:fjs2="http://mynamespace.org" xsi:type="fjs2:MyRecord">-
18547</fjs1:Val2>
</fjs1:Root>
```

XSTypename

Define the XML Schema name of a 4GL type definition.

NOTE 1 : the attribute must be used with the XSTypenamespace attribute; otherwise the compiler complains.
NOTE 2 : the attribute is only allowed on a type definition.

Example

```
TYPE myType RECORD
ATTRIBUTE(XMLSequence,XSTypeName="MyFirstType",XSTypeNamespace="http://tempuri.org")
                val1  FLOAT   ATTRIBUTE(XMLElement,XMLName="Val1"),
                val2  INTEGER ATTRIBUTE(XMLElement,XMLName="Val2",XMLOptional),
                attr  STRING  ATTRIBUTE(XMLAttribute,XMLName="Attr")
            END RECORD
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://tempuri.org" elementFormDefault="qualified" >
  <xsd:complexType name="MyFirstType">
    <xsd:sequence>
      <xsd:element name="Val1" type="xsd:double" />
      <xsd:element name="Val2" type="xsd:int" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="Attr" type="xsd:string" use="required" />
  </xsd:complexType>
</xsd:schema>
```

XSTypenamespace

Define the XML Schema namespace of a 4GL type definition.

NOTE 1 : the attribute must be used with the XSType attribute; otherwise the compiler complains.
NOTE 2 : the attribute is only allowed on a type definition.

Example

```
TYPE myType RECORD
ATTRIBUTE(XMLChoice,XSTypeName="MyFirstChoice",XSTypeNamespace="http://tempuri.org")
                val1  FLOAT   ATTRIBUTE(XMLElement,XMLName="Val1"),
                val2  INTEGER ATTRIBUTE(XMLElement,XMLName="Val2",XMLOptional),
                attr  STRING  ATTRIBUTE(XMLAttribute,XMLName="Attr",XMLOptional),
                set   INTEGER ATTRIBUTE(XMLSelector)
            END RECORD
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://tempuri.org" elementFormDefault="qualified" >
  <xsd:complexType name="MyFirstChoice">
    <xsd:choice>
      <xsd:element name="Val1" type="xsd:double" />
      <xsd:element name="Val2" type="xsd:int" minOccurs="0" />
    </xsd:choice>
    <xsd:attribute name="Attr" type="xsd:string" />
  </xsd:complexType>
</xsd:schema>
```

XMLElementNamespace

Define the default namespace of all members of a record also defined as XML elements.

Example

```
DEFINE myVar RECORD
ATTRIBUTE(XMLName="Root",XMLNamespace="http://tempuri.org",
XMLElementNamespace="http://www.4js.com")
                val1  FLOAT   ATTRIBUTE(XMLElement,XMLName="Val1"),
                val2  INTEGER ATTRIBUTE(XMLElement,XMLName="Val2"),
                attr  STRING
ATTRIBUTE(XMLAttribute,XMLName="Attr"),
             END RECORD
<fjs1:Root xmlns:fjs1="http://tempuri.org" Attr="Hello"
xmlns:fjs2="http://www.4js.com">
  <fjs2:Val1>0.5</fjs2:Val1>
  <fjs2:Val2>-18547</fjs2:Val2>
</fjs1:Root>
```

XMLAttributeNamespace

Define the default namespace of all members of a record also defined as XML attributes.

Example

```
DEFINE myVar RECORD ATTRIBUTE(XMLName="Root",XMLNamespace="http://tempuri.org",
XMLAttributeNamespace="http://www.4js.com")
                val1  FLOAT   ATTRIBUTE(XMLElement,XMLName="Val1"),
                val2  INTEGER ATTRIBUTE(XMLElement,XMLName="Val2"),
                attr1 STRING  ATTRIBUTE(XMLAttribute,XMLName="Attr1"),
                attr2 DATE
ATTRIBUTE(XMLAttribute,XMLName="Attr2",XMLNamespace="http://anyuri.org"),
             END RECORD
<fjs1:Root xmlns:fjs1="http://tempuri.org" fjs2:Attr1="Hello"
xmlns:fjs2="http://www.4js.com" xmlns:fjs3=
"http://anyuri.org" fjs3:Attr2="2006-06-24">
  <fjs1:Val1>0.5</fjs1:Val1>
  <fjs1:Val2>-18547</fjs1:Val2>
</fjs1:Root>
```

XSDLength

Restrict the length of the data to the exact number of XML characters allowed when set on a 4GL STRING, VARCHAR, CHAR or TEXT, or the number of bytes allowed when set on a 4GL BYTE.

NOTE 1 : XSDMinLength and XSDMaxLength can be used together, but XSDMaxLength value must be greater then XSDMinLength
NOTE 2 : XSDMaxLength cannot be used with XSDLength

Examples

```
DEFINE myStr STRING ATTRIBUTE(XSDString,XSDLength="12",XMLName="MyString")
DEFINE myByte BYTE ATTRIBUTE(XSDBase64Binary,XSDLength="8000",XMLName="MyPicture")
```

XSDMinLength

Restrict the length of the data to the minimum number of XML characters allowed when set on a 4GL STRING, VARCHAR, CHAR or TEXT, or the number of bytes allowed when set on a 4GL BYTE.

NOTE 1 : XSDMinLength and XSDMaxLength can be used together, but XSDMaxLength value must be greater then XSDMinLength
NOTE 2 : XSDMaxLength cannot be used with XSDLength

Examples

```
DEFINE myStr STRING ATTRIBUTE(XSDString,XSDMinLength="12",XMLName="MyString")
DEFINE myByte BYTE ATTRIBUTE(XSDBase64Binary,XSDMinLength="8000",XMLName="MyPicture")
```

XSDMaxLength

Restrict the length of the data to the maximum number of XML characters allowed when set on a 4GL STRING, VARCHAR, CHAR or TEXT, or the number of bytes allowed when set on a 4GL BYTE.

NOTE 1 : XSDMinLength and XSDMaxLength can be used together, but XSDMaxLength value must be greater then XSDMinLength
NOTE 2 : XSDMaxLength cannot be used with XSDLength

Examples

```
DEFINE myStr STRING ATTRIBUTE(XSDString,XSDMaxLength="12",XMLName="MyString")
DEFINE myByte BYTE ATTRIBUTE(XSDBase64Binary,XSDMaxLength="8000",XMLName="MyPicture")
```

XSDEnumeration

Restrict the allowed value-space to a list of values separated by the characters |.

NOTE 1 : To escape the separator character, simply double it like the following ||
NOTE 2 : This attribute can be set on any simple 4GL variable excepted on XSDBoolean.

Examples

```
DEFINE myStr STRING ATTRIBUTE(XSDString,XSDEnumeration=
"one|two|three|four",XMLName="MyString")
DEFINE myDec DECIMAL(3,1) ATTRIBUTE(XSDDecimal,XSDEnumeration=
"12.1|11.8|-24.7",XMLName="MyDecimal")
```

XSDWhiteSpace

Perform a XML string manipulation before serialization or deserialization according to one of the three allowed values : **preserve**, **replace** or **collapse**.

- **preserve** : the XML string is not modified.
- **replace** : the XML string is modified by replacing each **\n**,**\t**,**\r** by a single space.
- **collapse** : the XML string is modified by replacing each **\n**,**\t**,**\r** by a single space, then each sequence of several spaces are replaced by one single space. Leading and trainling spaces are removed too.

NOTE 1 : The whiteSpace facet is always performed before any other facet constraints, or serialization or deserialization process.
NOTE 2 : For any 4GL variable excepted STRING,CHAR and VARCHAR, only collapse is allowed.

Examples

```
DEFINE myStr STRING ATTRIBUTE(XSDString,XSDWhiteSpace="replace",XMLName="MyString")
DEFINE myDec DECIMAL(3,1) ATTRIBUTE(XSDDecimal,XSDWhiteSpace=
"collapse",XMLName="MyDecimal")
```

XSDPattern

Define a regular expression the value has to match to be serialized or deserialized without any error.

NOTE 1 : The regular expression is defined in the XML Schema Part 2 specification available here.
NOTE 2 : Backslash characters '\' in a regular expression must be escaped by duplicating it.

Examples

```
DEFINE myStr STRING ATTRIBUTE(XSDString,XSDPattern="A.*Z",XMLName="MyString")
DEFINE myZipCode INTEGER ATTRIBUTE(XSDInt,XSDPattern="[0-9]{5}",XMLName="MyZipCode")
DEFINE myOtherZipCode INTEGER ATTRIBUTE(XSDInt,XSDPattern="\\d{5}",
XMLName="myOtherZipCode") # regex is \d{5} see note
```

XSDMinInclusive

Define the minimum inclusive value allowed and depending on the datatype where it is set, namely all numeric, date and time datatypes.

NOTE : The minimum value cannot exceed the implicit minimum value supported by the datatype itself or the compiler will complain. For instance, with XSDShort the minimum value is -32768.

Examples

```
DEFINE myCode SMALLINT ATTRIBUTE(XSDShort,XSDMinInclusive="-1000",XMLName="MyCode")
DEFINE myRate DECIMAL(4,2) ATTRIBUTE(XSDDecimal,XSDMinInclusive="100.01",
XMLName="MyRate")
```

XSDMaxInclusive

Define the maximum inclusive value allowed and depending on the datatype where it is set, namely all numeric, date and time datatypes.

NOTE : The maximum value cannot exceed the implicit maximum value supported by the datatype itself or the compiler will complain. For instance, with XSDShort the maximum value is 32767.

Examples

```
DEFINE myCode SMALLINT ATTRIBUTE(XSDShort,XSDMaxInclusive="1000",XMLName="MyCode")
DEFINE myRate DECIMAL(4,2) ATTRIBUTE(XSDDecimal,XSDMaxInclusive="299.99",
XMLName="MyRate")
```

XSDMinExclusive

Define the minimum exclusive value allowed and depending on the datatype where it is set, namely all numeric, date and time datatypes.

NOTE : The minimum value cannot exceed or be equal to the implicit minimum value supported by the datatype itself or the compiler will complain. For instance, with XSDShort the minimum value is -32768.

Examples

```
DEFINE myCode SMALLINT ATTRIBUTE(XSDShort,XSDMinExclusive="-1000",
XMLName="MyCode")
DEFINE myRate DECIMAL(4,2) ATTRIBUTE(XSDDecimal,XSDMinExclusive=
"100.01",XMLName="MyRate")
```

XSDMaxExclusive

Define the maximum exclusive value allowed and depending on the datatype where it is set, namely all numeric, date and time datatypes.

NOTE : The maximum value cannot exceed or be equal to the implicit maximum value supported by the datatype itself or the compiler will complain. For instance, with XSDShort the maximum value is 32767.

Examples

```
DEFINE myCode SMALLINT ATTRIBUTE(XSDShort,XSDMaxExclusive="1000",
XMLName="MyCode")
DEFINE myRate DECIMAL(4,2) ATTRIBUTE(XSDDecimal,XSDMaxExclusive="299.99",
XMLName="MyRate")
```

XSDTotalDigits

Define the maximum number of digits allowed on a numeric datatype, fraction part inclusive if there is one.

NOTE 1 : The total digits value cannot be equal or lower then 0. NOTE 2 : On a 4GL decimal, the total digits value cannot be lower than the precision of the 4GL decimal itself. NOTE 3 : Notice that a decimal without any precision and scale value is a decimal(16), therfore the total digits value must be equal or greater than 16.

Examples

```
DEFINE myCode SMALLINT ATTRIBUTE(XSDShort,XSDTotalDigits="4",XSDMaxExclusive=
"1000",XMLName="MyCode")
```

```
DEFINE myRate DECIMAL(4,2)
ATTRIBUTE(XSDDecimal,XSDTotalDigits="5",XSDMaxExclusive="299.99",XMLName="MyRate")
```

XSDFractionDigits

Define the maximum number of digits allowed on the fraction part of a numeric datatype.

NOTE 1 : The fraction digits value set on a 4GL datatype without XSDDecimal set, can only be 0.
NOTE 2 : On a 4GL decimal, the fraction digits value cannot be lower than the scale of the 4GL decimal itself, and must be lower than the XSDTotalDigits value if set.

Examples

```
DEFINE myCode SMALLINT ATTRIBUTE(XSDShort,XSDFractionDigits="0",
XSDMaxExclusive="1000",XMLName="MyCode")
DEFINE myRate DECIMAL(4,2)
ATTRIBUTE(XSDDecimal,XSDTotalDigits="5",XSDFractionDigits="3",
XSDMaxExclusive="299.99",XMLName="MyRate")
```

The fglwsdl tool (WSDL and XSD)

To access a remote Web Service, you first must get the WSDL information from the service provider. Sample services can be found through UDDI registries (http://www.uddi.org), or on other sites such as XMethods (http://www.xmethods.net).

Using fglwsdl

The **fglwsdl** tool, provided as part of the Genero Web Services Extension (GWS), allows you to get the WSDL description of any Web Service that will be accessed by a GWS Client application, or to use a WSDL description when you are creating a corresponding GWS Server application. It also allows you to generate 4GL data types from XML schemas (also known as XSD).

Syntax:

```
fglwsdl [-command] <filename | url | regex value>
```

where:

- *command* is one of the command-line options
- *filename* is the name of a WSDL description file for a Web Service or a XML schema file
  or
  url is the web location of a WSDL description for a published Web Service or the location of an XSD schema resource on the web. For example, provide the host server and the name of the service: `http://<host-server>/<name-of-the-service>?WSDL`

Command Line Options:

Command	Description
-V	Display version information

-h	Display this help
-l	List services from a WSDL or variables from a XSD
-c [*options*]	Generate client stub (default) to be used in a GWS Client application
-s [*options*]	Generate server stub to be used in a GWS Server application
-x [*options*]	Generate data types from a XML schema (XSD)

The options for the generation of the client or server stub are listed below in the specific WSDL options and in the Common options. These options can also be used without -c, if you are generating a client stub (the default).

The options for the generation of data types are listed below in the specific XSD options and in the Common options. The -x option cannot be used with -c or -s.

There is also an extra command to validate a regular expression against a value as described in the specific Extra command.

Extra command	Description
-regex	Validate the *value* against the *regex* regular expression described in XML Schema specification

WSDL Options	Description
-o *file*	Specify a Base name for the output files for one service
-n *service port*	Generate only for the given service name and portType
-prefix *name*	Add *name* as the prefix of the generated web service functions, variables and types
-compatibility	Generate a Genero 1.xx compatibility client stub
-fRPC	Force RPC convention; use RPC Convention to generate the code, regardless of what the WSDL information contains.
-disk	Save WSDL and all dependencies from an URL on the disk. **Notice** that to generate code in the same time you must use option -c, -s or both, otherwise no code is generated.

XSD Options	Description
-o *file*	Name of the output file - if *file* has no extension, .inc is added
-n *name [ns]*	Generate only for the given variable name and namespace if there is one
-prefix	Add *name* as the prefix of the generated data types

name

-disk	Save XSD and all dependencies from an URL on the disk. **Notice** that no code is generated.

Common Options	Description
-comment	Add XML comments to the generation
-noHTTP	Disable HTTP - search for the WSDL description or the XML schema and its dependencies on the client instead of the internet, for example if a company has restricted access to the internet
-proxy *location*	Connect via proxy where *location* is host[:port] or ip[:port]
-fArray	Force XML array generation instead of XML list when possible - if the WSDL contains an XML definition of a 4GL list, generate a 4GL array matching the same definition
-fInheritance	Force generation of XML choice records for all inheritance types found in the schemas, otherwise only for abstract types and elements
-noFacets	Don't generate facet constraints restricting the value-space of simple datatype
-ext *schema*	Add an external schema - external schemas for dependencies won't be included in the WSDL description or in the XSD schema if their location attributes are missing. Use this option to add a missing external schema for a WSDL or XSD dependency.
-noValidation	Disable XML schema validation warnings.

Generating files for a GWS Client

Using the **fglwsdl** tool, you can obtain the WSDL information for a GWS Client application. The following example requests the Calculator Web Service information from the specified URL, and the output files will have the base name ws_calculator:

```
fglwsdl -o ws_calculator http://localhost:8090/Calculator?WSDL
```

For a client application, **fglwsdl** generates two output files, which should not be modified:

- <***filename***>**.inc** - the globals file, containing declarations of global variables that can be used as input or output for functions accessing Web Service operations, and the global wsError record. In our example, the file is **ws_calculator.inc.**

This file must be listed in a GLOBALS statement at the top of any .4gl modules that you write for your GWS Client application.

- **<filename>.4gl -** containing the definitions of the functions that can be used in your GWS client application to perform the requested Web Service operation, and the code that manages the Web Service request. In our example, the file is **ws_calculator.4gl**.

  This file must be compiled and linked into your GWS Client application.

The Web Service error structure

Check this structure, defined in the globals **.inc** file, for a detailed error description when a Web Service function returns with a non-zero status.
Notice that even if status is -2 due to asynchronous calls, the record contains a description.

```
DEFINE wsError RECORD
   code STRING,          -- short description of the error
   codeNS STRING,        -- the namespace of the error code
   description STRING,   -- description of the error
   action STRING         -- internal "SOAP action"
END RECORD
```

The generated functions

Genero Web Services client functions have the following requirements:

- The function cannot have input parameters.
- The function cannot have return values.
- The function's input message must be defined as a global or module RECORD.
- The function's output message must be defined as a global or module RECORD.

As a result, two types of GWS functions are generated for the Web Service operation that you requested:

- One function type uses global records for the input and output. The names of these functions end in "_g". Before calling the function in your GWS Client application, you must set the values in the global input record. After the function call, the status of the request is returned from the server, and the output message is stored in the global output record. In addition to performing the desired operation, this function handles the communication for the SOAP request and response, and sets the values in the wsError record as needed.
- The other function type serves as a "wrapper" for the "_g" function. It passes the values of input parameters to the "_g" function, and returns the output values and status received from the "_g" function. Your client application does not need to

directly access the global records. This function can only be used if the parameters are simple variables (no records or arrays).

The generated **.inc** globals file contains comments that list the prototypes of the functions for the GWS operation, and the definitions of the global INPUT and OUTPUT records.

Example output

The example Web Service for which the WSDL information was requested, Calculator, has an Add operation that returns the sum of two integers.

The generated file **ws_calculator.inc** lists the prototype for the **Add** and **Add_g** functions, the asynchronous **AddRequest_g** and **AddResponse_g** functions, as well as the definitions of the global variables **Add** and **AddResponse**:

```
# Operation: Add
#
# FUNCTION: Add_g()             -- Function that uses the global input
and output records
#   RETURNING: soapStatus       -- An integer where 0 represents
success
#   INPUT: GLOBAL Add
#   OUTPUT: GLOBAL AddResponse
#
# FUNCTION: Add(p_a, p_b)       -- Function with input parameters that
correspond
#  RETURNING: soapStatus ,p_r   -- to the a and b variables of the
global
                                -- INPUT record
                                -- Return values are the status integer
and the value
                                -- in the r variable of the global
OUTPUT record
#
# FUNCTION: AddRequest_g()      -- Asynchronous function that uses the
global input record
#  RETURNING: soapStatus        -- An integer where 0 represents
success, -1 error
#   INPUT: GLOBAL Add           --  and -2 means that a previous
request was sent and that a response is in progress.
#
# FUNCTION: AddResponse_g()     -- Asynchronous function that uses the
global output record
#  RETURNING: soapStatus        -- An integer where 0 represents
success, -1 error
#   OUTPUT: GLOBAL AddResponse  --  and -2 means that the response was
not yet received, and that a new call should be done later.
#VARIABLE : Add                 -- defines the global INPUT record
DEFINE Add RECORD ATTRIBUTE(XMLName="Add",
                            XMLNamespace="http://tempuri.org/")
         a INTEGER ATTRIBUTE(XMLName="a",XMLNamespace=""),
         b INTEGER ATTRIBUTE(XMLName="b",XMLNamespace="")
       END RECORD
```

```
# VARIABLE : AddResponse       -- defines the global OUTPUT record
DEFINE AddResponse RECORD ATTRIBUTE(XMLName="AddResponse",
                                    XMLNamespace="http://tempuri.org/")
         r INTEGER ATTRIBUTE(XMLName="r",XMLNamespace="")
       END RECORD
```

Using the generated functions

The information obtained from the **ws_calculator.inc** file allows you to write code in your own .4gl module as part of the Client application, using the Web Service operation Add.

- **Using parameters and return values**

  Since the input variables for our example are simple integers, you can call the **Add** function in your Client application, defining variables for the parameters and return values.

```
FUNCTION myWScall()
  DEFINE op1        INTEGER
  DEFINE op2        INTEGER
  DEFINE result     INTEGER
  DEFINE wsstatus   INTEGER
  ...
  LET op1 = 6
  LET op2 = 8
  CALL Add(op1, op2) RETURNING wsstatus, result
  ...
  DISPLAY result
```

- **Using global records**

  You could choose to call the **Add_g** function instead, using the global records **Add** and **AddResponse** directly. If the input variables are complex structures like records or arrays, you are required to use this function.

```
FUNCTION myWScall()
  DEFINE wsstatus INTEGER
  ...
  LET Add.a = 6
  LET Add.b = 8
  LET wsstatus = Add_g()
  ...
  DISPLAY AddResponse.r
```

  In this case, the status is returned by the function, which has also put the result in the **AddResponse** global record.

See Tutorial: Writing a Client Application for more information. The **demo/WebServices** subdirectory of your Genero installation directory contains complete examples of Client Applications.

- **Using asynchonous calls**

  If you don't want your application to be blocked when waiting for the response to a request, you should first call **AddRequest_g;** this will send the request using the global **Add** record to the server. It returns a status of 0 if everything goes well, -1 in case of error, or -2 if you tried to resend a new request before the previous response was retrieved.

```
FUNCTION sendMyWScall()
  DEFINE wsstatus INTEGER
  ...
  LET Add.a = 6
  LET Add.b = 8
  LET wsstatus = AddRequest_g()
  IF wstatus <> 0 THEN
    DISPLAY "ERROR :", wsError.code
  END IF
  ...
```

  Then you can call the **AddResponse_g** to retrieve the response in the **AddResponse** global record of the previous request. If returned status is 0 the response was successfully received, -1 means that there was an error, and -2 means that the response was not yet received and that the function should be called later.

```
FUNCTION retrieveMyWScall()
  DEFINE wsstatus INTEGER
  ...
  LET wsstatus = AddResponse_g()
  CASE wstatus
    WHEN -2
      DISPLAY "No response available, try later"
    WHEN 0
      DISPLAY "Response is :",AddResponse.r
    OTHERWISE
      DISPLAY "ERROR :", wsError.code
  END CASE
  ...
```

  Note that you can mix the asynchronous call with the synchronous one as they are using two different requests. In other words, you can perform an asynchronous request with **AddRequest_g**, then a synchronous call with **Add_g**, and then retrieve the response of the previous asynchonous request with **AddResponse_g**.

Warning: In development mode, a single 4GL Web Service server can only handle one request at a time, and several asynchronous requests in a row without retrieving the corresponding response will lead to a deadlock. To support several asynchronous requests in a row, it is recommended that you are in deployment mode with a GAS as the front end.

Generating files for a GWS Server

You can completely write a GWS Server application for a Web Service that you have created; see Tutorial: Writing a Server Application. However, if you want to make sure your Web Service is compatible with that of a third-party (an accounting application vendor, for example), you can use the **fglwsdl** tool to obtain the WSDL information that complies with that vendor's standards, and to generate corresponding files that can be used in your GWS Server application.

The following example requests the Calculator Web Service information from the specified URL, and the output files will have the base name "ws_calculator".

```
fglwsdl -s -o ws_calculator http://localhost:8090/Calculator?WSDL
```

For a server application, fglwsdl generates two files, which should not be modified:

- ***<filename>*.inc** - the globals file, containing declarations of global variables that can be used as input or output to functions accessing the Web Service operations. In our example, the file is **ws_calculatorService.inc.**

  This file must be listed in a GLOBALS statement at the top of any .4gl modules that you write for your GWS Server application.

- **<filename>.4gl -** containing a function that creates the service described in the WSDL, publishes the operations of the service, and registers the service. In our example, the file is **ws_calculatorService.4gl**.

  This file must be compiled and linked into your GWS Server application.

Example output

In the generated file **ws_calculatorService.inc**, the definitions of the variables for the input and output record are the same as that generated for the Web Service Client application:

```
#VARIABLE : Add                    -- defines the global INPUT record
DEFINE Add RECORD ATTRIBUTE(XMLName="Add",
                            XMLNamespace="http://tempuri.org/")
         a INTEGER ATTRIBUTE(XMLName="a",XMLNamespace=""),
```

```
        b INTEGER ATTRIBUTE(XMLName="b",XMLNamespace="")
      END RECORD
# VARIABLE : AddResponse        -- defines the global OUTPUT record
DEFINE AddResponse RECORD ATTRIBUTE(XMLName="AddResponse",
                                    XMLNamespace="http://tempuri.org/")
        r INTEGER ATTRIBUTE(XMLName="r",XMLNamespace="")
      END RECORD
```

The generated file **ws_calculatorService.4gl** contains a single function that creates the Calculator service, creates and publishes the service operations, and registers the Calculator service:

```
FUNCTION Createws_calculatorService()
  DEFINE service com.WebService
  DEFINE operation com.WebOperation
...
# Create Web Service
  LET service =
com.WebService.CreateWebService("Calculator","http://tempuri.org/")

# Publish Operation : Add
  LET operation =
com.WebOperation.CreateRPCStyle("Add","Add",Add,AddResponse)
  CALL service.publishOperation(operation,"")
...
# Register Service
  CALL com.WebServiceEngine.RegisterService(service)
  RETURN 0
...
END FUNCTION
```

Writing your functions

The ws_calculator.inc file provides you with the global input and output records and function names that allow you to write your own code implementing the Add operation. Your new code should not be written in the generated modules. For example, do not add your own version of the Add function to the generated ws_calculator.4gl module; it can be included in your module containing the MAIN program block, or in a separate module to be included as part of the Web server application. The function must use the generated definitions for the global input and output records.

In your version of the operation, this function adds 100 to the sum of the variables in the input record:

```
      FUNCTION Add()
          LET AddResponse.r = (Add.a + Add.b) * 100
      END FUNCTION
```

See Tutorial: Writing a Server application for more information. The **demo/WebServices** subdirectory of your Genero installation directory contains complete examples of Server Applications.

Error Messages

The following table lists all Genero Web Services Extension error messages by number and name, provides a description of each error message, and identifies a solution for the associated problem.

Number	Name	Message	Solution
-15500	INTERNAL_SERVER _ERROR	Internal runtime error occurred in WS server program	Contact your support center.
-15501	FUNCTION_ERROR	Cannot create WS operation because the given function is not defined.	Verify that the name of the BDL function of fgl_ws_server_publishFunction() is correct.
-15502	FUNCTION_ DECLARATION _ERROR	Invalid WS-function declaration, no parameters allowed.	Verify that the BDL function has no input and no output parameters.
-15503	FUNCTION_ ALREADY_EXISTS	Operation name is already used in the current web service.	You must change the name of the Web-Function operation in the function fgl_ws_server_publishFunction().
-15504	PORT_ALREADY _USED	WS server port already used by another application.	You must change the port number in the function fgl_ws_server_start().
-15505	BDL_XML_ERROR	Some BDL data types are not supported by XML.	Verify that all exposed functions don't contain one of the following data types: • DATETIME beginning with MINUTE • DATETIME beginning with SECOND • INTERVAL beginning with YEAR and/or MONTH
-15506	INTERNAL_CLIENT _ERROR	Internal runtime error occurred in WS client program.	Contact your support center.

-15507	ALL_HANDLE _ALLOCATED	All WS operation handlers already allocated.	You must free the unused client handler with the fgl_ws_client_freeMethod() function.
-15508	HANDLE_NOT _ALLOCATED	WS operation handle not allocated.	You must first call the fgl_ws_client_createMethod() function to get a valid handle.
-15509	BAD_INOUT _PARAMETER	Invalid WS input/output parameter specification.	Only 1 or 2 is allowed, all other values are incorrect.
-15510	VARIABLE_PATH _ERROR	Invalid WS variable path specification.	Verify that the variable path corresponds to the input or output record message.
-15511	INVALID_OPTION _NAME	Invalid fgl_ws_set/getOption() parameter.	Verify that the option flag of the fgl_ws_set/getOption() function exists.
-15512	INPUT_VARIABLE _ERROR	WS input record not defined.	Verify that the name of the input record on the fgl_ws_server_publishFunction() exists.
-15513	OUTPUT_VARIABLE _ERROR	WS output record not defined.	Verify that the name of the output record on the fgl_ws_server_publishFunction() exists.
-15514	PORT_NOT _NUMERIC	The port value from the FGLAPPSERVER environment variable or from the parameter of the fgl_ws_server_start() function is not a numeric one.	Verify that the port value contains only digits.
-15515	NO_AS_FOUND	No application server has been started at specified host.	Verify that FGLAPPSERVER contains the right host and port where the application server is listening.
-15516	LICENSE_ERROR	No more licenses available.	Contact your support center.

-15517	BAD_RUNNER _VERSION	Current runner version not compatible with the Web Services Extension.	Install the right version of the Genero BDL.
-15518	INPUT_NAMESPACE _MISSING	The input namespace of your Web function is missing.	Add a valid input namespace in fgl_ws_publishFunction().
-15519	OUTPUT _NAMESPACE _MISSING	The output namespace of your Web function is missing.	Add a valid output namespace in fgl_ws_publishFunction()
-15520	CFG_SECURITY_ID _FAILED	Cannot load a certificate or private key file.	Verify that each *ws.ident.security* FGLPROFILE entries contain a valid security identifier.
-15521	CFG_SECURITY _WINID _FAILED	Cannot find a certificate in the Windows key store.	Verify that each *ws.ident.security* FGLPROFILE entries contain a valid Windows security identifier.
-15522	CFG_SECURITY _CA _FAILED	Cannot load the Certificate Authorities file.	Verify that the *security.global.ca* FGLPROFILE entry contains the correct Certificate Authorities filename.
-15523	CFG_SECURITY_ WINCA_FAILED	Cannot create the Certificate Authorities from the Windows key store.	Verify that you have enough rights to access the Windows key store.
-15524	CFG_SECURITY_ CIPHER_FAILED	Cannot set the cipher list.	Verify that all ciphers in the list are valid ones and supported by openssl.
-15525	CFG_PROXY _HTTP _FAILED	Unable to reach the HTTP proxy.	Verify that the *proxy.http.location* FGLPROFILE entry contains the correct HTTP proxy address.
-15526	CFG_PROXY _HTTPS _FAILED	Unable to reach the HTTPS proxy.	Verify that the *proxy.https.location* FGLPROFILE entry contains the correct HTTPS proxy address.
-15527	CFG_PROXY _HTTP _AUTH_UNKNOWN	Unknown HTTP proxy authenticate identifier.	Verify that the *proxy.http.authenticate* FGLPROFILE entry contains a valid HTTP authenticate

			identifier.
-15528	CFG_PROXY _HTTPS _AUTH _UNKNOWN	Unknown HTTPS proxy authenticate identifier.	Verify that the *proxy.https.authenticate* FGLPROFILE entry contains a valid HTTP authenticate identifier.
-15529	CFG_AUTH_ CREATE_FAILED	Cannot create a HTTP authenticate configuration.	Verify that all authenticate logins and passwords are correctly set.
-15530	CFG_AUTH_ ENCRYPTED _CREATE_FAILED	Cannot create an encrypted HTTP authenticate configuration.	Verify that all authenticate logins and encrypted passwords are correctly set.
-15531	CFG_SERVER _CREATION _FAILED	Cannot create a server configuration.	Verify that all *ws.ident.url* FGLPROFILE entries are correctly set.
-15532	CFG_SERVER _SECURITY _UNKNOWN	Unknown server configuration security identifier.	Verify that all *ws.ident.security* FGLPROFILE entries contain a valid Security identifier.
-15533	CFG_SERVER _AUTH _UNKNOWN	Unknown server configuration authenticate identifier.	Verify that all *ws.ident.authenticate* FGLPROFILE entries contain a valid HTTP Authenticate identifier.

Server API Functions - version 1.3 only

The following table lists the APIs to create a Web Services server in BDL.

Note: These functions are valid for backwards compatibility, but they are not the preferred way to handle Genero Web Services. See the GWS COM Library classes and methods.

Function	Description
fgl_ws_server_setNamespace()	Defines the namespace of the service on the Web.
fgl_ws_server_start()	Creates and starts the Web Service server.
fgl_ws_server_publishFunction()	Publishes the BDL function as a Web Function.
fgl_ws_server_generateWSDL()	Generates the WSDL file.
fgl_ws_server_process()	Waits for and processes incoming SOAP requests.
fgl_ws_server_setFault()	Sets the SOAP fault string for a Web Function.
fgl_ws_server_getFault()	Retrieves the fault string that was set for a Web Function, for testing purposes.

---

fgl_ws_server_setNamespace() (version 1.3)

Purpose:

This function defines the namespace of the service on the Web and must be called first, before all other functions of the API.

Syntax:

```
FUNCTION fgl_ws_server_setNamespace(namespace VARCHAR)
```

Parameters:

1. *namespace* is the name of the namespace.

Return values:

None

Example:

```
01 CALL fgl_ws_server_setNamespace("http://tempuri.org/")
```

---

fgl_ws_server_start() (version 1.3)

Purpose:

This function creates and starts the server. For development or testing purposes, you may start a Web Service server as a single server where only one request at a time will be able to be processed. For deployment, you may start a Web Service server with an application server able to handle several connections at one time using a load-balancing algorithm. The value of the parameter passed to the function determines which method is used.

Syntax:

```
FUNCTION fgl_ws_server_start(tcpPort VARCHAR)
```

Parameters:

1. *tcpPort* is a string representing either:

    - the **socket port number** (for a single Web Service server), or
    - the **host** and **port** value separated by a colon (for a Web Service server connecting to an application server). The value of **port** is an offset beginning at 6400.

    Note: If the **FGLAPPSERVER** environment variable is set, the tcpPort value is ignored, and replaced by the value of FGLAPPSERVER.

Return values:

None

Examples:

To start a standalone Web Service server:

```
01 CALL fgl_ws_server_start("8080") # A single Server is listening
02                                  # on port number: 8080
```

To start a Web Service server attempting to connect to an application server:

```
01 CALL fgl_ws_server_start("zeus:5") # The server attempt to connect
02                                     # to an application server located
03                                     # on host zeus and listening
04                                     # on the port number 6405
```

Possible runtime errors:

- PORT_ALREADY_USED
- PORT_NOT_NUMERIC
- NO_AS_FOUND
- LICENSE_ERROR

---

fgl_ws_server_publishFunction() (version 1.3)

Purpose:

This function publishes the given BDL function as a Web-Function on the Web.

Syntax:

```
FUNCTION fgl_ws_server_publishFunction(operationName VARCHAR,
   inputNamespace VARCHAR, inputRecordName VARCHAR,
   outputNamespace VARCHAR,outputRecord VARCHAR,
   functionName VARCHAR)
```

Parameters:

1. *operationName* is the name by which the operation will be defined on the Web. The name is case sensitive.
2. *inputNamespace* is the namespace of the incoming operation message.

3. *inputRecordName* is the name of the BDL record representing the Web Function input message or "" if there is none.
4. *outputNamespace* is the namespace of the outgoing operation message.
5. *outputRecord* is the name of the BDL record representing the Web Function output message or "" if there is none.
6. *functionName* is the name of the BDL function that is executed when the Web Service engine receives a request with the operation name defined above.

Return values:

None

Example:

```
01 CALL fgl_ws_server_publishFunction(
02 "MyWebOperation",
03 "http://www.tempuri.org/webservices/","myfunction_input",
04 "http://www.tempuri.org/webservices/","myfunction_output",
05 "my_bdl_function")
```

Possible runtime errors:

- FUNCTION_ALREADY_EXISTS
- FUNCTION_ERROR
- FUNCTION_DECLARATION_ERROR
- INPUT_VARIABLE_ERROR
- OUTPUT_VARIABLE_ERROR
- BDL_XML_ERROR
- INPUT_NAMESPACE_MISSING
- OUTPUT_NAMESPACE_MISSING

---

fgl_ws_server_generateWSDL() (version 1.3)

Purpose:

This function generates the WSDL file according to the BDL-server program.

Syntax:

```
FUNCTION fgl_ws_server_generateWSDL(serviceName VARCHAR,
   serviceLocation VARCHAR, fileName VARCHAR)
RETURNING resultStatus INTEGER
```

Parameters:

1. *serviceName* is the name of the web service.
2. *serviceLocation* is the URL of the server.
3. *fileName* is the name of the file that will be generated.

Return value:

1. *resultStatus* is a status containing:

    - 0 if the file has been correctly generated.
    - Any other values if the operation has failed.

Example:

```
01 DEFINE mystatus INTEGER
02
03 LET mystatus=fgl_ws_server_generateWSDL(
04 "CustomerService",
05 "http://localhost:8080",
06 "C:/mydirectory/myfile.wsdl")
07
08 IF mystatus=0 THEN
09   DISPLAY "Generation of WSDL done..."
10 ELSE
11   DISPLAY "Generation of WSDL failed!"
12 END IF
```

---

fgl_ws_server_process() (version 1.3)

Purpose:

This function waits for an incoming SOAP request for a given time (in seconds) and then processes the request, or returns, if there has been no request during the given time. If a DEFER INTERRUPT or DEFER QUIT instruction has been defined, the function returns even if it is an infinite wait.

Syntax:

```
FUNCTION fgl_ws_server_process(timeout INTEGER)
   RETURNING resultStatus INTEGER
```

Parameter:

1. *timeout* is the maximum waiting time for an incoming request (or -1 for an infinite wait)

Return value:

1. *resultStatus* is a status containing:

    - 0 — Request has been processed
    - -1 — Timeout has been reached
    - -2 — The application server asks the runner to shutdown
    - -3 — A client connection has been unexpectedly broken
    - -4 — An interruption has been raised
    - -5 — The HTTP header of the request was incorrect
    - -6 — The SOAP envelope was malformed
    - -7 — The XML document was malformed

Example:

```
01 DEFER INTERRUPT
02 DEFINE mystatus  INTEGER
03 LET mystatus=fgl_ws_server_process(5)# wait for 5 seconds
04                                        # for incoming request
05 IF mystatus=0 THEN
06   DISPLAY "Request processed."
07 END IF
08 IF mystatus=-1 THEN
09   DISPLAY "No request."
10 END IF
11 IF mystatus=-2 THEN # terminate the application properly
12   EXIT PROGRAM       # if connected to application server
13 END IF
14 IF  mystatus=-3 THEN
15   DISPLAY "Client connection unexpectedly broken."
16 END IF
17 IF mystatus=-4 THEN
18   DISPLAY "Server process has been interrupted."
19 END IF
20 IF mystatus=-5 THEN
21   DISPLAY "Malformed or bad HTTP request received."
22 END IF
23 IF int_flag<>0 THEN
24   LET int_flag=0
25   EXIT PROGRAM
26 END IF
```

---

fgl_ws_server_setFault() (version 1.3)

Purpose:

This function can be called in a published Web-Function in order to return a SOAP fault string to the client at the end of the function's execution.

Syntax:

```
FUNCTION fgl_ws_server_setFault(faultMessage VARCHAR)
```

Parameter:

1. *faultMessage* is a string containing the SOAP Fault string that will be returned to the client.

Return values:

None

Example:

```
01 CALL fgl_ws_server_setFault(
    "The server is not able to manage this request.")
```

---

fgl_ws_server_getFault() (version 1.3)

Purpose:

This function retrieves the last fault string the user has set in a Web-Function, or an empty string if there is none.

Note: This function is only for testing the Web Services functions before they are published on the Web.

Syntax:

```
FUNCTION fgl_ws_server_getFault()
    RETURNING faultMessage VARCHAR
```

Parameters:

None

Return value:

1. *faultMessage* is the string containing the SOAP Fault string.

Example:

```
01 DEFINE div_input RECORD
```

```
02                    a INTEGER,
03                    b INTEGER
04                    END RECORD
05
06 DEFINE div_output RECORD
07                    result INTEGER
08                    END RECORD
09
10 FUNCTION TestServices()
11    DEFINE string VARCHAR(100)
12    ...
13    # Test divide by zero operation
14    LET div_input.a=15
15    LET div_input.b=0
16    CALL service_operation_div()
17    LET string=fgl_ws_server_getFault()
18    DISPLAY "Operation div error: ", string
19    ...
20 END FUNCTION
21
22 FUNCTION service_operation_div()
23    ...
24    IF div_input.b = 0 THEN
25      CALL fgl_ws_server_setFault("Divide by zero")
26      RETURN
27    END IF
28    ...
29 END FUNCTION
```

Configuration API Functions - version 1.3 only

The following table lists those configuration API functions that can modify the behavior of the Web Services engine for the client as well as for the server.

Note: These functions are valid for backwards compatibility, but they are not the preferred way to handle Genero Web Services. See the GWS Com Extension Library classes and methods.

Function	Description
fgl_ws_setOption()	Sets an option flag with a given value.
fgl_ws_getOption()	Returns the value of an option flag.

---

fgl_ws_setOption()

Purpose:

This function sets an option flag with a given value, changing the global behavior of the Web Services engine.

Syntax:

```
FUNCTION fgl_ws_setOption(optionName VARCHAR,
         optionValue INTEGER)
```

Parameters:

1. *optionName* is one of the global flags.
2. *optionValue* is the value of the flag.

Return values:

None

Possible runtime error:

INVALID_OPTION_NAME

EXample:

```
01 CALL fgl_ws_setOption("http_invoketimeout",5)
```

---

fgl_ws_getOption()

Purpose:

This function returns the value of an option flag.

Syntax:

```
FUNCTION fgl_ws_getOption(optionName VARCHAR)
     RETURNING optionValue INTEGER
```

Parameter:

1. *optionName* is one of the global flags.

Return value:

1. *optionValue* is the value of the flag.

Possible runtime error:

INVALID_OPTION_NAME

Example:

```
01 DEFINE value INTEGER
02 LET value=fgl_ws_getOption("http_invoketimeout")
```

Option Flags

Flags	Client or Server	Commentary
http_invoketimeout	Client	Defines the maximum time in seconds a client has to wait before the client connection raises an error because the server is not responding. **Note:** A value of -1 means that it has to wait until the

		server responds (the default value is -1).
tcp_connectiontimeout	Client	Defines the maximum time in seconds a client has to wait for the establishment of a TCP connection with a server. **Note:** A value of -1 means infinite wait (the default value is 30 seconds except for Windows, where it is 5 seconds).
soap_ignoretimezone	Both	Defines if, during the marshalling and unmarshalling process of a BDL DATETIME data type, the SOAP engine should ignore the time zone information. **Note:** A value of zero means false (the default value is false).
soap_usetypedefinition	Both	Defines if the Web Services engine must specify the type of data in all SOAP requests. (This will add an "xsi:type" attribute to each parameter of the request.) **Note:** A value of zero means false (the default value is false).
wsdl_decimalsize	Server	Defines if, during the WSDL generation, the precision and scale of a DECIMAL variable will be taken into account. See Notes on WSDL Generation Options below. **Note:** A value of zero means false (the default value is true).
wsdl_arraysize	Server	Defines if, during the WSDL generation, the size of a BDL array will be taken into account. See Notes on WSDL Generation Options below. **Note:** A value of zero means false (the default value is true).
wsdl_stringsize	Server	Defines if, during the WSDL generation, the size of a CHAR or VARCHAR variable will be taken into account. See Notes on WSDL Generation Options below. **Note:** A value of zero means false (the default value is true).

Notes on WSDL Generation Options

1. For a BDL type **DECIMAL(5,2)**, when "**wsdl_decimalsize**" is TRUE, the generated WSDL file contains the total size and the size of the fractional part of the decimal:

```
<types>
   <schema xmlns="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.4js.com/types/">
      <simpleType name="echoDecimal5_2_a_dec5_2_out_FGLDecimal">
         <restriction base="decimal">
            <totalDigits value="5" />
            <fractionDigits value="2" />
         </restriction>
      </simpleType>
   </schema>
</types>
<message name="echoDecimal5_2">
  <part name="dec5_2" type="f:echoDecimal5_2_a_dec5_2_in_FGLDecimal" />
</message>
```

When "**wsdl_decimalsize**" is FALSE, the total size and the size of the fractional part are not mentioned:

```
<message name="echoDecimal5_2">
  <part name="dec5_2" type="xsd:decimal" />
</message>
```

2. If the WSDL file does not contain the size, the client application has no way of knowing the size. In this scenario, a default value for the size is generated. For example, the exported server type **DECIMAL(5,2)** becomes a **DECIMAL(32)** on the client side.

3. It is better to keep the options "**wsdl_arraysize**", "w**sdl_stringsize"**, "**wsdl_decimalsize**" set to TRUE (default) so that the BDL client application can do an exact type mapping.

Genero Web Services COM Extension Library

The Genero Web Services COM Extension Library provides classes and methods that allow you to perform tasks associated with creating Services and Clients, and managing the services. Use the IMPORT statement at the top of your Genero .4gl module to import the library into your Genero application:

```
IMPORT com
```

The Web Services classes

Purpose: Manage Web Services servers

Summary:

- CLASS WebService
- CLASS WebOperation
- CLASS WebServiceEngine
  - GLOBAL option flags
  - Notes on WSDL Generation options
- CLASS HTTPServiceRequest

The HTTP classes

Purpose: Manage HTTP client network operations

Summary:

- CLASS HTTPRequest
- CLASS HTTPResponse
  - Example 1 : HTTP GET request
  - Example 2 : XForms HTTP POST request
  - Example 3 : Streaming HTTP PUT request
  - Example 4 : Asynchronous HTTP DELETE request

The TCP classes

Purpose: Manage TCP client network operations

Summary:

- CLASS TCPRequest
- CLASS TCPResponse

COM Library error codes

- Error codes

The Web Service class

Summary:

- Syntax
- Methods

See also: The Genero Web Services COM Extension Library

Syntax

The **Web Service** class provides an interface to create and manage Genero Web Services.

Note that *status* is set to zero after a successful method call.

Syntax

```
com.WebService
```

Methods

Class Methods	
Name	**Description**
`com.WebService.createWebService(` `  name STRING,` `  namespace STRING )` `  RETURNING com.WebService`	Creates a WebService object by providing the mandatory *name* and *namespace,* which must be unique in the entire application. Throws an exception in case of errors, and updates status with an error code.

Object Methods	
Name	**Description**
`setComment(` `  comment STRING )`	Adds a *comment* to a WebService; the comment will be visible in the generated WSDL file. Throws an exception in case of errors, and

	updates status with an error code.
`publishOperation( op com.WebOperation, role STRING )`	Publishes a WebOperation named *op*. The *role* identifies the operation if several operations have the same name. Throws an exception in case of errors, and updates status with an error code.
`saveWSDL( location STRING ) RETURNING status`	Saves the WSDL of the WebService to the file system; *location* is the URL where the service will be deployed. *Status* is 0 if the file was saved, -1 if there was an error.
`generateWSDL( location STRING ) RETURNING xml.DomDocument`	Returns a xml.DomDocument representing the WSDL of the WebService; *location* is the URL where the service will be deployed. Throws an exception in case of errors, and updates status with an error code.
`createHeader( header Variable, encoded INTEGER )`	Creates a global Header of the WebService; *header* is any Variable defining the header, *encoded* specifies the encoding mechanism, where `TRUE` indicates the SOAP Section 5 encoding mechanism and `FALSE` the XML Schema mechanism. Since Headers are always in Document Style, set the *encoded* parameter to FALSE. Throws an exception in case of errors, and updates status with an error code.

The Web Operation class

Summary:

- Syntax
- Methods
- Usage

See also: The Genero Web Services COM Extension Library

Syntax

The **Web Operation** class provides an interface to create and manage the operations of a Genero Web Service.

The Web Operation can be created as RPC Style or Document Style. Both RPC/Literal and Doc/Literal Styles are WS-I compliant (standards set by the Web Services Interoperability organization).

Note that *status* is set to zero after a successful method call.

Syntax

```
com.WebOperation
```

Methods

- Creation
- Configuration

Creation

Class Methods	
Name	**Description**
`com.WebOperation.CreateRPCStyle(` `  function STRING,` `  operation STRING,` `  input Variable,` `  output Variable )` `  RETURNING com.WebOperation`	Creates a **Request-Response RPC Style** WebOperation object, where *function* is the name of the 4GL function that is executed to process the XML operation; *operation* is the name of the XML operation; *input* is

	the input record defining the input parameters of the operation (or NULL if there is none) *output* is the output record defining the output parameters of the operation (or NULL if there is none). This method returns a WebOperation object. Throws an exception in case of errors, and updates status with an error code.
`com.WebOperation.CreateDOCStyle(` `  function STRING,` `  operation STRING,` `  input Variable,` `  output Variable)` `  RETURNING com.WebOperation`	Creates a **Request-Response Document Style** WebOperation object, where *function* is the name of the 4GL function that is executed to process the XML operation; *operation* is the name of the XML operation; *input* is the input record defining the input parameters of the operation (or NULL if there is none) *output* is the output record defining the output parameters of the operation (or NULL if there is none). This method returns a WebOperation object. Throws an exception in case of errors, and updates status with an error code.
`com.WebOperation.CreateOneWayRPCStyle(` `  function STRING,` `  operation STRING,` `  input Variable)` `  RETURNING com.WebOperation`	Creates a **One-Way RPC Style** WebOperation object, where *function* is the name of the 4GL function that is executed to process the XML operation; *operation* is the name of the XML operation; *input* is the input record defining the input parameters of the operation (or NULL if there is none). This method returns a WebOperation object. **Notice** that there is no output parameter to be returned to the client. Throws an exception in case of errors, and updates status with an error code.
`com.WebOperation.CreateOneWayDOCStyle(` `  function STRING,`	Creates a **One-Way Document Style** WebOperation object, where

`operation STRING,` `input Variable)` `RETURNING com.WebOperation`	*function* is the name of the 4GL function that is executed to process the XML operation; *operation* is the name of the XML operation; *input* is the input record defining the input parameters of the operation (or NULL if there is none). This method returns a WebOperation object. **Notice** that there is no output parameter to be returned to the client. Throws an exception in case of errors, and updates status with an error code.

Configuration

Object Methods	
Name	**Description**
`setInputEncoded(` `encoded INTEGER )`	Sets the input parameter *encoding* mechanism of the WebOperation, where values for encoded are `TRUE` indicating the SOAP Section 5 encoding mechanism, and `FALSE` indicating the XML Schema mechanism. Not recommended. Throws an exception in case of errors, and updates status with an error code.
`setOutputEncoded(` `encoded INTEGER )`	Sets the output parameter *encoding* mechanism of the WebOperation; where values for encoded are `TRUE` indicating the SOAP Section 5 encoding mechanism, and `FALSE` indicating the XML Schema mechanism. Not recommended. Throws an exception in case of errors, and updates status with an error code.
`addInputHeader(` `input Variable )`	Adds an input header to the WebOperation, where *input* is any variable previously created as the Header of the WebService object. Throws an exception in case of errors, and updates status with an error code.
`addOutputHeader(`	Adds an output header to the WebOperation,

`output Variable )`	where *output* is any variable previously created as the Header of the WebService object. Throws an exception in case of errors, and updates status with an error code.
`setComment( comment STRING )`	Adds a *comment* to the WebOperation. The comment will appear in the WSDL of the service. Throws an exception in case of errors, and updates status with an error code.

Usage

RPC Style Service (RPC/Literal) is generally used to execute a function, such as a service that returns a stock option. **Document Style Service** (**Doc/Literal**) is generally used for more sophisticated operations that exchange complex data structures, such as a service that sends an invoice to an application, or exchanges a Word document; this is the MS.Net default. The input or output RECORD cannot have **XMLNamespace** attributes set on their members.

Calling the appropriate function to create the desired style is the only difference in your Genero code that creates the service. The remainder of the code that describes the service is the same, regardless of whether you want to create an RPC or Document style of service.

Do not use the `setInputEncoded()` and `setOutputEncoded()` methods, as they will specify the **RPC/Encoded Style**, which is not recommended (see Choosing a Web Service Style).

Since release 2.0 GWS allows you to create RPC Style and Document Style operations in the same Web Service. However, we do not recommend this, as it is not WS-I compliant.

The Web Service Engine class

Summary:

See also: The Genero Web Services COM Extension Library

Syntax

The **Web Service Engine** class provides an interface to manage the Web Services Engine. This class does not have to be instantiated.

Note that *status* is set to zero after a successful method call.

Syntax:

```
com.WebServiceEngine
```

Methods:

Class Methods	
Name	**Description**
`com.WebServiceEngine.RegisterService( ws com.WebService )`	Registers a WebService to the DVM, where *ws* is the WebService to be registered. Throws an exception in case of errors, and updates status with an error code.
`com.WebServiceEngine.Start()`	Starts all registered Web Services; throws an exception if the services cannot be started.
`com.WebServiceEngine.ProcessServices( timeout INTEGER ) RETURNING status`	Specifies the wait period for an HTTP input request, to process an operation of one of the registered Web Services.

	timeout is the time in seconds to wait for an incoming request before returning; the value -1 specifies an infinite waiting time. Returns a value indicating the status.
`com.WebServiceEngine.getHTTPServiceRequest( timeout INTEGER) RETURNING com.HTTPServiceRequest`	Returns a HTTPServiceRequest object to handle an incoming HTTP request, or null if there was none during the given period of time. *timeout* is the time in seconds to wait for an incoming request, and -1 means infinite wait. **Notice** that any new call to this function will raise an error until the previous HTTP request was handled by sending a response back to the client, or destroyed. Throws an exception in case of errors, and updates status with an error code.
`com.WebServiceEngine.SetFaultCode( code STRING, codeNS STRING )`	Defines a user SOAP Fault code to be returned to the client, where *code* is the mandatory SOAP Fault code and *codeNS* is the mandatory Code namespace. This function has an effect only if called inside a WebService operation.
`com.WebServiceEngine.SetFaultString( str STRING )`	Defines a user SOAP Fault description to be returned to the client, where *str* is the description string. This function has an effect only if called inside a WebService operation.
`com.WebServiceEngine.SetOption( flag STRING, value INTEGER )`	Sets an option flag to change the global behavior of the Web Services engine, where *flag* is the option flag and *value* is the value of the flag. See Global option flags.

	Throws an exception in case of errors, and updates status with an error code.
`com.WebServiceEngine.GetOption( flag STRING ) RETURNING value`	Gets an option flag value of the Web Services engine, where *flag* is the option flag and *value* is the value of the flag. See Global option flags. Throws an exception in case of errors, and updates status with an error code.

Usage

A human-readable description of the error codes for `RegisterService` and `ProcessServices` is available in the SQLCA.SQLERRM structure

`ProcessServices` returns a status value with the following meaning:

```
  0 : A request has been processed successfully
 -1 : Time out reached
 -2 : Disconnected from application server
 -3 : Lost connection with the client
 -4 : Server has been interrupted with Ctrl-C
 -5 : Bad HTTP request
 -6 : Malformed SOAP envelope
 -7 : Malformed XML document
 -8 : HTTP error
 -9 : Unsupported operation
-10 : Internal server error
-11 : WSDL Generation failed
-12 : WSDL Service not found
-13 : Reserved
-14 : Incoming request overflow
-15 : Server was not started
-16 : Request still in progress
-17 : Stax response error
```

Global option flags for `SetOption` and `GetOption`:

Flag	Client or Server	Description
readwritetimeout	Client	Defines the default maximum time in

		seconds a client, a HTTP request/response and a TCP request/response have to wait before to raise an error because the server doesn't return or accept data. **Note:** A value of -1 means infinite wait (the default value is -1)
connectiontimeout	Client	Defines the default maximum time in seconds a client, a HTTPRequest and a TCPRequest have to wait for the establishment of a connection with a server. **Note:** A value of -1 means infinite wait (the default value is 30 seconds except for Windows, where it is 5 seconds).
maximumresponselength	Both	Defines the maximum size in **KBytes** a client, the server, a HTTP or TCP response, allows before to break. **Note:** A value of -1 means no limit (the default value is -1).
wsdl_decimalsize	Server	Defines whether the precision and scale of a DECIMAL variable will be taken into account during the WSDL generation. See Notes on WSDL Generation Options below. **Note:** A value of zero means false (the default value is true).
wsdl_arraysize	Server	Defines whether the size of a BDL array will be taken into account during the WSDL generation. See Notes on WSDL Generation Options below. **Note:** A value of zero means false (the default value is true).
wsdl_stringsize	Server	Defines whether the size of a CHAR or VARCHAR variable will be taken into account during the WSDL generation. See Notes on WSDL Generation Options below. **Note:** A value of zero means false (the default value is true).
http_invoketimeout (deprecated use readwritetimeout)	Client	Defines the default maximum time in seconds a client has to wait before the client connection raises an error because the server is not responding. **Note:** A value of -1 means that it has to wait until the server responds (the default value is -1)

tcp_connectiontimeout (deprecated use connectiontimeout)	Client	Defines the default maximum time in seconds a client has to wait for the establishment of a TCP connection with a server. **Note:** A value of -1 means infinite wait (the default value is 30 seconds except for Windows, where it is 5 seconds).

Notes on WSDL Generation Options

1. For a BDL type **DECIMAL(5,2)**, when "**wsdl_decimalsize**" is TRUE, the generated WSDL file contains the total size and the size of the fractional part of the decimal:

```
<types>
   <schema xmlns="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.4js.com/types/">
      <simpleType name="echoDecimal5_2_a_dec5_2_out_FGLDecimal">
         <restriction base="decimal">
            <totalDigits value="5" />
            <fractionDigits value="2" />
         </restriction>
      </simpleType>
   </schema>
</types>
<message name="echoDecimal5_2">
  <part name="dec5_2" type="f:echoDecimal5_2_a_dec5_2_in_FGLDecimal" />
</message>
```

When "**wsdl_decimalsize**" is FALSE, the total size and the size of the fractional part are not mentioned:

```
<message name="echoDecimal5_2">
  <part name="dec5_2" type="xsd:decimal" />
</message>
```

2. If the WSDL file does not contain the size, the client application has no way of knowing the size. In this case, a default value for the size is generated. For example, the exported server type **DECIMAL(5,2)** becomes a **DECIMAL(32)** on the client side.

3. It is better to keep the options "**wsdl_arraysize**", "**wsdl_stringsize**", "**wsdl_decimalsize**" set to TRUE (default) so that the BDL client application can do exact type mapping.

4. When setting a facet constraint attribute on a simple datatype, the generation of the WSDL will take this attribute into account even if an option has been set to perform the opposite.

5. Notice also that when setting one facet constraint attribute, all default ones won't be generated anymore excepted if you specify them as facet constraint attributes.

Examples

Placeholder

The HTTP Service Request class

Summary:

- Syntax
- Methods

See also: The Genero Web Services COM Extension Library

Syntax

The **HTTP Service Request** class provides an interface to process incoming XML and TEXT requests over HTTP on the server side, with an access to the HTTP layer and additional XML streaming possibilities.

Note that *status* is set to zero after a successful method call.

Syntax

```
com.HTTPServiceRequest
```

Methods

- Reading request from client
- Responding to the client

Reading request from client

Object Methods	
Name	**Description**
`getURL()` `  RETURNING STRING`	Returns the entire URL request containing the host, port, document and query string.
`getMethod()` `  RETURNING STRING`	Returns the HTTP method of the request (GET,POST,PUT,HEAD,DELETE)
`getRequestVersion()` `  RETURNING STRING`	Returns the HTTP version of the request (1.0 or 1.1).
`hasRequestKeepConnection()` `  RETURNING INTEGER`	Returns whether the request expect the connection to stay open after the sending of the response.

`getRequestHeader( name STRING ) RETURNING STRING`	Returns the value of the request header name, or NULL. Throws an exception in case of errors, and updates status with an error code.
`getRequestHeaderCount() RETURNING INTEGER`	Returns the number of request headers.
`getRequestHeaderName( index INTEGER ) RETURNING STRING`	Returns the name of the request header at given position (index is starting at 1). Throws an exception in case of errors, and updates status with an error code.
`getRequestHeaderValue( index INTEGER ) RETURNING STRING`	Returns the value of the request header at given position (index is starting at 1). Throws an exception in case of errors, and updates status with an error code.
`readFormEncodedRequest( utf8 INTEGER ) RETURNING STRING`	Returns the query of a POST **application/x-www-form-urlencoded** request or the query string of a GET request, decoded according to HTML4 or XFORM if utf8 is TRUE. **Note:** If utf8 is TRUE, the decoded query string is translated from utf-8 to the locale charset that might lead to a conversion error. Throws an exception in case of errors, and updates status with an error code.
`readTextRequest() RETURNING STRING`	Returns the body of the request as a string. Supported methods are PUT and POST. **Note:** The request Content-Type header must be of the form **text/*** as for instance text/richtext. Throws an exception in case of errors, and updates status with an error code.
`readXmlRequest() RETURNING xml.DomDocument`	Returns the body of the request as an entire XML document. Supported methods are PUT and POST. **Note:** The request Content-Type header must be of the form ***/xml** or ***/*+xml** as for instance application/xhtml+xml. Throws an exception in case of errors, and updates status with an error code.
`beginXmlRequest() RETURNING xml.StaxReader`	Begins the streaming HTTP request and returns a xml.StaxReader object ready to read the XML from the client. Supported methods are PUT and POST. **Note:** The request Content-Type header must be of the form ***/xml** or ***/*+xml** as for instance application/xhtml+xml. Throws an exception in case of errors, and updates status with an error code.

`endXmlRequest( stax xml.StaxReader )`	Ends the streaming HTTP request by closing the StaxReader. Throws an exception in case of errors, and updates status with an error code.

Responding to the client

Object Methods	
Name	**Description**
`setResponseVersion( version STRING )`	Sets the HTTP response version (1.0 or 1.1). **Notes:** • If no set, the same version as the request is used. • The method must be called before sending the response with *sendResponse*, *sendTextResponse*, *sendXmlResponse* or *beginXmlResponse* and *endXmlResponse* Throws an exception in case of errors, and updates status with an error code.
`setResponseCharset( charset STRING )`	

The HTTP Request class

Summary:

- Syntax
- Methods
- Examples

See also: The Genero Web Services COM Extension Library

Syntax

The **HTTP Request** class provides an interface to perform asynchronous XML and TEXT requests over HTTP for a specified URL, with additional XML streaming possibilities.

Note that *status* is set to zero after a successful method call.

Syntax

```
com.HTTPRequest
```

Methods

- Creation
- Configuration
- Sending
- Response

Creation

Class Methods	
Name	**Description**
`com.HTTPRequest.Create(` `  url STRING )` `   RETURNING com.HTTPRequest`	Creates an HTTPRequest object by providing a mandatory *url* with HTTP or HTTPS as the protocol. **Note:** *url* can be an identifier of an URL mapping with an optional **alias://** prefix. See FGLPROFILE Configuration for more details about URL mapping with aliases, and for proxy and security configuration.

	Throws an exception in case of errors, and updates status with an error code.

Configuration

Object Methods	
Name	**Description**
`setVersion( version STRING )`	Sets the HTTP *version* of the request. **Note:** Only 1.0 and 1.1 are supported. Default is 1.1. Throws an exception in case of errors, and updates status with an error code.
`setMethod( method STRING )`	Sets the HTTP *method* of the request. **Note:** Supported methods are GET, PUT, POST, HEAD and DELETE. Default is GET. Throws an exception in case of errors, and updates status with an error code.
`setHeader( name STRING, value STRING )`	Sets an HTTP header *name* and *value* for the request, and replaces the previous one if there was any. **Note:** Setting a header after the body has been sent, or if a streaming operation has been started, will only be taken into account when a new request is reissued. Throws an exception in case of errors, and updates status with an error code.
`removeHeader( name STRING )`	Removes an HTTP header *name* for the request if it exists. Throws an exception in case of errors, and updates status with an error code.
`clearHeaders()`	Removes all user-defined headers.
`setCharset( charset STRING )`	Defines the *charset* used when sending text or XML; by default no charset is set. **Note:** When sending text, HTTP specification defines ISO-8859-1 as an implicit charset. **Note:** When sending XML, the user-defined charset is used instead of the one set in the XML document itself, which can lead to a charset conversion error at the server side. Therefore it is recommended

	that you unset it by setting *charset* to NULL, or that you use the same charset that was set in the XML Document.
`setAuthentication( login STRING, password STRING, scheme STRING, realm STRING )`	Defines the mandatory user *login* and *password* to authenticate to the server. An optional *scheme* defines the method to be used during authentication. **Note:** Only Anonymous, Basic and Digest are supported. Default is Anonymous. An optional *realm* can also be set. **Note:** With Anonymous or Digest authentication, you must resend the request if you get a 401 or 407 HTTP return code (authorization required) **Note:** If a user-defined authentication is set and there is an authenticate entry for this URL in the FGLPROFILE file, the user-defined has priority. Throws an exception in case of errors, and updates status with an error code.
`clearAuthentication()`	Removes user-defined authentication. **Note:** If an authenticate entry exists in the FGLPROFILE file, it will be used for authentication, even if the user-defined was removed.
`setKeepConnection( keep INTEGER )`	Defines whether the connection should stay open if a new request occurs again. Default is FALSE.
`setTimeOut( timeout INTEGER )`	Sets the time value in seconds to wait for a reading or writing operation, before a break. **Note:** -1 means infinite.
`setConnectionTimeOut( timeout INTEGER )`	Sets the time value in seconds to wait for the establishment of the connection, before a break. **Note:** -1 means infinite.
`setMaximumResponseLength( length INTEGER )`	Sets the maximum authorized size in **Kbyte** of the whole response (composed of the headers, the body and all control characters), before a break. **Note:** -1 means no limit.

Sending

Object Methods	
Name	**Description**
`doRequest()`	Performs the request. Supported methods are GET, HEAD and DELETE. Throws an exception in case of errors, and updates status with an error code.
`doTextRequest( txt STRING )`	Performs the request by sending an entire string at once. Supported methods are PUT and POST. **Note:** The default Content-Type header is **text/plain**, but it can be changed if of the form **text/*,** for instance text/richtext. **Note:** Automatic conversion from locale to user-defined charset is performed when possible, otherwise throws an exception. **Note:** In HTTP 1.1, if the body size is greater than 32k, the request will be sent in several chunks of the same size. Throws an exception in case of errors, and updates status with an error code.
`doXmlRequest( doc xml.DomDocument )`	Performs the request by sending the entire xml.DomDocument at once. Supported methods are PUT and POST. **Note:** The default Content-Type header is **text/xml**, but it can be changed if of the form ***/xml** or ***/*+xml**, for instance application/xhtml+xml. **Note:** In HTTP 1.1, if the body size is greater than 32k, the request will be sent in several chunks of the same size. Throws an exception in case of errors, and updates status with an error code.
`doFormEncodedRequest( query STRING, utf8 INTEGER )`	Performs an **application/x-www-form-urlencoded** Forms encoded query. Supported methods are GET and POST. The *query* string is a list of name/value pairs separated with &. For example, name1=value1&name2=value2&name3=value3. **Note:** If utf8 is TRUE, the query string is encoded in UTF-8 as specified in XForms 1.0, otherwise in ASCII as specified in HTML 4. Throws an exception in case of errors, and updates status with an error code.
`beginXmlRequest() RETURNING xml.StaxWriter`	Begins the streaming HTTP request and returns an xml.StaxWriter object ready to send XML to the server. Supported methods are PUT and POST. **Note:** The default Content-Type header is **text/xml**, but it can be changed if of the form ***/xml** or

	***/*+xml,** for instance application/xhtml+xml. **Note:** In HTTP 1.1, if the body size is greater than 32k, the request will be sent in several chunks of the same size. Throws an exception in case of errors, and updates status with an error code.
`endXmlRequest( writer xml.StaxWriter )`	Ends the streaming HTTP request by closing the Stax *writer*. Throws an exception in case of errors, and updates status with an error code.

Response

Object Methods	
Name	**Description**
`getResponse() RETURNING com.HTTPResponse`	Returns the response of one of the *doRequest*, *doTextRequest*, *doXmlRequest*, *doFormEncodedRequest* or *beginXmlRequest and endXmlRequest* calls in an com.HTTPResponse object. Throws an exception in case of errors, and updates status with an error code.
`getAsyncResponse() RETURNING com.HTTPResponse`	Returns the response of one of the *doRequest*, *doTextRequest*, *doXmlRequest*, *doFormEncodedRequest* or *beginXmlRequest and endXmlRequest* calls in an com.HTTPResponse object, or NULL if the response was not yet received. **Remarks:** If a previous call returned NULL, a new call will return the expected response if it has already arrived, or NULL again if the response was still not received. Throws an exception in case of errors, and updates status with an error code.

Examples

Example 1 : HTTP GET request

```
    IMPORT com

    MAIN
      DEFINE req      com.HTTPRequest
      DEFINE resp     com.HTTPResponse

      TRY
        LET req =
com.HTTPRequest.Create("http://localhost:8090/MyPage")
        CALL req.setHeader("MyHeader","High Priority") # Set
additional HTTP header with name 'MyHeader', and value 'High
Priority'
        CALL req.doRequest()
        LET resp = req.getResponse()
        IF resp.getStatusCode() != 200 THEN
          DISPLAY  "HTTP Error ("||resp.getStatusCode()||")
",resp.getStatusDescription()
        ELSE
          DISPLAY  "HTTP Response is : ",resp.getTextResponse()
        END IF
      CATCH
        DISPLAY "ERROR :",STATUS||" ("||SQLCA.SQLERRM||")"
      END TRY
    END MAIN
```

Example 2 : XForms HTTP POST request

```
    IMPORT com
    IMPORT xml

    MAIN
      DEFINE req      com.HTTPRequest
      DEFINE resp     com.HTTPResponse
      DEFINE doc      xml.DomDocument

      TRY
        LET req =
com.HTTPRequest.Create("http://localhost:8090/MyProcess")
        CALL req.setMethod("POST") # Perform an HTTP POST method
        CALL req.doFormEncodedRequest("Param1=hello&Param2=how
are you ?",FALSE) # Param1 value is 'hello', Param2 value is
'how are you ?'
        LET resp = req.getResponse()
        IF resp.getStatusCode() != 200 THEN
          DISPLAY  "HTTP Error ("||resp.getStatusCode()||")
",resp.getStatusDescription()
        ELSE
          LET  doc = resp.getXmlResponse() # Expect a returned
content type of the form */xml
```

```
          DISPLAY  "HTTP XML Response is : ",doc.saveToString()
        END IF
      CATCH
        DISPLAY "ERROR :",STATUS||" ("||SQLCA.SQLERRM||")"
      END TRY
    END MAIN
```

Example 3 : Streaming HTTP PUT request

```
    IMPORT com
    IMPORT xml

    MAIN
      DEFINE req      com.HTTPRequest
      DEFINE resp     com.HTTPResponse
      DEFINE writer   xml.StaxWriter

      TRY
        LET req =
com.HTTPRequest.Create("http://localhost:8090/MyXmlProcess")
        CALL req.setMethod("PUT") # Perform an HTTP PUT method
        CALL req.setHeader("MyHeader","Value of my header")
        LET writer = req.beginXmlRequest() # Retrieve an
xml.StaxWriter to start xml streaming
        CALL writer.startDocument("utf-8","1.0",true)
        CALL writer.comment("My first XML document sent in
streaming with genero!!!")
        CALL writer.startElement("root")
        CALL writer.attribute("attr1","value1")
        CALL writer.endElement()
        CALL writer.endDocument()
        CALL req.endXmlRequest(writer) # End streaming request
        LET resp = req.getResponse()
        IF resp.getStatusCode() != 201 OR resp.getStatusCode()
!= 204 THEN
          DISPLAY  "HTTP Error ("||resp.getStatusCode()||")
",resp.getStatusDescription()
        ELSE
          DISPLAY  "XML document was correctly put on the
server"
        END IF
      CATCH
        DISPLAY "ERROR :",STATUS||" ("||SQLCA.SQLERRM||")"
      END TRY
    END MAIN
```

Example 4 : Asynchronous HTTP DELETE request

```
    IMPORT com

    MAIN
      DEFINE req         com.HTTPRequest
```

```
      DEFINE resp        com.HTTPResponse
      DEFINE url         STRING
      DEFINE quit        CHAR(1)
      DEFINE questionStr STRING
      DEFINE timeout     INTEGER

      TRY
        WHILE TRUE
          PROMPT "Enter http url you want to delete ? " FOR url
ATTRIBUTE (CANCEL=FALSE)
          LET req = com.HTTPRequest.Create(url)
          CALL req.setMethod("DELETE")
          CALL req.doRequest()
          LET resp = req.getAsyncResponse()
          # Retrieve asynchronous response for the first time
          CALL Update(resp) RETURNING questionStr,timeout
          WHILE quit IS NULL OR ( quit!="Y" AND quit!="N" )
            PROMPT questionStr FOR CHAR quit ATTRIBUTE
(CANCEL=FALSE,ACCEPT=FALSE,SHIFT="up")
              ON IDLE timeout
                IF resp IS NULL THEN
                # If no response at first try, retrieve it again
                  LET resp = req.getAsyncResponse()
                  # because we have time now !!!
                  CALL Update(resp) RETURNING
questionStr,timeout
                END IF
            END PROMPT
          END WHILE
          IF quit == "Y" THEN
            EXIT PROGRAM
          ELSE
            LET quit = NULL
          END IF
        END WHILE
      CATCH
        DISPLAY "ERROR :",STATUS,SQLCA.SQLERRM
      END TRY
    END MAIN

    FUNCTION Update(resp)
      DEFINE resp    com.HTTPResponse
      DEFINE ret     STRING
      IF resp IS NOT NULL THEN
        IF resp.getStatusCode() != 204 THEN
          LET  ret = "HTTP Error ("||resp.getStatusCode()||
  ") :"||resp.getStatusDescription()||". Do you want to quit ? "
        ELSE
          LET  ret = "HTTP Page deleted. Do you want to quit ? "
        END IF
        RETURN ret, 0
      ELSE
        LET ret = "Do you want to quit ? "
        RETURN ret, 1
      END IF
    END FUNCTION
```

The HTTP Response class

Summary:

- Syntax
- Methods
- Usage
- Examples

See also: The Genero Web ServicesCOM Extension Library

Syntax

The **HTTP Response** class provides an interface to perform XML and TEXT responses over HTTP, with additional XML streaming possibilities. Notice that *status* is set to zero after a successful method call.

Syntax

```
com.HTTPResponse
```

Methods

Class Methods	
Name	**Description**

Object Methods	
Name	**Description**
`getStatusCode()` `  RETURNING INTEGER`	Returns the HTTP status code. When the returned HTTP status code is 401 or 407, authorization is required. See the **setAuthentication()** method of the HTTPRequest class from the Genero Web Services Extension COM Library.

`getStatusDescription()` `RETURNING STRING`	Returns the HTTP status description.
`getHeader(` `name STRING )` `RETURNING STRING`	Returns the value of the HTTP header *name*, or NULL if there is none. Throws an exception in case of errors, and updates status with an error code.
`getHeaderCount()` `RETURNING INTEGER`	Returns the number of headers. Throws an exception in case of errors, and updates status with an error code.
`getHeaderName(` `index INTEGER)` `RETURNING STRING`	Returns the name of the header at position *index*. Throws an exception in case of errors, and updates status with an error code.
`getHeaderValue(` `index INTEGER )` `RETURNING STRING`	Returns the value of the header at position *index*. Throws an exception in case of errors, and updates status with an error code.
`beginXmlResponse()` `RETURNING xml.StaxReader`	Begins the streaming HTTP response and returns a xml.StaxReader object ready to read XML from the server. **Note:** The Content-Type header must be of the form ***/xml** or ***/*+xml**, for instance application/xhtml+xml. Throws an exception in case of errors, and updates status with an error code.
`endXmlResponse(` `reader xml.StaxReader )`	Ends the streaming HTTP response by closing the Stax *reader*. Throws an exception in case of errors, and updates status with an error code.
`getXmlResponse()` `RETURNING xml.DomDocument`	Returns an entire xml.DomDocument as response from the server. **Note:** The Content-Type header must be of the form ***/xml** or ***/*+xml**, for instance application/xhtml+xml. Throws an exception in case of errors, and updates status with an error code.
`getTextResponse()` `RETURNING STRING`	Returns an entire string as response from the server. **Note:** The Content-Type header must be of the form **text/***, for instance text/richtext. **Note:** Automatic conversion to the locale charset is performed when possible, otherwise throws an exception. Throws an exception in case of errors,

and updates status with an error code.

Examples

For examples, refer to The HTTP Request Class Examples.

The TCP Request class

Summary:

- Syntax
- Methods

See also: The Genero Web Services COM Extension Library

Syntax

The **TCP Request** class provides an interface to perform asynchronous XML and TEXT requests over TCP, with additional XML streaming possibilities. Notice that *status* is set to zero after a successful method call.

Syntax

```
com.TCPRequest
```

Methods

- Creation
- Management

Creation

Class Methods	
Name	**Description**
`com.TCPRequest.Create(` `  url STRING )` `  RETURNING com.TCPRequest`	Creates a TCPRequest object by providing a mandatory *url* with TCP or TCPS as the protocol. **Note:** *url* can be an identifier of an URL mapping with an optional **alias://** prefix. See FGLPROFILE Configuration for more details about URL mapping with aliases, and for proxy and security configuration. Throws an exception in case of errors, and updates status with an error code.

Management

Object Methods	
Name	**Description**
`setTimeOut( timeout INTEGER )`	Sets the time value in seconds to wait for a reading or writing operation, before a break. **Note:** -1 means infinite.
`setConnectionTimeOut( timeout INTEGER )`	Sets the time value in seconds to wait for the establishment of the connection, before a break. **Note:** -1 means infinite.
`setMaximumResponseLength( length INTEGER )`	Sets the maximum authorized size in **Kbyte** of the whole response, before a break. **Note:** -1 means no limit.
`doRequest()`	Performs the request. **Note:** The connection is shutdown for writing to notify that no data will be sent. Throws an exception in case of errors, and updates status with an error code.
`doXmlRequest( doc xml.DomDocument )`	Performs the request by sending the entire xml.DomDocument at once. **Note:** The connection is shutdown for writing to notify that no more data will be sent. Throws an exception in case of errors, and updates status with an error code.
`doTextRequest( str STRING)`	Performs the request by sending a string at once. **Note:** The connection is shutdown for writing to notify that no more data will be sent. Throws an exception in case of errors, and updates status with an error code.
`beginXmlRequest() RETURNING xml.StaxWriter`	Begins the streaming HTTP request and returns an xml.StaxWriter object ready to send XML to the server. Throws an exception in case of errors,

	and updates status with an error code.
`endXmlRequest( writer xml.StaxWriter )`	Ends the streaming HTTP request by closing the Stax *writer*. **Note:** The connection is shutdown for writing to notify that no more data will be sent. Throws an exception in case of errors, and updates status with an error code.
`getResponse() RETURNING com.TCPResponse`	Returns the response of one of the *doRequest*, *doXmlRequest*, *doTextRequest* or *beginXmlRequest and endXmlRequest* calls in a com.TCPResponse object. Throws an exception in case of errors, and updates status with an error code.
`getAsyncResponse() RETURNING com.TCPResponse`	Returns the response of one of the *doRequest*, *doXmlRequest*, *doTextRequest* or *beginXmlRequest and endXmlRequest* calls in a com.TCPResponse object, or NULL if the response was not yet received. **Remarks:** If a previous call returned NULL, a new call will return the expected response if it already arrived, or NULL again if the response was still not received. Throws an exception in case of errors, and updates status with an error code.

The TCP Response class

Summary:

- Syntax
- Methods

See also: The Genero Web Services COM Extension Library

Syntax

The **TCP Response** class provides an interface to perform XML and TEXT responses over TCP, with additional XML streaming possibilities. Notice that *status* is set to zero after a successful method call.

Syntax:

```
com.TCPResponse
```

Methods

Object Methods	
Name	**Description**
`beginXmlResponse()` `  RETURNING` `xml.StaxReader`	Begins the streaming TCP response and returns an xml.StaxReader object ready to read XML from the server. Throws an exception in case of errors, and updates status with an error code.
`endXmlResponse(` `  reader xml.StaxReader` `)`	Ends the streaming TCP response by closing the Stax *reader*. Throws an exception in case of errors, and updates status with an error code.
`getXmlResponse()` `  RETURNING` `xml.DomDocument`	Returns an entire xml.DomDocument as response from the server. Throws an exception in case of errors, and updates status with an error code.
`getTextResponse()` `  RETURNING STRING`	Returns a string as response from the server. Throws an exception in case of errors, and updates status with an error code.

The COM Library Error Codes

Code	Description
0	No error.
-15534	Invalid self object.
-15535	Cannot perform operation due to invalid parameters.
-15536	Service registration failed.
-15537	Cannot create web service.
-15538	Cannot create Web operation.
-15539	Cannot publish Web operation.
-15540	Published 4GL function not found.
-15541	Published 4GL function not correctly defined.
-15542	Input parameter of published operation error.
-15543	Output parameter of published operation error.
-15544	Web Service header configuration error.
-15545	Service is already registered.
-15546	Invalid option.
-15547	Unsupported web service operation.
-15548	Bad URI.
-15549	HTTP runtime exception, see SQLCA.SQLERRM for mode details.
-15550	XML runtime exception, see SQLCA.SQLERRM for mode details.
-15551	WSDL generation failed.
-15552	Charset conversion exception, see SQLCA.SQLERRM for mode details.
-15553	TCP runtime exception, see SQLCA.SQLERRM for mode details.
-15554	Index is out of bound.
-15555	Unsupported request-response feature.
-15556	No request was sent.
-15557	Request was already sent.
-15558	Waiting for a response.
-15559	No stream available.
-15560	Streaming is over.
-15561	Streaming in progress.

-15562 Streaming not yet started.

-15563 Streaming already started.

-15564 Unexpected peer stream was shutdown.

-15565 Cannot return incoming request, see SQLCA.SQLERRM for more details.

-15599 Internal error, should not happen.

See also: The Genero Web Services COM Extension Library

The Genero Web Services XML Extension Library

The Genero Web Services XML Extension Library provides classes and methods to handle any kind of XML documents, including documents with namespaces. The library provides a W3C-compatible DOM API, integrating additional XML Schema and DTD validation methods. There is also an API compatible with StAX for writing or reading XML documents where performance and speed are important.

Use the IMPORT statement at the top of your Genero .4gl module to import the library into your Genero application:

```
IMPORT xml
```

Remark: the DOM API of the *om* package is still in use, but was designed to handle specific FGL files or to manipulate the user interface tree (the AUI tree). For all other cases/scenarios, we recommend that you use the DOM API of the *xml* package.

The Document Object Modeling (DOM) classes

Purpose: Manages XML document entirely in memory with support of XML Schema and DTD validation

Summary:

- CLASS DomDocument
  - Features
- CLASS DomNode
  - Types
- CLASS DomNodeList

The Streaming API for XML (StAX) classes

Purpose: Uses streaming while managing XML documents

Summary:

- CLASS StaxWriter
  - Features
  - Example
- CLASS StaxReader
  - Event types
  - Features
  - Example

The XML serialization class

Purpose: Converts 4GL variables to XML and XML to 4GL variables

Summary:

- CLASS Serializer
  - Option flags

Library error codes

Summary:

- Error codes

The DomDocument class

Summary:

- Syntax
- Methods
- DomDocument Features
- Examples
    - Create a namespace qualifed XML document
    - Validating a document against XML schemas or an embedded DTD

See also: The Genero Web Services XML Extension Library

Syntax

The **DomDocument** class provides methods to manipulate a data tree, following the DOM standards.

The *status* is set to zero after a successful method call.

Syntax

```
xml.DomDocument
```

Methods

- Creation
- Navigation
- Management
- Node creation
- Load and save
- Configuration
- Validation
- Error management

Creation methods

Class Methods	
Name	**Description**
`xml.DomDocument.create()` `  RETURNING xml.DomDocument`	Constructor of an empty DomDocument object; returns a DomDocument object.
`xml.DomDocument.createDocument(` `  name STRING )` `  RETURNING xml.DomDocument`	Constructor of a DomDocument with an XML root element; where *name* is the name of the XML Element.

	Returns a DomDocument object or NULL. Throws an exception in case of errors, and updates status with an error code.
`xml.DomDocument.createDocumentNS( prefix STRING, name STRING, ns STRING ) RETURNING xml.DomDocument`	Constructor of a DomDocument with a root namespace-qualified XML root element where *prefix* is the prefix of the XML Element or NULL, *name* is the name of the XML Element, and *ns* is the namespace of the XML Element. Returns a DomDocument object. Throws an exception in case of errors, and updates status with an error code.

Usage

```
xml.domDocument.create()
```

Create a DomDocument without a root node.

```
xml.domDocument.create("ARoot")
```

Create a DomDocument with an initial root node named **ARoot**.

```
xml.domdocument.createDocumentNS("fjs","List","http://www.mysite.com/xmlapi")
```

Produces:

```
<fjs:List xmlns:fjs="http://www.mysite.com/xmlapi">
[...]
</fjs:List>
```

Create a DomDocument with an initial root node named **List** with `fjs` as the prefix and `http://www.mysite.com/xmlapi` as the namespace.

Navigation methods

Object Methods	
Name	**Description**
`getDocumentElement() RETURNING xml.DomNode`	Returns the root XML Element DomNode object for this DomDocument object. Returns a DomNode object, or NULL.
`getFirstDocumentNode() RETURNING xml.DomNode`	Returns the first child DomNode object for this DomDocument object, or NULL.

`getLastDocumentNode()` `RETURNING` `xml.DomNode`	Returns the last child DomNode object for this DomDocument object, or NULL.
`getDocumentNodesCount()` `RETURNING INTEGER`	Returns the number of child DomNode objects for this DomDocument object.
`getDocumentNodeItem(` `pos INTEGER )` `RETURNING` `xml.DomNode`	Returns the child DomNode object at a given position for this DomDocument object where *pos* is the position of the node to return (Index starts at 1), or NULL. Throws an exception in case of errors, and updates status with an error code.
`getElementsByTagName(` `name STRING)` `RETURNING` `xml.DomNodeList`	Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name in the entire document; *name* is the name of the XML Element tag to match, or "*" to match all tags. Returns a DomNodeList object, or NULL. **Note:** The returned list is ordered using a Depth-First pass algorithm. Throws an exception in case of errors, and updates status with an error code.
`getElementsByTagNameNS(` `name STRING,` `ns STRING )` `RETURNING` `xml.DomNodeList`	Returns a DomNodeList object containing all namespace qualified XML Element DomNode objects with the same tag name and namespace in the entire document; *name* is the name of the XML Element tag to match, or "*" to match all tags; *ns* is the namespace URI of the XML Element tag to match, or "*" to match all namespaces. Returns a DomNodeList object, or NULL. **Note:** The returned list is ordered using a Depth-First pass algorithm. Throws an exception in case of errors, and updates status with an error code.
`selectByXPath(` `expr STRING,` `NamespacesList ...)` `RETURNING` `xml.DomNodeList`	Returns a DomNodeList object containing all DomNode objects matching an XPath 1.0 expression **(Not part of W3C API)**; *expr* is the XPath 1.0 expression, *NamespacesList* is a list of prefixes bounded to namespaces in order to resolve qualified names in the XPath expression. This list must be filled with an even number of arguments, representing the prefix and its corresponding namespace. **Examples :** `selectByXPath("//d:Record", "d", "http://defaultnamespace")` `selectByXPath("//ns1:Record", NULL)` `selectByXPath("//ns1:Records/ns2:Record", "ns1", "http://namespace1", "ns2", "http://namespace2")` `selectByXPath("//ns1:Record", "ns1")` is invalid because the namespace definition is missing. **Note:** If the namespaces list is NULL, the prefixes and namespaces defined in the document itself are used if available. **Note:** A namespace must be an absolute URI (ex 'http://',

	'file://'). Throws an exception in case of errors, and updates status with an error code.
`getElementById( id STRING) RETURNING xml.DomNode`	Returns the Element that has an attribute of type ID with the given value, or NULL if there is none. **Note:** Attributes with the name "ID" or "id" are not of type ID unless so defined with setIdAttribute or setIdAttributeNS. However, there is a specific attribute called **xml:id** and belonging to the namespace **http://www.w3.org/XML/1998/namespace** that is always of type ID even if not set with setIdAttributeNS. Throws an exception in case of errors, and updates status with an error code.

Usage

DomDocument navigation functions deal with nodes immediately under the DomDocument object, except for search features. To navigate through all the nodes, you can refer to the navigation functions of the class xml.DomNode.

```
<?xml version="1.0" encoding="ISO-8859-1"?><?xml-stylesheet
type="text/xsl" href="card.xsl"?>
<!-- demo card --><CardList xml:id="1" >[...]</CardList>
```

The first node of the document is `xml-stylesheet`. Use getFirstDocumentNode to get the node.
The element at position 2 is the comment `<!-- demo card -->`. Use getDocumentNodeItem function to get the node.
The last node of the document is `CardList`. Use getLastDocumentNode to get the node.
The number of node of the document is 3. This is result of function getDocumentNodesCount. This function only count the number of children immediately under the DomDocument.
Note that the first line of the example, `<?xml version="1.0" encoding="ISO-8859-1"?>`, is not considered as a node. To access to the information of the first line, use getXmlVersion and getXmlEncoding functions.
Caution, if the example is in pretty printed format, the results are not the same. There are addition text nodes representing the carriage returns.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="card.xsl"?>
<!-- demo card -->
<CardList xml:id="1" >
[...]
</CardList>
```

See Cautions section for more details.

You can select nodes using their tag names, by XPath, or by their attributes value (if of type ID, xml:id for example). The `getElementsbyTagName` and `getElementsbyTagNameNS` methods return a DomNodeList object, unlike the other

methods that return a DomNode object. The DomNodeList is restricted to contain objects with the same tag name and/or namespace. The `selectByXPath` method also returns a DomNodeList object, but each node can have a different name.

```
getElementsByTagNameNS("message","http://schemas.xmlsoap.org/wsdl/")
```

Get the *message* nodes that have *http://schemas.xmlsoap.org/wsdl/* as the namespace.

```
getElementsByTagNameNS("message","*")
```

Get all the *message* nodes, regardless of the namespace they have.

```
getElementsByTagName("message")
```

Get all the *message* nodes that do not have any namespace.

```
selectByXPath("//xs:element", NULL)
```

Get all the xs:element nodes that has a namespace corresponding to prefix *xs*.

```
selectByXPath("//Card", NULL)
```

Get all the Card nodes that do not have any namespace.

```
getElementById("1")
```

Get the unique node whose attribute of type ID has a value of "1".

Management methods

Object Methods	
Name	**Description**
`importNode(` `  node xml.DomNode,` `  deep INTEGER )` `  RETURNING xml.DomNode`	Imports a DomNode from a DomDocument object into this DomDocument object, where *node* is the node to import. When *deep* is FALSE only the node is imported; when TRUE the node and all its child nodes are imported. Returns the DomNode object that has been imported to this DomDocument, or NULL. **Note:** Document and Document Type nodes cannot be imported. Throws an exception in case of errors, and updates status with an error code.

`clone()` `  RETURNING xml.DomDocument`	Returns a copy of this DomDocument object, or NULL.
`appendDocumentNode(` `  node xml.DomNode )`	Adds a child DomNode object to the end of the DomNode children for this DomDocument object, where *node* is the node to add. Status is updated with an error code. **Note:** Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node and one Document Type node allowed. Throws an exception in case of errors, and updates status with an error code.
`prependDocumentNode(` `  node xml.DomNode )`	Adds a child DomNode object to the beginning of the DomNode children for this DomDocument object **(Not part of W3C API)**; *node* is the node to add. **Note:** Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node and one Document Type node allowed. Throws an exception in case of errors, and updates status with an error code.
`insertBeforeDocumentNode(` `  node xml.DomNode,` `  ref xml.DomNode )`	Inserts a child DomNode object before another child DomNode for this DomDocument object;.*node* is the node to insert, *ref* is the reference node - the node before which the new node must be inserted. **Note:** Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node and one Document Type node allowed. Throws an exception in case of errors, and updates status with an error code.
`insertAfterDocumentNode(` `  node xml.DomNode,` `  ref xml.DomNode )`	Inserts a child DomNode object after another child DomNode for this DomDocument object **(Not part of W3C API)**; *node* is the node to insert; *ref* is the reference node - the node after which the new node must be inserted. **Note:** Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node and one Document Type node allowed. Throws an exception in case of errors, and updates status with an error code.
`removeDocumentNode(` `  node xml.DomNode )`	Removes a child DomNode object from the DomNode children for this DomDocument object, where *node* is the node to remove. **Note:** Only Text nodes, Processing Instruction nodes, Element nodes and Document Type nodes allowed. Throws an exception in case of errors, and updates status with an error code.
`declareNamespace(` `  node xml.DomNode,` `  alias STRING,` `  ns STRING )`	Forces namespace declaration to an XML Element DomNode for this DomDocument object **(Not part of W3C API)**; *node* is the XML Element DomNode that carries the namespace definition; *alias* is the alias of the

	namespace to declare, or NULL to declare the default namespace; *ns* is the URI of the namespace to declare (can only be NULL if alias is NULL). Throws an exception in case of errors, and updates status with an error code.

Node creation methods

Object Methods	
Name	**Description**
`createNode( str STRING ) RETURNING xml.DomNode`	Creates an XML DomNode object from a string for this DomDocument object **(not part of W3C API)**; *str* is the string representation of the DomNode to be created. Returns a DomNode object, or NULL. Throws an exception in case of errors, and updates status with an error code.
`createElement( name STRING ) RETURNING xml.DomNode`	Creates an XML Element DomNode object for this DomDocument object, where *name* is the name of the XML element, cannot be NULL. Returns a DomNode object, or NULL. Throws an exception in case of errors, and updates status with an error code.
`createElementNS(| prefix STRING, name STRING, ns STRING) RETURNING xml.DomNode`	Creates an XML namespace-qualified Element DomNode object for this DomDocument object, where *prefix* is the prefix of the XML element, or NULL to use the default namespace; *name* is the name of the XML element, cannot be NULL; *ns* is the namespace URI of the XML element, cannot be NULL. Returns a DomNode object, or NULL. Throws an exception in case of errors, and updates status with an error code.
`createAttribute( name STRING ) RETURNING xml.DomNode`	Creates an XML Attribute DomNode object for a DomDocument object, where *name* is the name of the XML attribute, cannot be NULL. Returns a DomNode object, or NULL. **Note:** To create a default namespace declaration attribute use **xmlns** as the name. (Using `declareNamespace` instead is recommended.) Throws an exception in case of errors, and updates status with an error code.
`createAttributeNS( prefix STRING, name STRING, ns STRING )`	Creates an XML namespace-qualified Attribute DomNode object for this DomDocument object, where *prefix* is the prefix of the XML attribute, cannot be NULL; *name* is the name of the XML

`RETURNING xml.DomNode`	attribute, cannot be NULL; *ns* is the namespace URI of the XML attribute, cannot be NULL. Returns a DomNode object, or NULL. **Note:** To create a namespace declaration attribute use **xmlns** as the prefix and **http://www.w3.org/XML/1998/namespace** as the namespace. (Using `declareNamespace` instead is recommended.) Throws an exception in case of errors, and updates status with an error code.
`createTextNode( text STRING ) RETURNING xml.DomNode`	Creates an XML Text DomNode object for this DomDocument object, where *text* is the data of the XML Text node, or NULL. Returns a DomNode object, or NULL. **Note:** Only the characters **#x9, #xA**, **#xD**, [**#x20-#xD7FF]**, [**#xE000-#xFFFD**] and [**#x10000-#x10FFFF**] are allowed in the content of an XML Text node. The `saveToFile()` and `normalize()` methods will fail if characters other than those allowed exist in a Text node.
`createComment( comment STRING ) RETURNING xml.DomNode`	Creates an XML Comment DomNode object for this DomDocument object, where *comment* is the data of the XML Comment node, or NULL. Returns a DomNode object, or NULL. **Note:** Only the characters **#x9, #xA**, **#xD**, [**#x20-#xD7FF]**, [**#xE000-#xFFFD**] and [**#x10000-#x10FFFF**] are allowed in the content of an XML Comment node. The character sequence (Double-Hyphen) '--' is not allowed in the content of an XML Comment node. The `saveToFile()` and `normalize()` methods will fail if this sequence or characters other than those allowed exist in a Comment node.
`createCDATASection( cdata STRING ) RETURNING xml.DomNode`	Creates an XML CData DomNode object for this DomDocument object, where *cdata* is the data of the XML CData node, or NULL. Returns a DomNode object, or NULL. **Note:** Only the characters **#x9, #xA**, **#xD**, [**#x20-#xD7FF]**, [**#xE000-#xFFFD**] and [**#x10000-#x10FFFF**] are allowed in the content of an XML CDATASection node. The character sequence (Double-Hyphen) '--' is not allowed in the content of an XML CDATASection node. The `saveToFile()` and `normalize()` methods will fail if this sequence or characters other than those allowed exist in a CDATASection node.
`createEntityReference( ref STRING ) RETURNING xml.DomNode`	Creates an XML EntityReference DomNode object for this DomDocument object, where *ref* is the name of the entity reference Returns a DomNode object, or NULL. **Note:** An Entity Reference node is read-only and

	cannot be modified. Throws an exception in case of errors, and updates status with an error code.
`createProcessingInstruction( target STRING, data STRING ) RETURNING xml.DomNode`	Creates an XML Processing Instruction DomNode object for this DomDocument object, where *target* is the target part of the XML Processing Instruction, cannot be NULL; *data* is the data part of the XML Processing Instruction, or NULL. Returns a DomNode object, or NULL. **Note:** Only the characters **#x9, #xA**, **#xD**, [**#x20-#xD7FF]**, [**#xE000-#xFFFD**] and [**#x10000-#x10FFFF**] are allowed in the content of an XML Processing Instruction node. The character sequence (Double-Hyphen) '--' is not allowed in the content of an XML Processing Instruction. The `saveToFile()` and `normalize()` methods will fail if this sequence or characters other than those allowed exist in a Processing Instruction node. Throws an exception in case of errors, and updates status with an error code.
`createDocumentType( name STRING, publicID STRING, systemID STRING, internalDTD STRING ) RETURNING xml.DomNode`	Creates an XML Document Type (DTD) DomNode object for this DomDocument object **(Not part of W3C API)**; *name* is the name of the document type; *publicId* is the URI of the public identifier or NULL; *systemId* is the URL of the system identifier or NULL (Specifies the file location of the external DTD subset); *internalDTD* is the internal DTD subset or NULL. Returns a DomNode object, or NULL if *internalDTD* is malformed. **Note:** Only internal DTDs are supported. **Note:** The public identifier cannot be set without the system identifier. Throws an exception in case of errors, and updates status with an error code.
`createDocumentFragment() RETURNING xml.DomNode`	Creates an XML Document Fragment DomNode object for this DomDocument object. Returns a DomNode object, or NULL.

Usage

Creating a node for the DomDocument are done in two steps:

- create the node
- add the node to the DomDocument

Each time you create a node you need to append it at the right place in the DomDocument. To add a node the document use the DomDocument management methods or the DomNode manipulation methods.

```
createNode("<LastName>PATTERSON</LastName><FirstName>Andrew</FirstName>
")
```

Creates a structure of nodes.

```
createElement("CardList")
```

Produces
`<CardList>`

```
createElementNS("cny","Company","http://www.mysite.com/")
```

Produces
`<cny:Company xmlns:cny="http://www.mysite.com/" />` or
`<cny:Company />`. See Cautions for more details.

```
createAttribute("Country")
```

Creates a *Country* attribute node.
To set a value to the attribute use the method setNodeValue of the xml.DomNomde class. To add the attribute to an element node use the method setAttributeNode of the xml.DomNode class.

```
createAttributeNS("tw","Town","http://www.mysite.com/cities")
```

Produces
`xmlns:tw="http://www.mysite.com/cities" tw:Town=""`
To set a value to the attribute use the method setNodeValue of the xml.DomNomde class. To add the attribute to an element node use the method setAttributeNodeNS of the xml.DomNode class. For optimization reasons, the namespace is not written aside the attribute until the saving of the DomDocument. When accessing the element node, the namespace is not listed in the list of children. In the example above, `tw:Town=""` is in the list of children not `xmlns:tw="http://www.mysite.com/cities"`. To access the namespace during the DomDocument building use the method normalize first. Normalize write the namespace declaration at the appropriate place. If there is no previous declaration, it will be accessible as an attribute of this element otherwise it will be an attribute of one of the ancestors of the element.

```
createTextNode("My Company")
```

Creates a text node.

```
createComment("End of the card")
```

Produces
`<!--End of the card-->`

```
createCDATASection("<website><a href=\"www.mysite.com\">My
Company</a></website>")
```

Produces

```
<![CDATA[<website><a href="www.mysite.com">My
Company</a></website>]]>
```

```
createEntityReference("title")
```

Creates the entity reference `&title`.

```
createProcessingInstruction("xml-stylesheet","type=\"text/xsl\"
href=\"card.xsl\"")
```

Produces

```
<?xml-stylesheet type="text/xsl" href="card.xsl"?>
```

```
createDocumentType("Card",NULL,NULL,"<!ELEMENT Card
(lastname,firstname,company,location)>")
```

Produces

```
<!DOCTYPE Card [ <!ELEMENT Card (lastname , firstname ,
company , location)>]>
```

Only inline DTD are supported. The DTD has to been inserted in the DomDocument at an appropriate place.

```
createDocumentFragment
```

Is a method that creates a lighweight DomDocument. It represents a sub-tree of nodes that do not need to conform to well-formed XML rules. This makes DocumentFragment easier to manipulate than a DomDocument.

```
for i=1 to 5
  let node = doc.createelement("Card")
  call root.appendchild(node)
end for
```

This produces a sub-tree with 5 Card nodes that do not have any root node. Once the sub-tree is completed, it can be added to the DomDocument object like any other node.

Load and save methods

Object Methods

Name	Description
`normalize()`	Normalizes the entire Document. This method merges adjacent Text nodes, removes empty Text nodes and sets namespace declarations as if the document had been saved. See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document normalization. Throws an exception in case of errors, and updates status with an error code.
`load(` `  url STRING )`	Loads an XML Document into a DomDocument object from a file or an URL, where *url* is a valid URL or the name of the file. **Note:** Only the following kinds of URLs are supported: **http://**, **https://**, **tcp://**, **tcps://**, **file:///** and **alias://**. See FGLPROFILE Configuration for more details about URL mapping with aliases, and for proxy and security configuration. **Note:** See `setFeature()` to specify how the document can be loaded. **Note:** See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document loading. Throws an exception in case of errors, and updates status with an error code.
`loadFromString(` `  str STRING )`	Loads an XML Document into a DomDocument object from a string where *str* is the string to load. **(Not part of W3C API)**. **Note:** See `setFeature()` to specify how the document can be loaded. **Note:** See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document loading. Throws an exception in case of errors, and updates status with an error code.
`save(` `  url STRING )`	Saves a DomDocument object as an XML Document to a file or URL, where *url* is a valid URL or the name of the file. **Note:** Only the following kinds of URLs are supported: **http://**, **https://**, **tcp://**, **tcps://**, **file:///** and **alias://**. See FGLPROFILE Configuration for more details about URL mapping with aliases, and for proxy and security configuration. **Note:** See `setFeature()` to specify how the document can be saved. **Note:** See `getErrorsCount()` and `getErrorDescription()` to retrieve error

	messages related to XML document saving. Throws an exception in case of errors, and updates status with an error code.
`saveToString()` `RETURNING STRING`	Saves a DomDocument object as an XML Document to a string. Returns the string that will contain the resulting document. **(Not part of W3C API)**. **Note:** See `setFeature()` to specify how the document can be saved. **Note:** See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document saving. Throws an exception in case of errors, and updates status with an error code.

Usage

You can load an existing xml document. Before loading an xml document you need to create the DomDocument object.

A DomDocument can load files using different URI: http://, https://, tcp://, tcps://, file:// and alias://. Use `getErrorsCount()` and `getErrorDescription()` to display errors about the document loading.

```
load("data.xml")
load("http://www.w3schools.com/xml/cd_catalog.xml")
load("https://localhost:6394/ws/r/calculator?WSDL")
load("file:///data/cd_catalog.xml")
load("tcp://localhost:4242/")
load("tcps://localhost:4243/")
load("alias://demo")
```

where `demo` alias is defined in fglprofile as `ws.demo.url = "http://www.w3schools.com/xml/cd_catalog.xml"`

```
loadfromstring("<List><elt>First element</elt><elt>Second element</elt>
<elt>Third element</elt></List>")
```

> Produces a sub-tree with a root node `List` and three nodes `elt` and three `textnode`.

A DomDocument can be saved at different URI begining with: http://, https://, tcp://, tcps://, file:// and alias://. Use `getErrorsCount()` and `getErrorDescription()` to display errors about the document saving.

```
save("myfile.xml")
save("http://myserver:8080/data/save1.xml")
save("file:///data/save.xml")
save("tcp://localhost:4242/")
save("alias://test")
```

where `test` alias is defined in fglprofile as ws.test.url = "http://localhost:8080/data/save3.xml"

`saveToString` saves the DomDocument in a string. Use `getErrorsCount()` and `getErrorDescription()`to display errors about the document saving

`normalize` function emulates a DomDocument save and load. It can be called at any stage of the DomDocument building. This removes empty Text nodes and sets namespace declarations as if the document had been saved.

Configuration methods

Object Methods	
Name	**Description**
`setFeature( feature STRING, value STRING)`	Sets a feature for the DomDocument object, where *feature* is the name of a DomDocument Feature, and *value* is the value of a feature. Throws an exception in case of errors, and updates status with an error code.
`getFeature( feature STRING) RETURNING STRING`	Gets a feature for the DomDocument object, where *feature* is the name of the DomDocument Feature. Returns the value of the feature. Throws an exception in case of errors, and updates status with an error code.
`getXmlVersion() RETURNING STRING`	Returns the document version as defined in the XML document declaration, which is **1.0**. No other versions are supported.
`getXmlEncoding() RETURNING STRING`	Returns the document encoding as defined in the XML document declaration, or NULL if there is none.
`setXmlEncoding( enc STRING )`	Sets the XML document encoding in the XML declaration, or NULL. Throws an exception in case of errors, and updates status with an error code.
`isXmlStandalone() RETURNING INTEGER`	Returns whether the XML standalone attribute is set in the XML declaration. Returns TRUE if the standalone attribute in the XML declaration is set to yes.
`setXmlStandalone( alone INTEGER )`	Sets the XML standalone attribute in the XML declaration to yes or no in the XML declaration, or NULL.

Validation methods

Object Methods	
Name	**Description**
`validate()` `  RETURNING INTEGER`	Performs a DTD or XML Schema validation for this DomDocument object **(Not part of W3C API)**. Returns the number of validation errors, or zero if there are none. **Note:** See `setFeature()` to specify what kind of validation to do. **Note:** See `getErrorsCount()` and `getErrorDescription()`to retrieve error messages related to validation errors. Throws an exception in case of errors, and updates status with an error code.
`validateOneElement(` `  elt xml.DomNode )` `  RETURNING INTEGER`	Performs a DTD or XML Schema validation of an XML Element DomNode object **(Not part of W3C API)**; *node* is the XML Element DomNode to validate. Returns the number of validation errors, or zero if there are none. **Note:** See `setFeature()` to specify what kind of validation to do. **Note:** See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to validation errors. Throws an exception in case of errors, and updates status with an error code.

Error management methods

Object Methods	
Name	**Description**
`getErrorsCount()` `  RETURNING INTEGER`	Returns the number of errors encountered during the loading, the saving or the validation of an XML document **(Not part of W3C API)**. Returns the number of errors, or zero if there are none.
`getErrorDescription(` `  pos INTEGER )` `  RETURNING STRING`	Returns the error description at given position **(Not part of W3C API)**; *pos* is the position of the error description (index starts at 1). Returns a string with an error description. Throws an exception in case of errors, and

updates status with an error code.

Usage

```
for i=1 to doc.getErrorsCount()
  display "[", i, "] ", doc.getErrorDescription(i)
end for
```

Displays all the errors encountered in the save, load or validate of `doc` DomDocument.

To display other errors, use the global variable status to get the error code and err_get(status) or sqlca.sqlerrm to get the description of the error. See error code for more details.

Cautions

Whitespaces, line feeds and carriage returns between elements are represented as text nodes in memory. An XML document written in a single line and a human readable (pretty printed format) do not have the same representation in the DomDocument. Take this under account when navigating in the document.

If a DomNode is not attached to a DomDocument and not referenced by any variable it can be destroyed. If one child of this node is still referenced, this child is not destroyed but its parent and the others node of the sub-tree are destroyed. To check if a node is attached to a DomDocument use isAttached method.

DomDocument remains in memory if any of its node is still referenced in a variable.

DomDocument Features

Feature	Description
format-pretty-print	Formats the output by adding white space to produce a pretty-printed, indented, human-readable form. **Note:** Default value is FALSE.
comments	Defines whether the XML comments are kept during the load of a document into a DomDocument object. **Note:** default value is TRUE.
whitespace-in-element-	Defines whether XML Text nodes that can be considered "Ignorable" are kept during the load of an XML document into a DomDocument object.

content	**Note:** default value is TRUE.
cdata-sections	Defines whether XML CData nodes are kept or replaced by XML Text nodes during the load of an XML document into a DomDocument object. **Note:** default value is TRUE.
expand-entity-references	Defines whether XML EntityReference nodes are kept or replaced during the load of an XML document into a DomDocument object. **Note:** default value is TRUE.
validation-type	Defines what kind of validation should be performed. **Note:** Only **DTD** and **Schema** are allowed (default is Schema)
external-schemaLocation	Defines a list of namespace-qualified XML schemas to use for validation on a DomDocument object. **Note:** Value is a space-separated string of one or several pairs of strings representing the namespace URI of the schema, followed by its location. **Example : "http://tempuri/org/NS mySchema1.xsd http://www.4js.com mySchema2.xsd"**
external-noNamespaceSchemaLocation	Defines a list of XML schemas to use for validation on a DomDocument object. **Note:** Value is a space-separated string of one or several strings representing the location of a schema. **Example : "mySchema1.xsd mySchema2.xsd"**
schema-uriRecovery	Resolves or changes the location of an XML schema referenced by import tags and namespace URIs in other schemas. **Note:** Value is a space-separated string of one or several pairs of strings representing the namespace URI to recover, followed by its location. **Example:** "http://tempuri/org/NS myFile.xsd http://www.4js.com myFJS.xsd"

Examples

Example 1 : Create a namespace qualified document with processing instructions

To get following XML document on disk:

```
<?Target1 This is my first PI ?>
<MyPre:RootNode xmlns:MyPre="http://www.tempuri.org" >
  <MyPre:Element />
</MyPre:RootNode>
<?Target2 This is my last PI ?>
```

You must write following code:

```
IMPORT xml

MAIN
  DEFINE doc      xml.DomDocument
```

```
      DEFINE pi       xml.DomNode
      DEFINE node     xml.DomNode
      DEFINE elt      xml.DomNode


      # Create a document with an initial namespace qualified root
node
      LET doc =
xml.DomDocument.CreateNS("MyPre","RootNode","http://www.tempuri.org")
      # Create a Processing instruction
      LET pi = doc.createProcessingInstruction("Target1,"This is my
first PI")
      # And add it at the begining of the document
      CALL doc.prependDocumentNode(pi)
      # Create another Processing instruction
      LET pi = doc.createProcessingInstruction("Target2,"This is my
last PI")
      # And add it at the end of the document
      CALL doc.appendDocumentNode(pi)
      # Retrieve initial root node of the document
      LET elt = doc.getDocumentElement()
      # Create a new Element node
      LET node =
doc.createElement("MyPre","Element","http://www.tempuri.org")
      # And add it as child of the RootNode
      CALL elt.appendChild(node)
      # Then save the document on disk
      CALL doc.save("MyFile.xml")
    END MAIN
```

Example 2 : Validating a document against XML schemas or a DTD

Following code loads one or more XML schemas or uses an embedded DTD to validate against a XML document:

```
    IMPORT xml

    MAIN
        DEFINE location STRING
        DEFINE xmlfile  STRING
        DEFINE doc      xml.DomDocument
        DEFINE ind      INTEGER

        IF num_args()THEN
          # Checks the number of arguments
          CALL ExitHelp()
        ELSE
          LET doc = xml.DomDocument.Create()
          LET xmlfile = arg_val(num_args())
          IF num_args() == 2 AND arg_val(1) == "-dtd" THEN
            # User choosed DTD validation
            CALL doc.setFeature("validation-type","DTD")
          ELSE
```

```
            # User choosed XML Schema validation
            IF arg_val(1) == "-ns" THEN
              # Handle namespace qualified XML schemas
              IF num_args() MOD 2 != 0 THEN
                CALL ExitHelp()
              END IF
              FOR ind = 2 TO num_args()-1 STEP 2
                IF location IS NULL THEN
                  LET location = arg_val(ind) || " " ||
arg_val(ind+1)
                ELSE
                  LET location = location || " " || arg_val(ind)
|| " " || arg_val(ind+1)
                END IF
              END FOR
              TRY
                CALL doc.setFeature("external-
schemaLocation",location)
              CATCH
                FOR ind = 1 TO doc.getErrorsCount()
                    DISPLAY "Schema error ("||ind||")
:",doc.getErrorDescription(ind)
                END FOR
                EXIT PROGRAM (-1)
              END TRY
            ELSE
              # Handle unqualified XML schemas
              FOR ind = 1 TO num_args()-1
                IF location IS NULL THEN
                  LET location = arg_val(ind)
                ELSE
                  LET location = location || " " || arg_val(ind)
                END IF
              END FOR
              TRY
                CALL doc.setFeature("external-
noNamespaceSchemaLocation",location)
              CATCH
                FOR ind = 1 TO doc.getErrorsCount()
                  DISPLAY "Schema error ("||ind||")
:",doc.getErrorDescription(ind)
                END FOR
                EXIT PROGRAM (-1)
              END TRY
            END IF
          END IF
        END IF
        TRY
          # Load XML document from disk
          CALL doc.load(xmlfile)
        CATCH
          # Display errors if loading failed
          IF doc.getErrorsCount()>0 THEN
            FOR ind = 1 TO doc.getErrorsCount()
              DISPLAY "LOADING ERROR #"||ind||"
:",doc.getErrorDescription(ind)
```

```
            END FOR
            EXIT PROGRAM(-1)
          ELSE
            DISPLAY "Unable to load file :",xmlfile
            EXIT PROGRAM(-1)
          END IF
        END TRY
        TRY
          # Validate loaded document
          LET ind = doc.validate()
          IF ind == 0 THEN
            # Successful validation
            DISPLAY "OK"
          ELSE
            # Display validation errors
            FOR ind = 1 TO doc.getErrorsCount()
              DISPLAY "VALIDATING ERROR #"||ind||"
:",doc.getErrorDescription(ind)
            END FOR
            EXIT PROGRAM(-1)
          END IF
        CATCH
          DISPLAY "Unable to validate file :",xmlfile
          EXIT PROGRAM(-1)
        END TRY
      END MAIN

      # Display help
      FUNCTION ExitHelp()
        DISPLAY "Validator < -dtd | -ns [namespace schema]+ |
[schema]+ > xmlfile"
        EXIT PROGRAM
      END FUNCTION
```

Examples

$ fglrun Validator -dtd MyFile.xml Validates XML file using DTD embedded in the XML file

$ fglrun Validator Schema1.xsd Schema2.xsd MyFile.xml Validates unqualified XML file using two unqualified XML schemas

$ fglrun Validator -ns http://tempuri.org/one Schema1.xsd http://tempuri.org/two Schema2.xsd MyFile.xml Validates namespace qualified XML file using two namespace qualified XML schemas

The DomNode class

Summary:

- Syntax
- Methods
- Example
    - Counting the number of nodes in an XML document

See also: The Genero Web Services XML Extension Library

Syntax

The **DomNode** class provides methods to manipulate a node of a DomDocument object. You can create a DomNode object using creation methods in the DomDocument class. Notice that *status* is set to zero after a successful method call.

Syntax

```
xml.DomNode
```

Methods

- Navigation
- Manipulation
- Access
- Modifier
- Attributes
- Search

Navigation methods

Object Methods	
Name	**Description**
`getParentNode() RETURNING xml.DomNode`	Returns the parent DomNode object for this DomNode object, or NULL. In the case of a DomDocument node, this method will return NULL (parent is not a DomNode object) but `isAttached()` will return TRUE.
`getFirstChild() RETURNING xml.DomNode`	Returns the first child DomNode object for this XML Element DomNode object, or NULL.
`getFirstChildElement() RETURNING`	Returns the first XML Element child DomNode

`xml.DomNode`	object for this DomNode object, or NULL. **(Not part of W3C API)**.
`getLastChild() RETURNING xml.DomNode`	Returns the last child DomNode object for this XML Element DomNode object, or NULL.
`getLastChildElement() RETURNING xml.DomNode`	Returns the last child XML element DomNode object for this DomNode object, or NULL. **(Not part of W3C API)**.
`getNextSibling() RETURNING xml.DomNode`	Returns the DomNode object immediately following this DomNode object, or NULL.
`getNextSiblingElement() RETURNING xml.DomNode`	Returns the XML Element DomNode object immediately following this DomNode object, or NULL. **(Not part of W3C API)**.
`getPreviousSibling() RETURNING xml.DomNode`	Returns the DomNode object immediately preceding this DomNode object, or NULL.
`getPreviousSiblingElement() RETURNING xml.DomNode`	Returns the XML Element DomNode object immediately preceding this DomNode object, or NULL. **(Not part of W3C API)**.
`getOwnerDocument() RETURNING xml.DomDocument`	Returns the DomDocument object containing this DomNode object, or NULL.
`hasChildNodes() RETURNING INTEGER`	Returns TRUE if this node has child nodes; otherwise, returns FALSE.
`getChildrenCount() RETURNING INTEGER`	Returns the number of child DomNode objects for this DomNode object.
`getChildNodeItem(pos INTEGER) RETURNING xml.DomNode`	Returns the child DomNode object at a given position for this DomNode object. Throws an exception in case of errors, and updates status with an error code.

Manipulation methods

Object Methods	
Name	**Description**
`prependChildElement(name STRING) RETURNING xml.DomNode`	Creates and adds a child XML Element node to the beginning of the list of child nodes for this XML Element DomNode object **(Not part of W3C API)**; *name* is the name of the XML element to add. Returns the XML Element DomNode object, or NULL. Throws an exception in case of errors, and updates status with an error code.
`prependChildElementNS(prefix STRING,name STRING, ns STRING) RETURNING xml.DomNode`	Creates and adds a child namespace-qualified XML Element node to the beginning of the list of child nodes for this XML Element DomNode object **(Not part of W3C API)**; *name* is the

	name of the XML Element to add; *ns* is the namespace URI of the XML Element to add. Returns the XML Element DomNode object, or NULL. Throws an exception in case of errors, and updates status with an error code.
`addPreviousSibling(node xml.DomNode)`	Adds a DomNode object as the previous sibling of this DomNode object **(Not part of W3C API)**; *node* is the node to add. **Note:** The DomNode object node must be the child of an element or document node; otherwise, the operation fails. Throws an exception in case of errors, and updates status with an error code.
`addNextSibling(node xml.DomNode)`	Adds a DomNode object as the next sibling of this DomNode object **(Not part of W3C API)**; *node* is the node to add. **Note:** The DomNode object node must be the child of an element or document node; otherwise, the operation fails. Throws an exception in case of errors, and updates status with an error code.
`prependChild(node xml.DomNode)`	Adds a child DomNode object to the beginning of the child list for this DomNode object **(Not part of W3C API)**; *node* is the node to add. **Note:** The DomNode object node must be the child of an element or document node; otherwise, the operation fails. Throws an exception in case of errors, and updates status with an error code.
`appendChild(node xml.DomNode)`	Adds a child DomNode object to the end of the child list for this DomNode object; *node* is the node to add. **Note:** The DomNode object node must be the child of an element or document node; otherwise, the operation fails. Throws an exception in case of errors, and updates status with an error code.
`insertBeforeChild(node xml.DomNode,ref xml.DomNode)`	Inserts a DomNode object before an existing child DomNode object; *node* is the node to insert, *ref* is the reference node - the node before which the new node must be inserted. Throws an exception in case of errors, and updates status with an error code.
`insertAfterChild(node xml.DomNode,ref xml.DomNode)`	Inserts a DomNode object after an existing child DomNode object **(Not part of W3C API)**; *node* is the node to insert, *ref* is the reference node - the node after which the new node must be inserted. Throws an exception in case of errors, and

	updates status with an error code.
`removeChild(node xml.DomNode)`	Removes a child DomNode object from the list of child DomNode objects, where *node* is the node to remove. Throws an exception in case of errors, and updates status with an error code.
`replaceChild(new xml.DomNode,old xml.DomNode)`	Replaces an existing child DomNode with another child DomNode object, where *old* is the child to be replaced and *new* is the replacement child. Throws an exception in case of errors, and updates status with an error code.
`appendChildElement(name STRING) RETURNING xml.DomNode`	Creates and adds a child XML Element node to the end of the list of child nodes for this XML Element DomNode object **(Not part of W3C API)**; *name* is the XML Element name. Returns the XML Element DomNode object, or NULL. Throws an exception in case of errors, and updates status with an error code.
`appendChildElementNS(prefix STRING,name STRING, ns STRING) RETURNING xml.DomNode`	Creates and adds a child namespace qualified XML Element node to the end of the list of child nodes for this XML Element DomNode object **(Not part of W3C API)**; *prefix* is the prefix of the XML Element to add; *name* is the name of the XML Element to add; **ns** is the namespace URI of the XML Element to add. Returns the XML Element DomNode object, or NULL. Throws an exception in case of errors, and updates status with an error code.
`clone(deep INTEGER) RETURNING xml.DomNode`	Returns a duplicate DomNode object of this node. If *deep* is TRUE, child DomNode objects are cloned too; otherwise, only the DomNode itself is cloned. Returns a copy of this DomNode object, or NULL.* Throws an exception in case of errors, and updates status with an error code.

Access methods

Object Methods	
Name	**Description**
`getNodeType() RETURNING STRING`	Gets the XML type for this DomNode object; returns one of the XML DomNode types, or NULL.
`getLocalName() RETURNING STRING`	Gets the local name for this DomNode object. If

	DomNode has a qualified name, only the local part is returned.
`getNodeName() RETURNING STRING`	Gets the name for this DomNode object; returns the qualified name of this DomNode object, or NULL. If DomNode does not have a qualified name, the local part is returned.
`getNamespaceURI()RETURNING STRING`	Returns the namespace URI for this DomNode object, or NULL.
`getNodeValue() RETURNING STRING`	Returns the value for this DomNode object, or NULL.
`getPrefix() RETURNING STRING`	Returns the prefix for this DomNode object, or NULL.
`isAttached() RETURNING INTEGER`	Returns whether the node is attached to the XML document **(Not part of W3C API)**. Returns TRUE if this DomNode object is attached to a DomDocument object as a child and was not removed later on; otherwise it returns FALSE.

Modifier methods

Object Methods	
Name	**Description**
`setNodeValue(val STRING)`	Sets the node value for this DomNode object, where *val* is the node value. Throws an exception in case of errors, and updates status with an error code.
`setPrefix(prefix STRING)`	Sets the *prefix* for this DomNode object. **Note:** This method is only valid on namespace qualified Element or Attribute nodes. Throws an exception in case of errors, and updates status with an error code.
`toString() RETURNING STRING`	Returns a string representation of this DomNode object, or NULL **(Not part of W3C API)**. Throws an exception in case of errors, and updates status with an error code.

Attributes methods

Object Methods	
Name	**Description**
`hasAttribute(name STRING) RETURNING INTEGER`	Checks whether this XML Element DomNode object has the XML Attribute specified by *name.* Returns TRUE if an XML Attribute with the given name is carried by this XML Element DomNode object; otherwise, returns FALSE. Throws an exception in case of errors, and updates status with an error code.
`hasAttributeNS(name STRING,ns STRING) RETURNING INTEGER`	Checks whether a namespace qualified XML Attribute of a given name is carried by this XML Element DomNode object, where name the name of the XML Attribute to check ns: the namespace URI of the XML Attribute to check Returns TRUE if an XML Attribute with the given name and namespace URI is carried by this XML Element DomNode object, FALSE otherwise. Throws an exception in case of errors, and updates status with an error code.
`getAttributeNode(name STRING) RETURNING xml.DomNode`	Returns an XML Attribute DomNode object for this XML Element DomNode object, or NULL; *name* is the name of the attribute to retrieve. Throws an exception in case of errors, and updates status with an error code.
`getAttributeNodeNS(name STRING,ns STRING) RETURNING xml.DomNode`	Returns a namespace-qualified XML Attribute DomNode object for this XML Element DomNode object, or NULL; *name* is the name of the XML Attribute to retrieve and *ns* is the namespace URI of the XML Attribute to retrieve. Throws an exception in case of errors, and updates status with an error code.
`setAttributeNode(node xml.DomNode)`	Sets (or resets) an XML Attribute DomNode object to an XML Element DomNode object, , where *node* is the XML Attribute DomNode object to set. Throws an exception in case of errors, and updates status with an error code.
`setAttributeNodeNS(node xml.DomNode)`	Sets (or resets) a namespace-qualified XML Attribute DomNode object to an XML Element DomNode object, where *node* is the XML Attribute DomNode object to set. Throws an exception in case of errors, and updates status with an error code.
`getAttribute(name STRING) RETURNING STRING`	Returns the value of an XML Attribute for this XML Element DomNode object, where *name* is the name of the XML attribute to retrieve; returns the XML Attribute value, or NULL.

	Throws an exception in case of errors, and updates status with an error code.
`getAttributeNS(name STRING,ns STRING)` `  RETURNING STRING`	Returns the value of a namespace qualified XML Attribute for this XML Element DomNode object, where *name* is the name and *ns* is the namespace URI of the XML Attribute to retrieve; returns the XML Attribute value, or NULL. Throws an exception in case of errors, and updates status with an error code.
`setAttribute(name STRING,value STRING)`	Sets (or resets) an XML Attribute for this XML Element DomNode object, where *name* is the name of the XML Attribute and *val* is the value of the XML Attribute. Throws an exception in case of errors, and updates status with an error code.
`setAttributeNS(prefix STRING,name STRING,` `  ns STRING,value STRING)`	Sets (or resets) a namespace-qualified XML Attribute for this XML Element DomNode object, where *prefix* is the prefix of the XML Attribute, *name* is the name of the XML Attribute, *ns* is the namespace URI of the XML Attribute, and *val* is the value of the XML Attribute. Throws an exception in case of errors, and updates status with an error code.
`setIdAttribute(name STRING,isID INTEGER)`	Declare (or undeclare) the XML Attribute of given name to be of type ID. Use the value TRUE for the parameter *isID* to declare that attribute for being a user-determined ID attribute, false otherwise. **Note:** This affects the behavior of getElementById. Throws an exception in case of errors, and updates status with an error code.
`setIdAttributeNS(name STRING,` `ns STRING,isID INTEGER)`	Declare (or undeclare) the namespace-qualified XML Attribute of given name and namespace to be of type ID. Use the value TRUE for the parameter *isID* to declare that attribute for being a user-determined ID attribute, false otherwise. **Note:** This affects the behavior of getElementById. Throws an exception in case of errors, and updates status with an error code.
`removeAttribute(name STRING)`	Removes an XML Attribute for this XML Element DomNode object, where *name* is the name of the XML attribute to remove. Status is updated with an error code. Throws an exception in case of errors, and updates status with an error code.
`removeAttributeNS(name STRING,ns STRING)`	Removes a namespace qualified XML Attribute for this XML Element DomNode object, where

	name is the name and *ns* is the namespace URI of the XML Attribute to remove. Throws an exception in case of errors, and updates status with an error code.
`hasAttributes() RETURNING INTEGER`	Returns TRUE if this node has XML Attribute nodes; otherwise, returns FALSE.
`getAttributesCount() RETURNING INTEGER`	Returns the number of XML Attribute DomNode objects on this XML Element DomNode object. Throws an exception in case of errors, and updates status with an error code.
`getAttributeNodeItem(pos INTEGER)` `RETURNING Xml.DomNode`	Returns the XML Attribute DomNode object at a given position on this XML Element DomNode object, where *pos* is the position of the node to return (Index starts at 1). Returns the XML Attribute DomNode object at the given position, or NULL. Throws an exception in case of errors, and updates status with an error code.

Search methods

Object Methods	
Name	**Description**
`selectByXPath(expr STRING, NamespacesList ...) RETURNING xml.DomNodeList`	Returns a DomNodeList object containing all DomNode objects matching an XPath 1.0 expression **(Not part of W3C API)**; *expr* is the XPath 1.0 expression, *NamespacesList* is a list of prefixes bounded to namespaces in order to resolve qualified names in the XPath expression. This list must be filled with an even number of arguments, representing the prefix and it corresponding namespace. **Examples :** `selectByXPath("../../d:Record/*[last()]", "d", "http://defaultnamespace")` `selectByXPath("ns:Record", NULL)` `selectByXPath("ns1:Records/ns2:Record", "ns1", "http://namespace1", "ns2", "http://namespace2")` `selectByXPath("ns1:Record", "ns1")` is invalid because the namespace definition is missing. **Note:** If the namespaces list is NULL, the prefixes and namespaces defined in the document itself are used if available. **Note:** A namespace must be an absolute URI (ex 'http://', 'file://').

	Throws an exception in case of errors, and updates status with an error code.
`getElementsByTagName(name STRING)` `RETURNING xml.DomNodeList`	Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name, or NULL; *name* is the name of the XML Element tag to match, or "*" to match all tags. Throws an exception in case of errors, and updates status with an error code.
`getElementsByTagNameNS(name STRING,ns STRING)` `RETURNING xml.DomNodeList`	Returns a DomNodeList object containing all namespace-qualified XML Element DomNode objects with the same tag name and namespace, OR NULL. *name* is the name of the XML Element tag to match, or "*" to match all tags; *ns* is the namespace URI of the XML Element tag to match, or "*" to match any namespace. Throws an exception in case of errors, and updates status with an error code.
`isDefaultNamespace(ns STRING) RETURNING INTEGER`	Checks whether the specified namespace URI is the default namespace, where *ns* is the namespace URI to look for. Returns TRUE if the given namespace is the default namespace, FALSE otherwise. Throws an exception in case of errors, and updates status with an error code.
`lookupNamespaceURI(prefix STRING) RETURNING STRING`	Looks up the namespace URI associated to a prefix, starting from this node, where *prefix* is the prefix to look for; if NULL, the default namespace URI will be returned. Returns a namespace URI, or NULL. Throws an exception in case of errors, and updates status with an error code.
`lookupPrefix(ns STRING) RETURNING STRING`	Looks up the prefix associated to a namespace URI, starting from this node, where *ns* is the namespace URI to look for. Returns the prefix associated to this namespace URI, or NULL. Throws an exception in case of errors, and updates status with an error code.

Usage

The `getElementsByTagName` and `getElementsByTagNameNS` methods return a DomNodeList object, unlike the other methods that return a DomNode object. The DomNodeList is restricted to contain objects with the same tag name and/or namespace.

DomNode types

Type	Description
ELEMENT_NODE	The DomNode is an XML Element node.
ATTRIBUTE_NODE	The DomNode is an XML Attribute node.
TEXT_NODE	The DomNode is an XML Text node.
CDATA_SECTION_NODE	The DomNode is an XML CData Section node.
ENTITY_REFERENCE_NODE	The DomNode is an XML Entity Reference node.
PROCESSING_INSTRUCTION_NODE	The DomNode is an XML Processing Instruction node.
COMMENT_NODE	The DomNode is an XML Comment node.
DOCUMENT_TYPE_NODE	The DomNode is an XML DTD node.
DOCUMENT_FRAGMENT_NODE	The DomNode is an XML Document Fragment node.

Examples

Example 1 : Counting the number of nodes in an XML document

Following code counts the number of nodes of each type :

```
IMPORT XML

DEFINE  nbElt       INTEGER
DEFINE  nbAttr      INTEGER
DEFINE  nbComment   INTEGER
DEFINE  nbPI        INTEGER
DEFINE  nbTxt       INTEGER
DEFINE  nbCData     INTEGER

MAIN
  DEFINE document   Xml.DomDocument
  DEFINE ind        INTEGER
  # Handle arguments
  IF num_args() !=1 THEN
    CALL ExitHelp()
  END IF
  # Create document, load it, and count the nodes
  LET document = Xml.DomDocument.Create()
  CALL document.load(arg_val(1))
  CALL CountDoc(document)
  # Display result
  DISPLAY "Results: "
```

```
  DISPLAY "  Elements:  ",nbElt
  DISPLAY "  Attributes:",nbAttr
  DISPLAY "  Comments:  ",nbComment
  DISPLAY "  PI:        ",nbPI
  DISPLAY "  Texts:     ",nbTxt
  DISPLAY "  CData:     ",nbCData
END MAIN

FUNCTION CountDoc(d)
  DEFINE d    Xml.DomDocument
  DEFINE n    Xml.DomNode
  LET n = d.getFirstDocumentNode()
  WHILE (n IS NOT NULL)
    CALL Count(n)
    LET n = n.getNextSibling()
  END WHILE
END FUNCTION

FUNCTION Count(n)
  DEFINE n      Xml.DomNode
  DEFINE child  Xml.DomNode
  DEFINE next   Xml.DomNode
  DEFINE node   Xml.DomNode
  DEFINE ind    INTEGER
  DEFINE name   STRING

  IF n IS NOT NULL THEN
    IF n.getNodeType() == "COMMENT_NODE" THEN
      LET nbComment = nbComment + 1
    END IF
    IF n.getNodeType() == "ATTRIBUTE_NODE" THEN
      LET nbAttr = nbAttr + 1
    END IF
    IF n.getNodeType() == "PROCESSING_INSTRUCTION_NODE "
    THEN
      LET nbPI = nbPI + 1
    END IF
    IF n.getNodeType() == "ELEMENT_NODE" THEN
      LET nbElt = nbElt + 1
    END IF
    IF n.getNodeType() == "TEXT_NODE" THEN
      LET nbTxt = nbTxt +1
    END IF
    IF n.getNodeType() == "CDATA_SECTION_NODE" THEN
      LET nbCData = nbCData + 1
    END IF
    IF n.hasChildNodes() THEN
      LET name = n.getLocalName()
      LET child = n.getFirstChild()
      WHILE (child IS NOT NULL)
        CALL Count(child)
        LET child = child.getNextSibling()
      END WHILE
    END IF
    IF n.hasAttributes() THEN
      FOR ind = 1 TO n.getAttributesCount()
```

```
          LET node = n.getAttributeNodeItem(ind)
          CALL Count(node)
        END FOR
      END IF
    END IF
END FUNCTION

FUNCTION ExitHelp()
  DISPLAY "DomCount <xml>"
  EXIT PROGRAM
END FUNCTION
```

The DomNodeList class

Summary:

- Syntax
- Methods

See also: The Genero Web Services XML Extension Library

Syntax

The **DomNodeList** class provides methods to manipulate a list of DomNode objects. You can create a DomNodeList object using selection methods in the DomDocument and DomNode classes. The relationship between the DomNode objects in the list depends on the method used to create the DomNodeList object. Notice that *status* is set to zero after a successful method call.

Syntax

```
xml.DomNodeList
```

Methods

Object Methods	
Name	**Description**
`getCount() RETURNING INTEGER`	Returns the number of DomNode objects in a DomNodeList object.
`getItem(pos INTEGER) RETURNING xml.DomNode`	Returns the DomNode object at a given position in a DomNodeList object, where *pos* is the position of the DomNode object to return (Index starts at 1); Returns the DomNode object at the given position in this DomNodeList object, or NULL. Throws an exception in case of errors, and updates status with an error code.

The Stax Writer class

Summary:

- Syntax
- Methods
- Examples

See also: The Genero Web Services XML Extension Library

Syntax

The **StaxWriter** class provides methods compatible with StAX (Streaming API for XML), for writing XML documents. Notice that *status* is set to zero after a successful method call.

Syntax

```
xml.StaxWriter
```

Methods

- Creation
- Configuration
- Output
- Document
- NameSpace
- Nodes

Creation methods

Class Methods	
Name	**Description**
`xml.StaxWriter.create()` `  RETURNING` `xml.StaxWriter`	Constructor of a StaxWriter object; returns the StaxWriter object.

Configuration methods

Object Methods	
Name	**Description**
`setFeature( feature STRING, value STRING )`	Sets a feature of a StaxWriter object, where *feature* is name of a feature, and *value* is the value of the feature. The features can be changed at any time, but will only be taken into account at the beginning of a new stream (see `writeTo` or `writeToDocument`). Throws an exception in case of errors, and updates status with an error code.
`getFeature( feature STRING ) RETURNING STRING`	Gets a feature of a StaxWriter object, where *feature* is the name of a feature; returns the feature value. Throws an exception in case of errors, and updates status with an error code.

Output methods

Object Methods	
Name	**Description**
`writeTo( url STRING )`	Sets the output stream of the StaxWriter object to a file or an URL, and starts the streaming; *url* is a valid URL or the name of the file that will contain the resulting XML document. Throws an exception in case of errors, and updates status with an error code. **Note:** Only the following kinds of URLs are supported: **http://**, **https://**, **tcp://**, **tcps://**, **file:///** and **alias://**. See FGLPROFILE Configuration for more details about URL mapping with aliases, and for proxy and security configuration.
`writeToDocument( doc xml.DomDocument )`	Sets the output stream of the StaxWriter object to an Xml.DomDocument object, and starts the streaming; *doc* is the empty xml.DomDocument object that will contain the resulting XML document. Throws an exception in case of errors, and updates status with an error code.
`close()`	Closes the StaxWriter streaming, and releases all associated resources.

Usage

```
writeTo("printerList.xml")
writeTo("http://myserver:1100/documents/ptinterList.xml")
writeTo("https://myserver:1100/documents/ptinterList.xml")
writeTo("alias://printerlist")
```

where `printerlist`alias is defined in fglprofile as `ws.printerlist.url = "http://myserver:1100/documents/ptinterList.xml".`

Document methods

Object Methods	
Name	**Description**
`startDocument( encoding STRING, version STRING, standalone INTEGER )`	Writes an XML declaration to the StaxWriter stream, where encoding is the encoding of the XML declaration, or NULL to use the default UTF-8 encoding; version is the XML version of the XML declaration, or NULL to use the default 1.0 version; standalone when TRUE sets the standalone of the XML declaration to "yes", when FALSE sets it to "no" or NULL. Throws an exception in case of errors, and updates status with an error code.
`endDocument()`	Closes any open tags and writes corresponding end tags. Throws an exception in case of errors, and updates status with an error code.
`dtd( data STRING )`	Writes a DTD to the StaxWriter stream, where **data** is a string representing a valid DTD, cannot be NULL. Throws an exception in case of errors, and updates status with an error code.

Usage

```
startDocument("utf-8","1.0",true)
```
produces
```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

dtd("note [<!ENTITY writer \"Donald Duck.\">]")
```

NameSpace methods

Object Methods	
Name	**Description**
`setPrefix( prefix STRING, ns STRING )`	Binds a namespace URI to a prefix. The prefix scope is the current START_ELEMENT / END_ELEMENT pair; *prefix* is the prefix to be bind to the URI, cannot be NULL; *ns* is the namespace URI to be bind to the prefix, cannot be

	NULL. Throws an exception in case of errors, and updates status with an error code.
`setDefaultNamespace( defaultNS STRING )`	Binds a namespace URI to the default namespace. The default namespace scope is the current START_ELEMENT / END_ELEMENT pair; *defaultNS* is the URI to bind to the default namespace, cannot be NULL. Throws an exception in case of errors, and updates status with an error code.
`declareNamespace( prefix STRING, ns STRING )`	Binds a namespace URI to a prefix, and forces the output of the XML namespace definition to the StaxWriter stream. The stream must point to a START_ELEMENT, and the prefix scope is the current START_ELEMENT / END_ELEMENT pair. *prefix* is the prefix to be bind to the URI, cannot be NULL; *ns* is the URI to bind to the default namespace, cannot be NULL. Throws an exception in case of errors, and updates status with an error code.
`declareDefaultNamespace( defaultNS STRING )`	Binds a namespace URI to the default namespace, and forces the output of the default XML namespace definition to the StaxWriter stream. The stream must point to a START_ELEMENT, and the prefix scope is the current START_ELEMENT / END_ELEMENT pair; *defaultNS* is the URI to bind to the default namespace, cannot be NULL. Throws an exception in case of errors, and updates status with an error code.

Nodes methods

Object Methods	
Name	**Description**
`startElement( name STRING )`	Writes an XML start element to the StaxWriter stream. All startElement methods open a new scope and set the stream to a START_ELEMENT; then, writing the corresponding endElement causes the scope to be closed. *name* is the local name of the XML start element, cannot be NULL. Throws an exception in case of errors, and updates status with an error code.
`startElementNS( name STRING, ns STRING )`	Writes a namespace-qualified XML start element to the StaxWriter stream. All startElementNS methods open a new scope and set the stream to a START_ELEMENT; then, writing the corresponding endElement causes the scope to

	be closed. *name* is the local name of the XML start element, cannot be NULL; *ns* is the namespace URI of the XML start element, cannot be NULL. If namespace URI has not been bound to a prefix with one of the functions `setPrefix()`, `declareNamespace()`, `setDefaultNamespace()` or `declareDefaultNamespace()`, the operation will fail with an exception. Throws an exception in case of errors, and updates status with an error code.
`emptyElement( name STRING )`	Writes an empty XML element to the StaxWriter stream, where *name* is the local name of the XML empty element, cannot be NULL. Throws an exception in case of errors, and updates status with an error code.
`emptyElementNS( name STRING, ns STRING )`	Writes an empty namespace qualified XML element to the StaxWriter stream, where *name* is the local name of the XML empty element, cannot be NULL; *ns* is the namespace URI of the XML empty element, cannot be NULL. If namespace URI has not been bound to a prefix with one of the functions: `setPrefix()`, `declareNamespace()`, `setDefaultNamespace()` or `declareDefaultNamespace()`, operation will fail with an exception. Throws an exception in case of errors, and updates status with an error code.
`endElement()`	Writes an end tag to the StaxWriter stream relying on the internal state to determine the prefix and local name of the last START_ELEMENT. Throws an exception in case of errors, and updates status with an error code.
`attribute( name STRING, value STRING )`	Writes an XML attribute to the StaxWriter stream, where name is the local name of the XML attribute, cannot be NULL; *value* is the value of the XML attribute, cannot be NULL. Attributes can only be written on the StaxWriter stream if it points to a START_ELEMENT or an EMPTY_ELEMENT, otherwise the operation will fail with an exception; that is, this method can only be called after a `startElement()`, `startElementNS()`, `emptyElement()`, `emptyElementNS()` or `attribute()` and `attributeNS()`. Throws an exception in case of errors, and updates status with an error code.
`attributeNS( name STRING, ns STRING, value STRING )`	Writes an XML namespace qualified attribute to the StaxWriter stream, where *name* is the local name of the XML attribute, cannot be NULL; *ns:* the namespace URI of the XML attribute, cannot be NULL; *value* is the value of the XML attribute, cannot be NULL. Attributes can only be written on the StaxWriter stream if it points to a START_ELEMENT or an EMPTY_ELEMENT,

	otherwise the operation will fail with an exception; that is, this method can only be called after a `startElement()`, `startElementNS()`, `emptyElement()`, `emptyElementNS()` or `attribute()` and `attributeNS()`. If namespace URI has not been bound to a prefix with one of these functions : `setPrefix()`, `declareNamespace()`, `setDefaultNamespace()` or `declareDefaultNamespace()`, the operation will fail with an exception. Throws an exception in case of errors, and updates status with an error code.
`processingInstruction( target STRING, data STRING )`	Writes an XML ProcessingInstruction to the StaxWriter stream, where *target* is the target of the Processing Instruction, cannot be NULL; *data* is the data of the Processing Instruction, or NULL. Throws an exception in case of errors, and updates status with an error code.
`comment( data STRING )`	Writes an XML comment to the StaxWriter stream where *data* is the data in the XML comment, or NULL. Throws an exception in case of errors, and updates status with an error code.
`characters( text STRING )`	Writes an XML text to the StaxWriter stream, where *tex*t is the value to write, cannot be NULL. Throws an exception in case of errors, and updates status with an error code.
`cdata( data STRING )`	Writes an XML CData to the StaxWriter stream, where *data* is the data contained in the CData section, or NULL. Throws an exception in case of errors, and updates status with an error code.
`entityRef( name STRING )`	Writes an XML EntityReference to the StaxWriter stream, where name is the name of the entity, cannot be NULL. Throws an exception in case of errors, and updates status with an error code.

StaxWriter Features

Feature	Description
format-pretty-print	Formats the output by adding whitespace to produce a pretty-printed, indented, human-readable form. **Note:** default value is FALSE.
smart-ending-elements	Outputs each tag closed with an endElement() call, as empty elements if they have no children. **Note:** default value is FALSE.

Examples

```
    IMPORT xml

    FUNCTION save(file)
      DEFINE file STRING
      DEFINE writer xml.StaxWriter
      TRY
        LET writer = xml.StaxWriter.Create()
        CALL writer.setFeature("format-pretty-print",TRUE)
        CALL writer.writeTo(file)
        CALL writer.startDocument("utf-8","1.0",true)
        CALL writer.comment("This is my first comment using a
stax writer")
        CALL writer.setPrefix("c","http://www.4js.com/c")
        CALL writer.setPrefix("d","http://www.4js.com/d")
        CALL writer.setDefaultNamespace("http://www.4js.com/d")
        CALL writer.startElementNS("root",
"http://www.4js.com/d")
        CALL writer.attribute("attr1","value1")
        CALL writer.attribute("attr2","value2")
        CALL writer.attributeNS("attr3",
"http://www.4js.com/d","value3")
        CALL writer.comment("This is a comment using a stax
writer")
        CALL writer.startElementNS("eltA",
"http://www.4js.com/d")
        CALL writer.CData("<this is a CData section>")
        CALL writer.endElement()
        CALL writer.startElementNS("eltB",
"http://www.4js.com/c")
        CALL writer.characters("Hello world, I'm from the Four
J's development team")
        CALL writer.entityRef("one")
        CALL writer.endElement()
        CALL writer.processingInstruction("command1","do what
you want")
        CALL writer.endElement()
        CALL writer.comment("This is my last comment using a
stax writer")
        CALL writer.endDocument()
        CALL writer.close()
        RETURN TRUE
      CATCH
        DISPLAY "StaxWriter ERROR :",STATUS, SQLCA.SQLERRM
        RETURN FALSE
      END TRY
    END FUNCTION
```

The StaxReader class

Summary:

- Syntax
- Methods
- Examples

See also: The Genero Web Services XML Extension Library

Syntax

The **StaxReader** class provides methods compatible with StAX (Streaming API for XML), for reading XML documents. Notice that *status* is set to zero after a successful method call.

Syntax:

```
xml.StaxReader
```

Methods:

- Creation
- Configuration
- Input
- Access
- Document
- Nodes
- Processing Instructions
- Attributes
- Namespaces
- Navigation

Creation Methods

Class Methods	
Name	**Description**
`xml.StaxReader.Create()` `  RETURNING` `xml.StaxReader`	Constructor of a StaxReader object; returns the object.

Configuration Methods

Object Methods	
Name	**Description**
`setFeature( feature STRING, value STRING )`	Sets a feature of a StaxReader object, where *feature* is the name of a feature, and *value* is the value of the feature. The features can be changed at any time, but will only be taken into account at the beginning of a new stream (see `readFrom` or `readFromDocument`). Throws an exception in case of errors, and updates status with an error code.
`getFeature( feature STRING ) RETURNING STRING`	Gets a feature of a StaxReader object; where *feature* is the name of a feature. Returns the feature value. **Status** is updated with an error code.

Input Methods

Object Methods	
Name	**Description**
`readFrom( url STRING)`	Sets the input stream of the StaxReader object to a file or an URL, and starts the streaming; *url* is a valid URL or the name of the file to read. Throws an exception in case of errors, and updates status with an error code. **Note:** Only the following kinds of URLs are supported: **http://**, **https://**, **tcp://**, **tcps://**, **file:///** and **alias://**. See FGLPROFILE Configuration for more details about URL mapping with aliases, and for proxy and security configuration.
`readFromDocument( doc xml.DomDocument )`	Sets the input stream of the StaxReader object to a DomDocument object, and starts the streaming; *doc* is an xml.DomDocument object that contains an XML document. Raises a 4GL exception in case of errors, and updates status with an error code.
`close()`	Closes the StaxReader streaming, and releases all associated resources.

Access Methods

Object Methods	
Name	**Description**
`getEventType()` `RETURNING STRING`	Returns a string that indicates the type of event the cursor of the StaxReader object is pointing to. **Status** is updated with an error code.
`hasName()` `RETURNING INTEGER`	Checks whether the StaxReader cursor points to a node with a name. Returns TRUE if the current XML node has a name, FALSE otherwise. This method returns TRUE for START_ELEMENT and END_ELEMENT, FALSE for all other nodes. Throws an exception in case of errors, and updates status with an error code.
`hasText()` `RETURNING INTEGER`	Checks whether the StaxReader cursor points to a node with a text value. Returns TRUE if the current XML node has a text value, FALSE otherwise. This method returns TRUE for CHARACTERS, SPACE, CDATA, COMMENT, ENTITY_REFERENCE and DTD, FALSE for all other nodes. Throws an exception in case of errors, and updates status with an error code.
`isEmptyElement()` `RETURNING INTEGER`	Checks whether the StaxReader cursor points to an empty element node. Returns TRUE if the current XML element node has no children, FALSE otherwise. Throws an exception in case of errors, and updates status with an error code.
`isStartElement()` `RETURNING INTEGER`	Checks whether the StaxReader cursor points to a start element node. Returns TRUE if the current XML node is a start element node, FALSE otherwise. Throws an exception in case of errors, and updates status with an error code.
`isEndElement()` `RETURNING INTEGER`	Checks whether the StaxReader cursor points to an end element node. Returns TRUE if the current XML node is an end element node, FALSE otherwise. Throws an exception in case of errors, and updates status with an error code.
`isCharacters()` `RETURNING INTEGER`	Checks whether the StaxReader cursor points to a character node. Returns TRUE if the current XML node is a character node, FALSE otherwise. Throws an exception in case of errors, and updates status with an error code.
`isIgnorableWhitespace()` `RETURNING INTEGER`	Checks whether the StaxReader cursor points to ignorable whitespace. Returns TRUE if the current XML node is an ignorable character node, FALSE otherwise. Throws an exception in case of errors, and updates status with an error code.

Document Methods

Object Methods	
Name	**Description**
`getEncoding()` `RETURNING STRING`	Returns the document encoding defined in the XML Document declaration, or NULL. Throws an exception in case of errors, and updates status with an error code.
`getVersion()` `RETURNING STRING`	Returns the document version defined in the XML Document declaration, or NULL. Throws an exception in case of errors, and updates status with an error code.
`isStandalone()` `RETURNING STRING`	Checks whether the document standalone attribute defined in the XML Document declaration is set to yes. Returns TRUE if the standalone attribute in the XML declaration is set to yes, FALSE otherwise. Throws an exception in case of errors, and updates status with an error code.
`standaloneSet()` `RETURNING STRING`	Checks whether the document standalone attribute is defined in the XML Document declaration. Returns TRUE if the standalone attribute in the XML declaration is set, FALSE otherwise. Throws an exception in case of errors, and updates status with an error code.

Nodes Methods

Object Methods	
Name	**Description**
`getPrefix()` `RETURNING STRING`	Returns the prefix of the current XML node, or NULL. Throws an exception in case of errors, and updates status with an error code.
`getLocalName()` `RETURNING STRING`	Returns the local name of the current XML node, or NULL. Throws an exception in case of errors, and updates status with an error code.
`getName()` `RETURNING STRING`	Returns the qualified name of the current XML node, or NULL. Throws an exception in case of errors, and updates status with an error code.

`getNamespace()` `RETURNING STRING`	Returns the namespace URI of the current XML node, or NULL. Throws an exception in case of errors, and updates status with an error code.
`getText()` `RETURNING STRING`	Returns as a string the value of the current XML node, or NULL. This method is only valid on CHARACTERS, CDATA, SPACE, COMMENT, DTD and ENTITY_REFERENCE nodes. For an ENTITY_REFERENCE, this method returns the replacement value, or NULL if none. Throws an exception in case of errors, and updates status with an error code.

Processing Instructions Methods

Object Methods	
Name	**Description**
`getPITarget()` `RETURNING STRING`	Returns the target part of an XML ProcessingInstruction node, or NULL. This method is only valid on a PROCESSING_INSTRUCTION node. Throws an exception in case of errors, and updates status with an error code.
`getPIData()` `RETURNING STRING`	Returns the data part of an XML ProcessingInstruction node,or NULL. This method is only valid on a PROCESSING_INSTRUCTION node. Throws an exception in case of errors, and updates status with an error code.

Attributes Methods

Object Methods	
Name	**Description**
`getAttributeCount()` `RETURNING INTEGER`	Returns the number of XML attributes defined on the current XML node. Returns the number of attributes defined on the current XML node, or zero. This method is only valid on a START_ELEMENT node. Throws an exception in case of errors, and updates status with an error code.
`getAttributeLocalName(` `pos INTEGER )`	Returns the local name of an XML attribute defined at a given position on the current XML node, where *pos* is the

`RETURNING STRING`	position of the attribute to return (Index starts at 1). Returns the local name of an attribute defined at given position on the current XML node, or NULL. **Note:** This method is only valid on a START_ELEMENT node. Throws an exception in case of errors, and updates status with an error code.
`getAttributeNamespace( pos INTEGER ) RETURNING STRING`	Returns the namespace URI of an XML attribute defined at a given position on the current XML node; *pos* is the position of the attribute to return (Index starts at 1). Returns the namespace URI of an attribute defined at the given position on the current XML node, or NULL. **Note:** This method is only valid on a START_ELEMENT node. Throws an exception in case of errors, and updates status with an error code.
`getAttributePrefix( pos INTEGER ) RETURNING STRING`	Returns the prefix of an XML attribute defined at a given position on the current XML node; *pos* is the position of the attribute to return (Index starts at 1). Returns the prefix of an attribute defined at the given position on the current XML node, or NULL. **Note:** This method is only valid on a START_ELEMENT node. Throws an exception in case of errors, and updates status with an error code.
`getAttributeValue( pos INTEGER ) RETURNING STRING`	Returns the value of an XML attribute defined at a given position on the current XML node; *pos* is the position of the attribute to return (Index starts at 1). Returns the value of an attribute defined at the given position on the current XML node, or NULL. **Note:** This method is only valid on a START_ELEMENT node. Throws an exception in case of errors, and updates status with an error code.
`findAttributeValue( name STRING, ns STRING ) RETURNING STRING`	Returns the value of an XML attribute of a given name and/or namespace on the current XML node, or NULL; *name* is the name of the attribute to retrieve, cannot be NULL; *ns* is the namespace URI of the attribute to retrieve, or NULL if the attribute is not namespace-qualified. **Note:** This method is only valid on a START_ELEMENT node. Throws an exception in case of errors, and updates status with an error code.

Namespace Methods

Object Methods	
Name	**Description**
`lookupNamespace( prefix STRING ) RETURNING STRING`	Looks up the namespace URI associated with a given prefix starting from the current XML node the StaxReader cursor is pointing to, where *prefix* is the prefix to look for; if NULL the default namespace URI will be returned. Returns the namespace URI associated with the prefix, or NULL if there is none. Throws an exception in case of errors, and updates status with an error code.
`lookupPrefix( ns STRING ) RETURNING STRING`	Looks up the prefix associated with a given namespace URI, starting from the current XML node the StaxReader cursor is pointing to, where *ns* is the namespace URI to look for, cannot be NULL. Returns the prefix associated with this namespace URI, or NULL if there is none. Throws an exception in case of errors, and updates status with an error code.
`getNamespaceCount() RETURNING INTEGER`	Returns the number of namespace declarations defined on the current XML node, or zero. **Note:** This method is only valid on a START_ELEMENT node. Throws an exception in case of errors, and updates status with an error code.
`getNamespacePrefix( pos INTEGER ) RETURNING STRING`	Returns the prefix of a namespace declaration defined at a given position on the current XML node, or NULL; *pos* is the position of the namespace declaration (Index starts at 1). **Note:** This method is only valid on a START_ELEMENT node. Throws an exception in case of errors, and updates status with an error code.
`getNamespaceURI( pos INTEGER ) RETURNING STRING`	Returns the URI of a namespace declaration defined at a given position on the current XML node, or NULL; *pos* is the position of the namespace declaration (Index starts at 1). **Note:** This method is only valid on a START_ELEMENT node. Throws an exception in case of errors, and updates status with an error code.

Navigation Methods

Object Methods	
Name	**Description**

`hasNext()` `RETURNING INTEGER`	Checks whether the StaxReader cursor can be moved to a XML node next to it. Returns TRUE if there is still an XML node in the stream, FALSE otherwise. Throws an exception in case of errors, and updates status with an error code.
`next()`	Moves the StaxReader cursor to the next XML node. Raises a 4GL exception in case of errors, and updates status with an error code.
`nextTag()`	Moves the StaxReader cursor to the next XML open or end tag. The cursor points to the end of the document if there is no tag any longer. Raises a 4GL exception in case of errors, and updates status with an error code.
`nextSibling()`	Moves the StaxReader cursor to the immediate next sibling XML Element of the current node, skipping all its child nodes. The cursor points to the parent end tag if there are no siblings any longer. Raises a 4GL exception in case of errors, and updates status with an error code.

Examples

```
IMPORT xml

FUNCTION parse(file)
  DEFINE file     String
  DEFINE event    String
  DEFINE ret      INTEGER
  DEFINE ind      INTEGER
  DEFINE reader   xml.StaxReader
  TRY

    LET reader=xml.StaxReader.Create()
    CALL reader.readFrom(file)

    WHILE (true)
      LET event=reader.getEventType()
      CASE event
        WHEN "START_DOCUMENT"
          DISPLAY "Document reading started"
          DISPLAY "XML Version  : ",reader.getVersion()
          DISPLAY "XML Encoding : ",reader.getEncoding()
          IF reader.standaloneSet() THEN
            IF reader.isStandalone() THEN
              DISPLAY "Standalone   : yes"
            ELSE
              DISPLAY "Standalone   : no"
            END IF
          END IF
        WHEN "END_DOCUMENT"
```

```
            DISPLAY "Document reading finished"
          WHEN "START_ELEMENT"
            IF reader.isEmptyElement() THEN
              DISPLAY "<"||reader.getName()||"/>"
            ELSE
              DISPLAY "<"||reader.getName()||">"
            END IF
            FOR ind=1 TO reader.getNamespaceCount()
              DISPLAY "xmlns:"||reader.getNamespacePrefix(ind)||"="||
              reader.getNamespaceURI(ind)
            END FOR
            FOR ind=1 TO reader.getAttributeCount()
              IF reader.getAttributePrefix(ind) THEN
                DISPLAY
reader.getAttributePrefix(ind)||":"||reader.getAttributeLocalName(ind)||
         "="||reader.getAttributeValue(ind)
              ELSE
                DISPLAY
reader.getAttributeLocalName(ind)||"="||reader.getAttributeValue(ind)
              END IF
            END FOR
          WHEN "END_ELEMENT"
            DISPLAY "</"||reader.getName()||">"
          WHEN "CHARACTERS"
            IF reader.hasText() AND NOT reader.isIgnorableWhitespace()
            THEN
              DISPLAY "CHARACTERS :",reader.getText()
            END IF
          WHEN "COMMENT"
            IF reader.hasText() THEN
              DISPLAY "Comment :",reader.getText()
            END IF
          WHEN "CDATA"
            IF reader.hasText() THEN
              DISPLAY "CDATA :", reader.getText()
            END IF
          WHEN "PROCESSING_INSTRUCTION"
            DISPLAY "PI :",reader.getPITarget(),reader.getPIData()
          WHEN "ENTITY_REFERENCE"
            DISPLAY "Entity name :",reader.getName()
          OTHERWISE
            DISPLAY "Unknown "||event||" node"
        END CASE
        IF reader.hasNext() THEN
          CALL reader.next()
        ELSE
          CALL reader.close()
          EXIT WHILE
        END IF
      END WHILE
    CATCH
      DISPLAY "StaxReader ERROR :",STATUS||" ("||SQLCA.SQLERRM||")"
    END TRY
  END FUNCTION
```

StaxReader event types

Type	Description	XML sample
START_DOCUMENT	StaxReader cursor points to the beginning of the XML document.	`<?xml version="1.0" standalone="no"?>`
END_DOCUMENT	StaxReader cursor has reached the end of the XML document. **Note :** No additional parsing operation will succeed.	
START_ELEMENT	StaxReader cursor points to an XML start element or empty element node.	`<p:elt attr="val">` or `<p:elt attr="val"/>`
END_ELEMENT	StaxReader cursor points to an XML end element node.	`</p:elt>`
CHARACTERS	StaxReader cursor points to an XML text node.	`... eltA/>This is text<eltB ...`
CDATA	StaxReader cursor points to an XML CData node.	`<![CDATA[<Hello, world!>]]>`
SPACE	StaxReader cursor points to an XML text node containing only whitespaces.	`... eltA/>  <eltB ...`
COMMENT	StaxReader cursor points to an XML comment node.	`<!-- a comment -->`
DTD	StaxReader cursor points to a DTD string.	`<!DOCTYPE A [ <!ELEMENT B (C+)> ]>`
ENTITY_REFERENCE	StaxReader cursor points to an XML entity reference node.	`&ref;`
PROCESSING_INSTRUCTION	StaxReader cursor points to an XML processing instruction node.	`<?target data?>`
ERROR	StaxReader cursor points to an unexpected XML node.	

StaxReader Features

Feature	Description
expand-entity-references	Defines whether XML EntityReference nodes are kept or replaced during the parsing of an XML document. **Note:** default value is TRUE.

The Serializer class

Summary:

- Syntax
- Methods

See also: The Genero Web Services XML Extension Library

Syntax

The **Serializer** class provides methods to manage options for the serializer engine, and to use the serializer engine to serialize variables and XML element nodes. This class is a static class and does not have to be instantiated. Notice that *status* is set to zero after a successful method call.

Syntax

```
xml.Serializer
```

Methods

Class Methods	
Name	**Description**
`xml.Serializer.SetOption(` `  flag STRING,` `  value STRING )`	Sets a global option value for the serializer engine, where *flag* is the option flag, and *value* is the value of the flag. Throws an exception in case of errors, and updates status with an error code.
`xml.Serializer.GetOption(` `  flag STRING )` `  RETURNING STRING`	Gets a global option value from the serializer engine, where *flag* is the option flag. Returns the value of the flag. Throws an exception in case of errors, and updates status with an error code.
`xml.Serializer.VariableToStax(` `  var VARIABLE,` `  stax xml.StaxWriter )`	Serializes a 4GL variable into an XML element node using a StaxWriter object, where *var* is any 4GL variable with optional XML mapping attributes, and *stax* is a StaxWriter object. The resulting XML element node of the serialization process will be added at the current cursor position of the StaxWriter object. Throws an exception in case of errors, and updates status with an error code.

```
xml.Serializer.StaxToVariable(
  stax xml.StaxReader,
  var VARIABLE )
```

Serializes an XML element node into a 4GL variable using a StaxReader object, where *stax* is a StaxReader object where the cursor points to an XML Element node, and *var* is any 4GL variable with optional XML mapping attributes.
Throws an exception in case of errors, and updates status with an error code.

```
xml.Serializer.VariableToDom(
  var VARIABLE,
  node xml.DomNode )
```

Serializes a 4GL variable into an XML element node using a DomNode object, where *var* is any 4GL variable with optional XML mapping attributes, and *node* is a DomNode object of type ELEMENT_NODE or DOCUMENT_FRAGMENT_NODE.
The resulting XML element node of the serialization process will be appended to the last child of the given node.
Throws an exception in case of errors, and updates status with an error code.

```
xml.Serializer.DomToVariable(
  node xml.DomNode,
  var VARIABLE )
```

Serializes an XML element node into a 4GL variable using a DomNode object, where *node* is a DomNode object of type ELEMENT_NODE, and *var* is any 4GL variable with optional XML mapping attributes.
Throws an exception in case of errors, and updates status with an error code.

```
xml.Serializer.VariableToSoapSection5(
  var VARIABLE,
  node xml.DomNode )
```

Serializes a 4GL variable into an XML element node in Soap Section 5 encoding, where *var* is any 4GL variable with optional XML mapping attributes, and *node* is a DomNode object of type ELEMENT_NODE or DOCUMENT_FRAGMENT_NODE.
The resulting XML element node of the serialization process will be appended to the last child of the given node.
Throws an exception in case of errors, and updates status with an error code.

```
xml.Serializer.SoapSection5ToVariable(
  node xml.DomNode,
  var VARIABLE )
```

Serializes an XML element node into a 4GL variable in Soap Section 5 encoding, where *node* is a DomNode object of type ELEMENT_NODE, and *var* is any 4GL variable with optional XML mapping attributes.
Throws an exception in case of errors, and updates status with an error code.

```
xml.Serializer.CreateXmlSchemas(
  var VARIABLE,
```

Creates XML schemas corresponding to the given variable **var,** and fills the

`ar DYNAMIC ARRAY OF xml.DomDocument )`	dynamic array **ar** with xml.DomDocument objects each representing an XML schema. Throws an exception in case of errors, and updates status with an error code.

Serialization option flags

Flag	Description
xml_ignoretimezone	Defines whether, during the marshalling and un-marshalling process of a BDL DATETIME data type, the Serializer should ignore the time zone information. **Note:** A value of zero means FALSE (the default value is FALSE). Throws an exception in case of errors, and updates status with an error code.
xml_usetypedefinition	Defines whether the Serializer must specify the type of data during serialization. (This will add an "xsi:type" attribute to each XML data type.) **Note:** A value of zero means FALSE (the default value is FALSE). Throws an exception in case of errors, and updates status with an error code.

The XML Library Error Codes

Code	Description
0	No error.
-15600	Operation failed.
-15601	Name cannot be NULL.
-15602	Namespace cannot be NULL.
-15603	Prefix cannot be NULL.
-15604	Value cannot be NULL.
-15605	Node cannot be NULL.
-15606	Text cannot be NULL.
-15607	Target of a processing instruction cannot be NULL.
-15608	Name of an entity reference cannot be NULL.
-15609	XPath expression cannot be NULL.
-15610	Filename cannot be NULL.
-15611	Document cannot be NULL.
-15612	DTD string cannot be NULL.
-15613	Stax cannot be NULL.
-15614	Malformed XML name.
-15615	Malformed XML string.
-15616	Malformed XML prefix.
-15617	Malformed XML namespace.
-15618	Bad validation type.
-15619	No XML schema found.
-15620	No DTD schema found.
-15621	Feature or option cannot be NULL.
-15622	Feature or option is unsupported.
-15623	Feature or option value is invalid.
-15624	Node is not part of the document.
-15625	Node does not have the correct parent node.
-15626	Node is already linked to another node.
-15627	Cannot add a node to itself.
-15628	Index is out of bounds.
-15629	StaxWriter runtime exception, see SQLCA.SQLERRM for mode details.
-15630	StaxReader runtime exception, see SQLCA.SQLERRM for mode details.
-15631	Serializer runtime exception, see SQLCA.SQLERRM for mode details.
-15632	Document loading runtime exception, see getErrorDescription() for more

details.

-15633 Document saving runtime exception, see getErrorDescription() for more details.

-15634 Invalid encoding.

-15635 PublicID of a DTD cannot be set with a SystemID.

-15636 Undefined namespace prefix in the XPath expression.

-15637 XPath expression error.

-15638 A namespace in the XPath namespace list is missing.

-15639 XPath function has two mandatory parameters.

-15640 Internal XPath error.

-15641 Invalid XPath namespace.

-15642 Unable to load schema.

-15643 Schemas are malformed or inconsistent.

-15644 URI is malformed.

-15645 Protocol layer needs a new try to complete operation.

-15646 Charset conversion error.

-15699 Internal error, should not happen.

See also: The Genero Web Services XML Extension Library

OM to XML Migration

This topic assists you in migrating code you have written using methods from the OM class to code that uses methods from the XML Extension Library class.

Why Migrate?

- You need to be able to utilize a feature (such as a StyleSheet) that requires the code to use XML class methods.

OM - XML Mapping

OM class method	XML class method(s)
om.DomDocument.create	xml.DomDocument.createDocument
om.DomDocument.createFromXmlFile	xml.DomDocument.load
om.DomDocument.createFromString	xml.DomDocument.loadFromString
om.DomDocument.copy	xml.DomNode.clone
om.DomDocument.createChars	xml.DomDocument.createTextNode
om.DomDocument.createEntity	xml.DomDocument.createEntityReference
om.DomDocument.createElement	xml.DomDocument.createElement
om.DomDocument.getDocumentElement	xml.DomDocument.getFirstDocumentNode
om.DomDocument.getElementById	xml.DomDocument.getElementById + xml.DomNode.setIdAttribute or xml.DomNode.setIdAttributeNS
om.DomDocument.removeElement	xml.DomDocument.removeDocumentNode
om.DomNode.appendChild	xml.DomDocument.createNode + xml.DomNode.appendChild
om.DomNode.createChild	xml.DomDocument.createNode + xml.DomNode.appendChild
om.DomNode.insertBefore	xml.DomNode.insertBeforeChild
om.DomNode.removeChild	xml.DomNode.removeChild
om.DomNode.replaceChild	xml.DomNode.replaceChild
om.DomNode.loadXml	xml.DomDocument.loadFromString
om.DomNode.parse	xml.DomDocument.createNode + add it to the DomDocument
om.DomNode.toString	xml.DomNode.toString
om.DomNode.writeXml	xml.DomDocument.save
om.DomNode.write	xml.DomNode.toString
om.DomNode.getId	xml.DomNode.getAttributeName("Id")

om.DomNode.getTagName	xml.DomNode.getLocalName
om.DomNode.setAttribute	xml.DomNode.setAttribute
om.DomNode.getAttribute	xml.DomNode.getAttribute
om.DomNode.getAttributeInteger	xml.DomNode.getAttribute + condition for the default value and the cast
om.DomNode.getAttributeString	xml.DomNode.getAttribute + condition for the default value and the cast
om.DomNode.getAttributeName	xml.DomNode.getAttributeNodeItem + xml.DomNode.getLocalName
om.DomNode.getAttributesCount	xml.DomNode.getAttributesCount
om.DomNode.getAttributeValue	xml.DomNode.getAttributeNodeItem + xml.DomNode.getNodeValue
om.DomNode.removeAttribute	xml.DomNode.removeAttribute
om.DomNode.getChildCount	xml.DomNode.getChildrenCount
om.DomNode.getChildByIndex	xml.DomNode.getChildNodeItem
om.DomNode.getFirstChild	xml.DomNode.getFirstChild
om.DomNode.getLastChild	xml.DomNode.getLastChild
om.DomNode.getNext	xml.DomNode.getNextSibling
om.DomNode.getParent	xml.DomNode.getParentNode
om.DomNode.getPrevious	xml.DomNode.getPreviousSibling
om.DomNode.selectByTagName	xml.DomNode.getElementsByTagName
om.DomNode.selectByPath	xml.DomNode.selectByXPath
om.DomNodeList.item	xml.DomNodeList.getItem
om.DomNodeList.getLength	xml.DomNodeList.getCount

See also: The Genero Web Services XML Extension Library

For more information on Genero built-in classes (such as the OM class), refer to the *Genero Business Development Language Manual*.