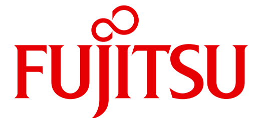
User Guide - English



WebTransactions V7.5

Access to Dynamic Web Contents

Edition August 2010

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to:
manuals@ts.fujitsu.com

Certified documentation according to DIN EN ISO 9001:2008

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2008.

cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Copyright and Trademarks

Copyright © Fujitsu Technology Solutions GmbH 2010.

All rights reserved.
Delivery subject to availability; right of technical modifications reserved.

All hardware and software names used are trademarks of their respective manufacturers.

Contents

1 Preface . . . 7

1.1 Product characteristics . . . 7

1.2 Architecture of WebTransactions for HTTP . . . 9

1.3 WebTransactions documentation . . . 10

1.4 Structure and target group of this manual . . . 12

1.5 New features . . . 13

1.6 Notational conventions . . . 13

2 Creating a base directory . . . 15

3 Controlling communication . . . 17

3.1 System object attributes . . . 17
3.1.1 Overview . . . 18
3.1.2 Interaction between system object attributes and methods . . . 21

3.2 Host objects . . . 24
3.2.1 Sending a message with the host object sendData . . . 24
3.2.1.1 Message without the sendData object . . . 25
3.2.1.2 Single-part messages . . . 26
3.2.1.3 Multipart messages . . . 29
3.2.2 The Header attribute . . . 32
3.2.3 Receiving a message in the receiveData host object . . . 33

3.3 Processing HTTP raw data . . . 34
3.3.1 WTScript filters . . . 34
3.3.2 User exits . . . 34
3.3.3 Built-in filters . . . 35

3.4 Start templates for HTTP . . . 36
3.4.1 HTTP-specific start template of the start template set (wtstartHTTP.htm) . . . 36
3.4.2 Simple start template (StartTemplateHTTP.htm) . . . 40

3.5 Creating a new HTTP communication object (wtcHTTP) . . . 45

4 Connecting web services via SOAP . . . 47

4.1 Concept of SOAP integration in WebTransactions . . . 47
4.1.1 SOAP (Simple Object Access Protocol) . . . 47
4.1.2 Describing SOAP services with WSDL . . . 48
4.1.3 UDDI (Universal Description, Discovery and Integration Project) . . . 50
4.1.4 SOAP support in WebTransactions . . . 50

4.2 WT_SOAP - client-side class . . . 52
4.2.1 Structure of a WT_SOAP object . . . 53
4.2.1.1 Representation in the WebLab object tree . . . 54
4.2.1.2 Example: WSDL document . . . 58
4.2.2 Constructor for the WT_SOAP class . . . 60
4.2.3 Proxy methods . . . 62
4.2.4 WT_SOAP class object methods . . . 65
4.2.4.1 initFromWSDLUri method . . . 65
4.2.4.2 setRunMode method . . . 66
4.2.4.3 executeRequest method . . . 67
4.2.4.4 executeGetRequest method . . . 68
4.2.4.5 analyseResponse method . . . 68
4.2.4.6 setSOAPVersion method . . . 69
4.2.4.7 addHeader method . . . 70
4.2.4.8 removeAllHeaders method . . . 70
4.2.4.9 getHeaderObjects method . . . 70
4.2.4.10 getHeaderObjectTree method . . . 72
4.2.4.11 createProxysWithPrefix method . . . 73
4.2.5 Methods for configuring access to the WT_SOAP_COM_FUNCTIONS subclass . . . 75
4.2.5.1 setAuthorization method . . . 75
4.2.5.2 setProxy method . . . 76
4.2.5.3 setProxyAuthorization method . . . 76
4.2.5.4 setTimeout method . . . 77
4.2.6 Exceptions . . . 77
4.2.7 WT_SOAP attributes . . . 79
4.2.8 Data types for the SOAP request in SOAP body . . . 80
4.2.9 Example: Checking the spelling of a text . . . 83

4.3 WT_SOAP_HEADER - class for support of SOAP headers 84
4.3.1 Constructor of the WT_SOAP_HEADER class 84

5 Examples 89

5.1 Using an existing CGI script 89

5.2 Using information from the Web 91

5.3 Communicating via HTTP and processing with WT_Filter 93
5.3.1 Basic concept of the WT_RPC class 93
5.3.2 Implementation of the WT_RPC class 95

6 Appendix 101

6.1 HTTP error messages 101

6.2 WSDL Schema 103

Glossary 109

Abbreviations 127

Related publications 129

Index 131

1 Preface

Over the past years, more and more IT users have found themselves working in heterogeneous system and application environments, with mainframes standing next to Unix systems and Windows systems and PCs operating alongside terminals. Different hardware, operating systems, networks, databases and applications are operated in parallel. Highly complex, powerful applications are found on mainframe systems, as well as on Unix servers and Windows servers. Most of these have been developed with considerable investment and generally represent central business processes which cannot be replaced by new software without a certain amount of thought.

The ability to integrate existing heterogeneous applications in a uniform, transparent IT concept is a key requirement for modern information technology. Flexibility, investment protection, and openness to new technologies are thus of crucial importance.

1.1 Product characteristics

With WebTransactions, Fujitsu Technology Solutions offers a best-of-breed web integration server which will make a wide range of business applications ready for use with browsers and portals in the shortest possible time. WebTransactions enables rapid, cost-effective access via standard PCs and mobile devices such as tablet PCs, PDAs (Personal Digital Assistant) and mobile phones.

WebTransactions covers all the factors typically involved in web integration projects. These factors range from the automatic preparation of legacy interfaces, the graphic preparation and matching of workflows and right through to the comprehensive frontend integration of multiple applications. WebTransactions provides a highly scaleable runtime environment and an easy-to-use graphic development environment.

On the first integration level, you can use WebTransactions to integrate and link the following applications and content directly to the Web so that they can be easily accessed by users in the internet and intranet:

- Dialog applications in BS2000/OSD
- MVS or z/OS applications
- System-wide transaction applications based on openUTM
- Dynamic web content

Users access the host application in the internet or intranet using a web browser of their choice.

Thanks to the use of state-of-the-art technology, WebTransactions provides a second integration level which allows you to replace or extend the typically alphanumeric user interfaces of the existing host application with an attractive graphical user interface and also permits functional extensions to the host application without the need for any intervention on the host (dialog reengineering).

On a third integration level, you can use the uniform browser interface to link different host applications together. For instance, you can link any number of previously heterogeneous host applications (e.g. MVS or OSD applications) with each other or combine them with dynamic Web contents. The source that originally provided the data is now invisible to the user.

In addition, you can extend the performance range and functionality of the WebTransactions application through dedicated clients. For this purpose, WebTransactions offers an open protocol and special interfaces (APIs).

Host applications and dynamic Web content can be accessed not only via WebTransactions but also by “conventional” terminals or clients. This allows for the step-by-step connection of a host application to the Web, while taking account of the wishes and requirements of different user groups.

1.2 Architecture of WebTransactions for HTTP

The figure below illustrates the architecture of WebTransactions for HTTP:

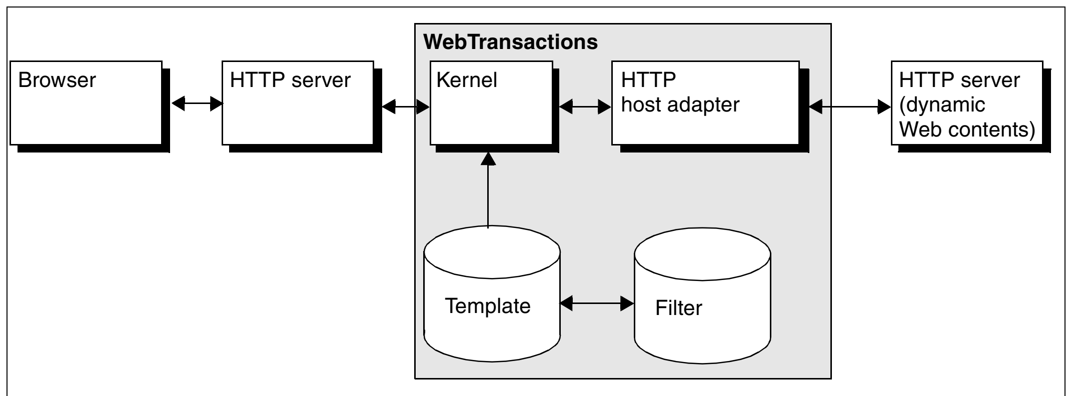


Figure 1: Architecture of WebTransactions for HTTP

HTTP host adapter

WebTransactions uses the HTTP host adapter for managing communication between the WebTransactions kernel and any HTTP server and for accessing dynamic Web material.

The HTTP host adapter supports data transport based on the HTTP protocol. The host objects comply with the simple object model of this protocol with a message header and body. However, the adapter does not support any further interpretation of the message content, which may be encoded in HTML or XML for example.

Filter

Individual filters must be used for interpreting incoming messages and converting WebTransactions information into HTTP messages. Some filters are supplied ready-made (e.g. HTTP messages for `WT_REMOTE`), but it is also possible to implement your own filters in the form of WTML script functions or user exits (see examples in chapter “Examples” on page 89).

1.3 WebTransactions documentation

The WebTransactions documentation consists of the following documents:

- An introductory manual which applies to all supply units:

  Concepts and Functions

  This manual describes the key concepts behind WebTransactions:

  – The various possible uses of WebTransactions.

  – The concept behind WebTransactions and the meanings of the objects in WebTransactions, their main characteristics and methods, their interaction and life cycle.

  – The dynamic runtime of a WebTransactions application.

  – The administration of WebTransactions.

  – The WebLab development environment.

- A Reference Manual which also applies to all supply units and which describes the WebTransactions template language WTML. This manual describes the following:

  Template Language

  After an overview of WTML, information is provided about:

  – The lexical components used in WTML.

  – The class-independent global functions, e.g. `escape()` or `eval()`.

  – The integrated classes and methods, e.g. `array` or `Boolean` classes.

  – The WTML tags which contain functions specific to WebTransactions.

  – The WTScript statements that you can use in the WTScript areas.

  – The class templates which you can use to automatically evaluate objects of the same type.

  – The master templates used by WebTransactions as templates to ensure a uniform layout.

  – A description of Java integration, showing how you can instantiate your own Java classes in WebTransactions and a description of user exits, which you can use to integrate your own C/C++ functions.

  – The ready-to-use user exits shipped together with WebTransactions.

  – The XML conversion for the portable representation of data used for communication with external applications via XML messages and the conversion of WTScript data structures into XML documents.

- A User Guide for each type of host adapter with special information about the type of the partner application:

  Connection to openUTM applications via UPIC

  Connection to OSD applications

  Connection to MVS applications

  All the host adapter guides contain a comprehensive example session. The manuals describe:

  – The installation of WebTransactions with each type of host adapter.

  – The setup and starting of a WebTransactions application.

  – The conversion templates for the dynamic conversion of formats on the web browser interface.

  – The editing of templates.

  – The control of communications between WebTransactions and the host applications via various system object attributes.

  – The handling of asynchronous messages and the print functions of WebTransactions.

- A User Guide valid for all the supply units which describes the open protocol, and the interfaces for the client development for WebTransactions:

  Client APIs for WebTransactions

  This manual describes:

  – The concept of the client-server interface in WebTransactions.

  – The `WT_RPC` class and the `WT_REMOTE` interface. An object of the `WT_RPC` class represents a connection to a remote WebTransactions application which is run on the server side via the `WT_REMOTE` interface.

  – The Java package `com.siemens.webta` for communication with WebTransactions supplied with the product.

- A User Guide valid for all the supply units which describes the web frontend of WebTransactions that provides access to the general web services:

  Web-Frontend for Web Services

  This manual describes:

  – The concept of web frontend for object-oriented backend systems.

  – The generation of templates for the connection of general web services to WebTransactions.

  – The testing and further development of the web frontend for general web services.

1.4 Structure and target group of this manual

This manual is intended for all users who wish to access dynamic Web material via WebTransactions.

The individual chapters describe the steps involved in this procedure, which is also explained in the final chapter with the help of practical examples.

This manual supplements the WebTransactions introductory manual “Concepts and Functions” and the WebTransactions reference manual “Template Language” with the information required for HTTP connection.

Scope of this description

WebTransactions for HTTP runs on the Windows, Solaris, Linux and BS2000/OSD system platforms. This documentation applies to all the platforms. If an item of information applies only to a specific platform then this will be clearly stated in the text.

1.5 New features

This section lists only the HTTP-specific innovations. For a general overview of the new features, refer to the WebTransactions manual “Concepts and Functions.

Type of new feature	**Description**
New system object attribute `COMMUNICATION_FILE_NAME`	page 18
New system object attribute `COMMUNICATION_FILE_TYPE`	page 18
New system object attribute `METHOD`	page 18

1.6 Notational conventions

The following notational conventions are used in this documentation:

Name	**Description**
`typewriter font`	Fixed components which are input or output in precisely this form, such as keywords, URLs, file names
italic font	Variable components which you must replace with real specifications
bold font	Items shown exactly as displayed on your screen or on the graphical user interface; also used for menu items
[ ]	Optional specifications; do not enter the square brackets themselves
{*alternative1* | *alternative2* }	Alternative specifications. You must select one of the expressions inside the curly brackets. The individual expressions are separated from one another by a vertical bar. Do not enter the curly brackets and vertical bars themselves.
...	Optional repetition or multiple repetition of the preceding components
i	Important notes and further information
►	Prompt telling you to do something.
⇨	Refers to detailed information

2 Creating a base directory

Following the installation of one of the WebTransactions supply units on the WebTransactions server, the files and programs required to dynamically access Web material are automatically installed. After installing WebLab on your own personal Windows system, you can use it to create one or more base directories. A base directory contains all the files used to configure WebTransactions for a specific WebTransactions application.

If you deinstall WebTransactions or install a new product version, the individual configurations are retained.

WebTransactions for HTTP can be used for two purposes:

- It can be used for independent access to a HTTP server, in which case you can create one or more base directories exclusively for WebTransactions for HTTP.
- It can also be used in conjunction with one of the supply units for converting host applications (such as WebTransactions for OSD), e.g. in order to implement a Web interface for an existing host application which allows for access to a Web search engine.In this case, you create a base directory for HTTP and for the supply unit you have purchased for conversion. For further information, see the chapter “Creating a base directory” in the corresponding product manual.

The following sections describe the first scenario. They also clarify the files that must be created for HTTP access in both cases, and the system object attributes or host objects that are available specifically for HTTP access.

Creating a base directory with WebLab

Before you can create a base directory for a WebTransactions application, the WebTransactions administrator must have created a user ID for you and then subsequently released one or more pools for this user ID in which you can create a base directory.

Before you create a base directory, it is recommended that you first create a project to store most important data required by WebLab when working with the WebTransactions application. When creating a project, you are automatically offered the opportunity to create a base directory.

To do this, proceed as follows:

- ► Call WebLab, e.g. via **Start/Programs/WebTransactions 7.5/WebLab**

- ► There are two possibilities for starting to create a base directory:

  - ► Select the **Project/New...** command and when asked whether you want to create a base directory, answer **Yes**.

  or

  - ► Choose the **Generate/Basedir...** command and specify that a new project is to be created when the relevant query appears.

  In both instances, the **Connect** dialog box is opened.

- ► Edit the connection parameters in the **Connect** dialog box using the **Change** button and click **OK** to confirm. The parameters set in the **Options** menu, **Preferences** command, **Server** tab, are displayed.
- ► In the following dialog box, type your user ID and password and click **OK** to confirm.
- ► Make the following settings in the **Create Basedir** dialog box:
  - – From the list of proposed pools, select the pool in which you want to create the base directory.
  - – Type a name for the new directory.
  - – Check the **HTTP** box in the **Host Adapter** area.
  - – Click **OK**.

WebLab now sets up the base directory with all the associated files that are required to run the WebTransactions application. The structure and contents of the base directory are described in the WebTransactions manual “Concepts and Functions”.

Converting a base directory to a new version

- ► Select **Generate/Update Base Directory**. This opens the **Update Base Directory** dialog box.
- ► If you only want to change the links from the base directory to the new installation directory, select the **Update all links** option. Select this option when you have updated the files that are supplied or generated by WebTransactions.
- ► If all files which are copied or generated on creation of the base directory need to be recreated, select the **Overwrite all files** option.

3 Controlling communication

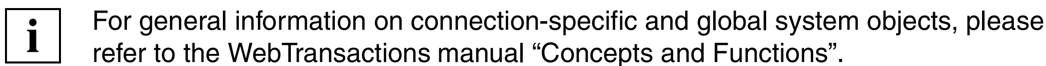
This chapter describes how to use WebTransactions for HTTP to access the HTTP server and its resources. Practical examples of the concepts discussed here can be found in chapter “Examples” on page 89.

3.1 System object attributes

To control communication between WebTransactions and an HTTP server, you use some of the system object attributes.

This section concentrates only on those attributes which are provided specifically for HTTP connections or which are of special significance in that context. The system object attributes that apply to all supply units of WebTransactions are described in the WebTransactions manual “Concepts and Functions”.

If a subobject `WT_SYSTEM` (private system object) exists under the communication object used, the attributes described in this section must be defined there. Otherwise, they must be declared as attributes of the global system object `WT_SYSTEM`.



For general information on connection-specific and global system objects, please refer to the WebTransactions manual “Concepts and Functions”.

The attributes can be set in the first template (start template) when starting WebTransactions and can be retained for the entire session or actively modified during the session (see the WebTransactions manual “Concepts and Functions”).

3.1.1 Overview

The table below provides an overview of the attributes and their effects. The system object attributes can be subdivided into the categories specified in the right-hand column of table below:

t (**t**emporary)
Attributes used during communication and which can be modified at any time in the templates.

c (**c**ommunication module)
Attributes set automatically by the host adapter.

Attribute name	Meaning	Description/category	
COMMUNICATION_FILE_NAME	Name of the destination file	File in which data received during a receive operation is to be stored. The body of the message is stored. The file must be located below the base directory. It is not permitted to store the file in the directory wwwdocs. You can use this attribute to transfer graphical images, for example (see also COMMUNICATION_FILE_TYPE below).	t
COMMUNICATION_FILE_TYPE	Filter for storing	Filter for storing a message. The attribute contains the type of the message. Only when the type of the received message (Content-Type field in the HTTP header) corresponds to the value specified here is the body of the message stored in the file you specified with COMMUNICATION_FILE_NAME (see above).	t
HTTP_RETURN_CODE	Error number	Return code of a HTTP request. This attribute is set by the receive method and specifies the return code of the HTTP request. The value 200 means OK. All other values indicate an error. HTTP_RETURN_CODE is empty if no answer to the request has been received from the HTTP server. An overview of possible error codes and their meaning can be found in the section “HTTP error messages” on page 101.	c
METHOD	Method for HTTP request	The method specified here is used for the HTTP request initiated by send()/receive(). Alternatively, GET or POST will be used, depending on whether the object Body exists.	t

Attribute name	Meaning	Description/category	
PASSWORD	Password	Password for the HTTP connection.. Together with the USER attribute (see below), this attribute allows for authentication on the remote HTTP server. This attribute has priority over the corresponding value in URL.[1]	t
PROXY	HTTP proxy	Name or IP address of the system to be used as the HTTP proxy for the HTTP connection (see also PROXY_PORT below).	t
PROXY_PASSWORD	Password for the proxy	Password for HTTP connection via a proxy system.	
PROXY_PORT	Port for HTTP proxy	Port of the HTTP proxy to be used (see also PROXY). If no port is specified, the default value is 80.	t
PROXY_USER	User name for the proxy	User name for HTTP connection via a proxy system; in conjunction with the attribute PROXY_PASSWORD (see above), this attribute is used for authentication at the remote HTTP server via a proxy system.	t
SSL_CERT_FILE	SSL certificate	Name of the file containing one of the client-side certifcates to be used. If the file name is not entered as absolute, it will be relative to the base directory.	t
SSL_KEY_FILE	Key to the certificate	Name of the file containing the private key to the certificate (seeSSL_CERT_FILE). If the file name is not entered as absolute, it will be relative to the base directory. If this attribute is not specified, the value of SSL_CERT_FILE will be taken over. It is assumed that the certificate and the key are contained in the same file.	t
SSL_PASSPHRASE	Passphrase for the private key	Passphrase to be used with the private key.	t
SSL_PROTOCOL	SSL protocol selected	Entry indicating which version of the SSL protocol and the TLS protocol is to be activated. OpenSSL supports SSL protocol versions 2 and 3, and TLS protocol version 1. Possible values: SSLv2 SSLv3 TLSv1 All Default value: All You can enter several protocols but these should be separated by blank spaces.	t

Attribute name	Meaning	Description/category	
TIMEOUT_HTTP	Timeout for HTTP requests	Time in seconds, after a `send` method call, that the system waits for an answer from the HTTP server. If this attribute is not set, the default value of 60 seconds will be used. If the `TIMEOUT_HTTP` is greater than the global system object attribute `TIMEOUT_APPLICATION`, a value derived from `TIMEOUT_APPLICATION` will be used: – If `TIMEOUT_APPLICATION` > 10: `TIMEOUT_HTTP corresponds to TIMEOUT_APPLICATION` -5 – If `TIMEOUT_APPLICATION` > 1: `TIMEOUT_HTTP corresponds to TIMEOUT_APPLICATION` -1 – If `TIMEOUT_APPLICATION` =1 `TIMEOUT_HTTP corresponds to TIMEOUT_APPLICATION`.	t
`URL`	Target URL	Target URL of the HTTP resource addressed. The syntax for this attribute is as follows: [http[s]://][*user*[:*password*]@] *machine*[:*port*]/*file*[/*pathinfo*][?*query*][2] If the `https` protocol is specified for this attribute, the default value for the *port* is `443`; in all other cases the default setting is `80`. The optional values for *user* and *password* will be overwritten by the values of the attributes `USER` and `PASSWORD`.	t
USER	User name	User name for the HTTP connection. Together with the attribute `PASSWORD` (see above), this attribute allows for authentication on the remote HTTP server. This attribute takes priority over the corresponding `URL` value.[1]	t

[1] The user name, password, and port can also be specified to the HTTP server using the URL.
The format is:
[http:][//][*user*[:*password*]@]*machine*[:*port*]/...

[2] Short description of the URL syntax:
user - user
password - password
machine - computer name or IP address of the HTTP server
port - port number of the HTTP server
file - file name on the HTTP server
pathinfo, query - information only relevant for the CGI program, specifying the environment variables for the CGI.

3.1.2 Interaction between system object attributes and methods

This section describes which HTTP-specific system object attributes play a role in which method calls.

open - activate the HTTP host adapter

The `open` method call initializes a communication method object for HTTP connections. Since the HTTP protocol does not recognize the term “session”, `open` does not open a session but simply activates the HTTP host adapter for the communication object used.

System object attributes need not be taken into consideration at this point. They are significant only for the HTTP requests issued with the `send` and `receive` methods described below. All you need to specify here is the protocol used (`HTTP`).

Example

```
http_host = new WT_Communication('myHTTP');
http_host.open('HTTP');
```

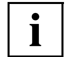
This example creates a new communication object with the name `myHTTP` under `WT_HOST`, as well as a second reference `http_host` to this communication object. It then activates the HTTP host adapter.

send - send the HTTP request

The `send` method establishes a HTTP connection using the following system object attributes in the process:

System object attribute	Use
`URL`	`send` establishes an HTTP connection to the specified HTTP resource.
`PROXY` `PROXY_PORT`	These system object attributes are interpreted as an HTTP proxy.
`USER` `PASSWORD`	If these attributes are set, they are used for user authentication.
`PROXY_USER` `PROXY_PASSWORD`	If these attributes are set, they are sent for user authentication at the proxy system.
`TIMEOUT_HTTP`	This attribute sets the timeout for the response from the addressed HTTP server. Default value: 60 seconds.

Table 1: System object attributes used in send



If the `send` method is executed several times in succession, the connection from the previous call is first closed. This may cause the result from the previous call to be lost.

Example

```
// Define the URL and the HTTP proxy:
http_host.WT_SYSTEM.URL = 'www.mycompany.de/webtransactions';
http_host.WT_SYSTEM.PROXY = 'proxy.mycompany.de';
http_host.WT_SYSTEM.PROXY_PORT = '80';
// Delete the host object sendData:
delete http_host.sendData;
// Start the HTTP request:
http_host.send();
```

This example executes the HTTP `GET` method for the WebTransactions home page. The proxy server `proxy.mycompany.de` (port `80`) is used, and no user is predefined. The timeout for the server response is 60 seconds.

receive - retrieve response to the HTTP request

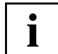
The `receive` method call returns the result of an HTTP request in the `receiveData` host object (see section “Host objects” on page 24) and uses the following system object attributes in the process:

System object attribute	Use
`HTTP_RETURN_CODE`	This attribute is supplied with the return code. If a return code value other than 200 is output when retrieving response data, the system object `ERROR` attribute is set in the global system object. An overview of possible error codes and their meaning can be found in the section “HTTP error messages” on page 101. If no response data is received, `HTTP_RETURN_CODE` is empty and the system object attribute `ERROR` is set at the global system object.
`TIMEOUT_HTTP`	As soon as the timer runs out, `receive` returns immediately without a response, setting the `ERROR` attribute in the global system object and `HTTP_RETURN_CODE` in the system object accordingly.

Table 2: System object attributes used in receive

As soon as the response data is received, the connection to the HTTP server is closed. The response message is analyzed and its contents are stored in the `receiveData` host object without further processing. In the case of a multisection message, the contents of each section are stored in an array of objects (`receiveData.0`, `receiveData.1`, ...).

WebTransactions does not interpret the contents of received data. This can only be carried out using filters that have been specially adapted to suit the type of contents and their structure (e.g. `text/html`, `text/xml`, etc.).



If the `receive` method is executed without a preceding `send`, the `send` actions will be executed implicitly, automatically taking into account the system object attributes described in the section on `send` and the `sendData` object.

The combination of `send` and `receive` is preferable to the exclusive use of `receive`, particularly if you require parallel loading, i.e. you can execute a `send` call one dialog step in advance and after the dialog step use `receive` to query whether any data has arrived. This allows you to minimize wait times.

Example

```
// Define the URL:
http_host.WT_SYSTEM.URL = 'www.mycompany.de/webtransactions';
// Delete the host object sendData:
delete http_host.sendData;
// Start the HTTP request:
http_host.send();
// Retrieve the response data:
http_host.receive();
```

This example executes the HTTP `GET` method for the WebTransactions home page. No proxy server is used, and no user is predefined. The timeout for the server response is set to 60 seconds. The specified page is received and can be processed further using `http_host.receiveData`.

close - deactivate the HTTP module

The `close` method call deactivates the HTTP host adapter. This causes the HTTP module to release its internal memory. It should be called therefore if no further HTTP requests are required in the WebTransactions application.

3.2 Host objects

WebTransactions uses the host objects `sendData` and `receiveData` to perform data transfer with an HTTP server:

– You can create `sendData` yourself if you want to send data to the HTTP server

– `receiveData` contains the server response data.

Both host objects may possess the following attributes:

Attribute	Meaning
`ContentType`	Attribute of `string` type: type of HTTP message
`Header`	Object of `Array` class: information for the HTTP server For a description of the `Header` attribute, refer to section “The Header attribute” on page 32
`Body`	Attribute of `string` type: content of HTTP message

Table 3: Host object attributes

Whenever a message is sent, a header containing predefined fields is sent together with these attributes. The number and meaning of these fields depends on the associated HTTP request. See the following section for more details.

3.2.1 Sending a message with the host object sendData

Depending on whether or not you want to send data to the HTTP server and, if you do, what the nature of this data is, the host object `sendData` and consequently the HTTP request may be constructed. An HTTP request consists of the `GET` or `POST` method and argument, followed by a protocol version, a header and possibly a message. A request may therefore have the following structure:

`POST` *request* `HTTP/1.0 Header senddata`

The *request* itself is constructed as follows:

`/file[/pathinfo][?query]`

You can send the following message types to the HTTP server via the host adapter

– without the `sendData` host object

– single-part message

– multi-part messages

These different message types are described below.

3.2.1.1 Message without the sendData object

If you have not created a `sendData` host object for the first connection request then the HTTP `GET` method is executed:

`GET` *request* `HTTP/1.0 Header`

The information sent to the HTTP server consists of a predefined header with the following fields:

`Host`	Contains the host name from the URL (and possibly the port number if this is not 80).
`User-Agent`	Contains the text “WebTransactions HTTP/7.5” (version dependent)
`Authorization`	Contains a text generated from the `USER` and `PASSWORD` attributes by means of `uuencode` (see also the system object attributes).
`Proxy-Authorization`	Contains a text generated from the `PROXY_USER` and `PROXY_PASSWORD` attributes by means of `uuencode` (see also the system object attributes).

Example

```
host = new WT_Communication( 'http' );
host_system = host.WT_SYSTEM;
host.open('HTTP');
host_system.URL = "//localhost/webtav75/wtadm_admin.htm";
host.send();
host.receive();
```

These specifications result in the following information being sent to the HTTP server

```
Hexadecimal:                                      Ascii:
00000: 47 45 54 20 2f 77 65 62 74 61 76 37 35 2f 77 74 !GET /webtav75/wt!.......@./......!
00016: 61 64 6d 5f 61 64 6d 69 6e 2e 68 74 6d 20 48 54 !adm_admin.htm HT!/]_^/]_.>..._...!
00032: 54 50 2f 31 2e 30 0d 0a 48 6f 73 74 3a 20 6c 6f !TP/1.0..Host: lo!.&.......?....%?!
00048: 63 61 6c 68 6f 73 74 0d 0a 55 73 65 72 2d 41 67 !calhost..User-Ag![/%.?..........~!
00064: 65 6e 74 3a 20 57 65 62 54 72 61 6e 73 61 63 74 !ent: WebTransact!.>.....@../>./[.!
00080: 69 6f 6e 73 20 48 54 54 50 2f 37 2e 35 41 30 30 !ions HTTP/7.5A00!.?>.....&.......!
00096: 0d 0a 0d 0a                                     !....            !....            !
```

3.2.1.2 Single-part messages

If you want to send a single-part message without its own header to the HTTP server then `sendData` must be an object of `Object` class and possess the `Body` attribute:

```
sendData
      Body
```

The `Body` attribute contains the actual message to the HTTP server. WebTransactions uses this information to formulate the `POST` method before sending the data to the HTTP server. The information sent to the HTTP server consists of a predefined header with the following fields together with `sendData.Body`:

`Host`	Contains the host name from the URL (and possibly the port number if this is not 80).
`User-Agent`	Contains the text "WebTransactions HTTP/7.5" (version dependent)
`Authorization`	Contains a text generated from the `USER` and `PASSWORD` attributes by means of `uuencode` (see also the system object attributes).
`Proxy-Authorization`	Contains a text generated from the `PROXY_USER` and `PROXY_PASSWORD` attributes by means of `uuencode` (see also the system object attributes).
`Content-Length`	Specifies the length of `sendData.body`

The content of `Body` must be url-encoded. This means that some characters are replaced by `'%'` followed by their hexadecimal value (e.g.: `':'` by `%3A`, `'\'` by `%5C`).

Example

```
host.sendData = new Object();
host.sendData.ContentType = 'application/x-www-form-urlencoded';
host.sendData.Body = 'question=sense+of+universe&answer=42';
host.WT_SYSTEM.URL = 'www.deepThought.mt/cgi-bin/verify.exe';
host.WT_SYSTEM.USER = 'user27';
host.WT_SYSTEM.PASSWORD = 'pass';
host.send();
...
```

The following values are sent via the predefined header fields in the HHTP request:

`Content-Type`	application/x-www-form-urlencoded
`Host`	www.deepThought.mt
`User-Agent`	WebTransactions HTTP/7.5
`Authorization`	Basic dXNlcjI3OnBhc3M=

`Content-Length`	36

If you want to supply your own header for the message then you also need the attributes `ContentType` and/or `Header` for the `sendData` host object:

- If you simply want to specify the message type then you use the `ContentType` attribute as before.
- You only use the `Header` attribute if you want to send additional information to the HTTP server. `Header` must be an object of `Array` type. For more information, refer to section "The Header attribute" on page 32.

```
sendData
      Body
      [ContentType]
      [Header]
          0
           Name
           Value
          1
           Name
           Value
```

The `Content-Type` field is added to the predefined header:

`Content-Type`	Contains the value of the `sendData.ContentType` attribute or the corresponding element from a user-defined header. If this attribute does not exist then no such header field is sent.

i You should note that the sequence and value of the predefined header fields may be affected by a user-defined header, see section "The Header attribute" on page 32.

Example

```
host.sendData = new Object();
host.sendData.ContentType = 'application/x-www-form-urlencoded';
host.sendData.Header = new Array();
host.sendData.Header[0] = new Object();
host.sendData.Header[0].Name = 'Pragma';
host.sendData.Header[0].Value = 'nocache';
host.sendData.Header[1] = {Name:'Authorization'};
host.sendData.Header[2] = {Name:'ConTenT-Type', Value:'text'};
host.sendData.Header[3] = {Name:'Content-Length'};
host.sendData.Body = 'question=sense+of+universe&answer=42';
host.WT_SYSTEM.URL = 'www.deepThought.mt/cgi-bin/verify.exe';
host.WT_SYSTEM.USER = 'user27';
host.WT_SYSTEM.PASSWORD = 'pass';
host.send();
...
```

The `Header` attribute affects the sequence, notation and contents of the header fields. The following fields are sent with the HTTP request:

`Host`	www.deepThought.mt	1
`User-Agent`	WebTransactions HTTP/7.5	1
`Pragma`	nocache	2
`Authorization`	Basic dXNlcjI3OnBhc3M=	3
`ConTenT-Type`	text	4
`Content-Length`	36	3

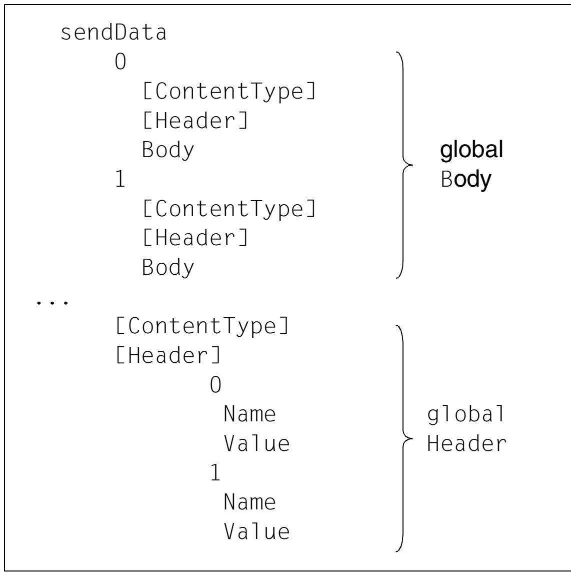
Comments

1. Predefined field at start of list since the corresponding element is not present in the array header.
2. Additional field from the array header.
3. Position controlled by the element in the array header; content is calculated internally (predefined field) because the attribute value does not exist.
4. The predefined field is ignored and the position and value of the field are determined from the array header (including the meaningless notation `ConTenT-Type`).

3.2.1.3 Multipart messages

If you want to send a multipart message then you must create `sendData`as an object of the `Array` class. Individual message segments are elements of the `sendData` array and are numbered sequentially `sendData[0]`, `senddata[1]`, and so on.

Each message segment (i.e. each element) must contain an object `Body` and can also contain a `ContentType` object and a `Header` header.

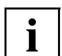


The global body contains the message itself which is further divided into multiple message segments for the HTTP server. A message segment thus corresponds to a single-part message with the attributes `Body`, `ContentType` and `Header`. The global body has the following predefined header fields:

`Host`	Contains the host name from the URL (and possibly the port number if this is not 80).
`User-Agent`	Contains the text “WebTransactions HTTP/7.5” (version dependent)
`Authorization`	Contains a text generated from the `USER` and `PASSWORD` attributes by means of `uuencode` (see also the system object attributes).
`Proxy-Authorization`	Contains a text generated from the `PROXY_USER` and `PROXY_PASSWORD` attributes by means of `uuencode` (see also the system object attributes).

`Content-Type` Contains the value of the `sendData.ContentType` attribute or the corresponding element from the user-defined header. If this attribute does not exist then the following value is sent:
`multipart/mixed; boundary=<<'<<"<<42>>">>'>>`

`Content-Length` Specifies the length of `sendData.body`



You should note that the sequence and value of the predefined header fields may be affected by a user-defined header.

If the global `ContentType` attribute is used for the type of global message then it must always start with the text `multipart`. No distinction is made between uppercase and lowercase notation.

Example

In this example, a global header is sent together with an individual header for each of the message segments.

```
host_system.URL = "//localhost/Scripts/Cgitest.exe";
host.sendData = new Array();
host.sendData.Header = new Array();
host.sendData.Header[0] = { Name:'global-header-field', Value:'global
content' };
host.sendData[0] = new Object();
host.sendData[0].ContentType = "application/x-www-form-urlencoded";
host.sendData[0].Header = new Array();
host.sendData[0].Header[0] = { Name:'part1-header-field', Value:'header
part 1' };
host.sendData[0].Body =
"WT_SYSTEM_BASEDIR=%2Fhome1%2Fpuls%2Fhttpd%2Fpulswww&command=Refresh";
host.sendData[1] = new Object();
host.sendData[1].ContentType = "type1";
host.sendData[1].Header = new Array();
host.sendData[1].Header[0] = { Name:'part2-header-field', Value:'any
other' };
host.sendData[1].Body = "WT_SYSTEM_BASEDIR=val1&command=Refresh";
host.send();
```

These specifications result in the following information being sent to the HTTP server:

```
Hexadecimal:                                               Ascii:
00000: 50 4f 53 54 20 2f 53 63 72 69 70 74 73 2f 43 67 !POST /Scripts/Cg!
00016: 69 74 65 73 74 2e 65 78 65 20 48 54 54 50 2f 31 !itest.exe HTTP/1!
00032: 2e 30 0d 0a 43 6f 6e 74 65 6e 74 2d 54 79 70 65 !.0..Content-Type!
00048: 3a 20 6d 75 6c 74 69 70 61 72 74 2f 6d 69 78 65 !: multipart/mixe!
00064: 64 3b 20 62 6f 75 6e 64 61 72 79 3d 3c 3c 27 3c !d; boundary=<<'<!
00080: 3c 22 3c 3c 34 32 3e 3e 22 3e 3e 27 3e 3e 0d 0a !<"<<42>>">>'>>..!
00096: 48 6f 73 74 3a 20 6c 6f 63 61 6c 68 6f 73 74 0d !Host: localhost.!
00112: 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 57 65 62 !.User-Agent: Web!
00128: 54 72 61 6e 73 61 63 74 69 6f 6e 73 20 48 54 54 !Transactions HTT!
00144: 50 2f 37 2e 35 0d 0a 43 6f 6e 74 65 6e 74 2d 4c !P/7.5..Content-L!
00160: 65 6e 67 74 68 3a 20 33 35 33 0d 0a 67 6c 6f 62 !ength: 353..glob!
00176: 61 6c 2d 68 65 61 64 65 72 2d 66 69 65 6c 64 3a !al-header-field:!
00192: 20 67 6c 6f 62 61 6c 20 63 6f 6e 74 65 6e 74 0d ! global content.!
00208: 0a 0d 0a 2d 2d 3c 3c 27 3c 3c 22 3c 3c 34 32 3e !...--<<'<<"<<42>!
00224: 3e 22 3e 3e 27 3e 3e 0d 0a 43 6f 6e 74 65 6e 74 !>">>'>>..Content!
00240: 2d 54 79 70 65 3a 20 61 70 70 6c 69 63 61 74 69 !-Type: applicati!
00256: 6f 6e 2f 78 2d 77 77 77 2d 66 6f 72 6d 2d 75 72 !on/x-www-form-ur!
00272: 6c 65 6e 63 6f 64 65 64 0d 0a 43 6f 6e 74 65 6e !lencoded..Conten!
00288: 74 2d 4c 65 6e 67 74 68 3a 20 36 37 0d 0a 70 61 !t-Length: 67..pa!
00304: 72 74 31 2d 68 65 61 64 65 72 2d 66 69 65 6c 64 !rt1-header-field!
00320: 3a 20 68 65 61 64 65 72 20 70 61 72 74 20 31 0d !: header part 1.!
00336: 0a 0d 0a 57 54 5f 53 59 53 54 45 4d 5f 42 41 53 !...WT_SYSTEM_BAS!
00352: 45 44 49 52 3d 25 32 46 68 6f 6d 65 31 25 32 46 !EDIR=%2Fhome1%2F!
00368: 70 75 6c 73 25 32 46 68 74 74 70 64 25 32 46 70 !puls%2Fhttpd%2Fp!
00384: 75 6c 73 77 77 77 26 63 6f 6d 6d 61 6e 64 3d 52 !ulswww&command=R!
00400: 65 66 72 65 73 68 0d 0a 2d 2d 3c 3c 27 3c 3c 22 !efresh..--<<'<<"!
00416: 3c 3c 34 32 3e 3e 22 3e 3e 27 3e 3e 0d 0a 43 6f !<<42>>">>'>>..Co!
00432: 6e 74 65 6e 74 2d 54 79 70 65 3a 20 74 79 70 65 !ntent-Type: type!
00448: 31 0d 0a 43 6f 6e 74 65 6e 74 2d 4c 65 6e 67 74 !1..Content-Lengt!
00464: 68 3a 20 33 38 0d 0a 70 61 72 74 32 2d 68 65 61 !h: 38..part2-hea!
00480: 64 65 72 2d 66 69 65 6c 64 3a 20 61 6e 79 20 6f !der-field: any o!
00496: 74 68 65 72 0d 0a 0d 0a 57 54 5f 53 59 53 54 45 !ther....WT_SYSTE!
00512: 4d 5f 42 41 53 45 44 49 52 3d 76 61 6c 31 26 63 !M_BASEDIR=val1&c!
00528: 6f 6d 6d 61 6e 64 3d 52 65 66 72 65 73 68 0d 0a !ommand=Refresh..!
00544: 2d 2d 3c 3c 27 3c 3c 22 3c 3c 34 32 3e 3e 22 3e !--<<'<<"<<42>>">!
00560: 3e 27 3e 3e                                     !>'>>            !
```

3.2.2 The Header attribute

The `Header` attribute is an object of `Array` type. You can use this array to supply the HTTP server with additional information at send time. At receive time, it enables you to access the information in the returned HTTP header fields.

The array is constructed as follows:

Field	Content
n	Number of the field in the array, starting with 0
n.`Name`	Name of the header field
n.`Value`	Value of the header field

Table 4: Structure of the `header` array object

If you want to use the `Header` attribute to send additional header information, you should note the following comments:

– The elements in the `Header` array have priority. This means that they overwrite the predefined header fields as well as an additional `sendData.ContentType` attribute.

Observe the notational difference between the host object attribute `ContentType` (without a hyphen '-') and content of the name attribute of the array element `senddata.Header[n].Name='Content-Type'` (with a hyphen '-').

Since names in WebTransactions may not contain any hyphens whereas hyphens are frequently found in HTTP header fields, mapping with the two attributes `Name` and `Value` in the header array was selected to make it possible to generate each header field.

```
sendData.ContentType='type1';
```

is therefore the abbreviated notation for

```
sendData.Header[0]={Name:'Content-Type', Value:'type1'};
```

or even

```
sendData.Header[n]=new Object;
sendData.Header[n].Name='Content-Type';
sendData.Header[n].Value='type1';
```

– The header fields are generated in the same sequence and using the same notation (no distinction is made between uppercase and lowercase) as in the `Header` array. Any predefined fields not present in the array are inserted first.

– Any element in the array for which the `Name` but not the `Value` attribute is set determines only the position within the generated header fields. The content can be determined by a predefined field.

The following table summarizes the possible sequences of the header fields that are to be sent:

	Predefined field	**Undefined field**
Field not present in the `Header` **array**	Field located before the fields of the `Header` array Value: internal	
Field present in the `Header` **array and has the** `Name` **attribute**	Field located at the position defined in the `Header` array Value: internal	Field located at the position defined in the `Header` array Value: empty
Field present in the `Header` **array and has the** `Value` **attribute**	Field located at the position defined in the `Header` array Value: defined in the array	

Table 5: Sequence of header fields for sending

The sequence of the header fields is not defined in the HTTP protocol. It is recommended to send the `general header fields` first followed by the `request header fields` and finally the `entity header fields`.

3.2.3 Receiving a message in the receiveData host object

The received host object, `receiveData`, has the same structure as `sendData`.

Single-part message	**Multipart message**

<pre>receiveData
      Body
      ContentType
      Header
          0
           Name
           Value
          1
           Name
           Value</pre>
	<pre>receiveData
	    0
	        [ContentType]
	        [Header]
	        Body
	  1
	        [ContentType]
	        [Header]
	        Body
	      ContentType
	      Header
	          0
	           Name
	           Value
	          1
	           Name
	           Value
	...</pre>

receiveData always contains the Header attribute. The host adapter creates an element in the Header array for each received header field.

3.3 Processing HTTP raw data

The HTTP host adapter itself merely processes complete HTTP bodies as text objects in the host objects sendData and receiveData.

To interpret the documents exchanged via HTTP, you must use special filters. This section describes the options available for defining your own send and receive functions with the corresponding filter effects.

3.3.1 WTScript filters

Within WebTransactions applications, it is possible to implement WTScript functions for processing raw data further. Examples of such functions can be found in chapter “Examples” on page 89: section “Using an existing CGI script” on page 89 and section “Communicating via HTTP and processing with WT_Filter” on page 93.

The functions for creating and interpreting HTTP messages enable you to:

- create HTTP messages from internal WTScript objects for sending to the HTTP server (ContentType and Body)
- analyze data (Body) received from the HTTP server and break it down into internal WTScript objects

3.3.2 User exits

It is also possible to use user exits in order to interpret incoming HTTP messages and create outgoing HTTP messages. This makes sense in particular if the appropriate libraries are readily available and only require minor adaptations.

3.3.3 Built-in filters

WebTransactions comes with a filter class, `WT_Filter`, which is capable of processing contents of type `text/xml`. The methods of this class allow you to:

- convert WTScript objects into an XML document and vice versa, e.g. for exchanging data with other WebTransactions applications via `WT_REMOTE`
- convert XML text into a WTScript object tree and back again into XML text, e.g. for communicating with remote applications
- analyse XML text with the integrated SAX parser
- convert WTScript method calls into XML documents and vice versa, e.g. for calling the `WT_REMOTE` interface for distributed WebTransactions applications

Further information on the `WT_Filter` class can be found in the WebTransactions manual "Template Language" (class description and examples), in the WebTransactions manual "Client APIs for WebTransactions", and in section "Communicating via HTTP and processing with WT_Filter" on page 93 of the present manual.

3.4 Start templates for HTTP

When you start the WebTransactions application (from a home page or by directly specifying a URL), you must define the parameters for the connection to the partner application in a start template.

WebTransactions supplies you with a set of ready-made start templates which you can use as the basis for your own start templates. There are two variants:

- **Start template set (ready to use)**

  This start template set is ready to use. All required parameters can be entered in a dialog. The start template set is suitable both for starting an individual host application and for starting multiple host or partner applications integrated in a WebTransactions application. It consists of the general start template `wtstart.htm (`which can be used, for example, to create communication objects and switch back and forth between a number of parallel host connections) plus specific start templates for the individual host adapters. The start template `wtstartHTTP.htm` is supplied specifically for use with WebTransactions for HTTP, and is described in section “HTTP-specific start template of the start template set (wtstartHTTP.htm)” on page 36. The general start template is described in the WebTransactions manual “Concepts and Functions”.

- **Simple start template (must be customized)**

  When connecting to an individual HTTP server, you can use the simple start template `StartTemplateHTTP.htm`. Thanks to its clear structure and detailed comments, this start template is ideal for defining concrete start parameters for the immediate execution of the WebTransactions application. Unlike the start template set, these start parameters do not need to be defined on every restart. The simple start template is described in section “Simple start template (StartTemplateHTTP.htm)” on page 40.

  This simple start template is also suitable as a basis for your own WTScript functions which retrieve information from the Web via HTTP.

3.4.1 HTTP-specific start template of the start template set (wtstartHTTP.htm)

If you select the `HTTP` protocol in the general start template `wtstart.htm` (described in the WebTransactions manual “Concepts and Functions”) and create a new communication object, the system branches to the `wtstartHTTP.htm` template.

`wtstartHTTP.htm` allows you to interactively set the connection parameters and begin communication.

The figure below shows the interface generated by `wtstartHTTP.htm`.

Fujitsu Powered by WebTransactions

HTTP communication

status	communication object	**WT_HOST.HTTP_0**
	HTTP_RETURN_CODE:	200
workflow	destination:	main menu [go to]
	access URL:	[send] [receive]
	parameters:	[update] [reset]
connection parameters	URL:	example.net/wtscripts/WTPublish.exe/startup
	USER:	
	PASSWORD:	
	PROXY:	
	PROXY_PORT:	
	PROXY_USER:	
	PROXY_PASSWORD:	
	TIMEOUT_HTTP:	60 (1 minute)
	SSL_CERT_FILE:	
	SSL_KEY_FILE:	
	SSL_PASSPHRASE:	
	SSL_PROTOCOL:	ALL
host objects	sendData.ContentType	
	sendData.Body	WT_SYSTEM_BASEDIR=d:/basedirs/manual/beispiel_osd&WT_SYSTEM_FORMAT=wtstart
	receiveData.ContentType	text/html;charset=ISO-8859-1
	receiveData.Header	[0]:Date=Mon, 14 Jun 2010 12:37:42 GMT [1]:Server=Apache/2.2.12 (Win32) DAV/2 mod_ssl/2.2.12 OpenSSL/0.9.8k mod_autoindex_color PHP/5.3.0 mod_perl/2.0.4 Perl/v5.10.0 [2]:Connection=close [3]:Content-Type=text/html;charset=ISO-8859-1
	receiveData.Body	Display in a separate window as [Text] [HTML]

The **status** section contains the following information:

communication object
Name of the underlying communication object.

HTTP_RETURN_CODE
Return code of the last `receive` call.If no response was received, e.g. because the addressed server is not available, then `HTTP_RETURN_CODE` is empty. A table with the possible errors and their meaning is given in the section “HTTP error messages” on page 101.

The **workflow** section allows you to define what happens next.

destination
Here you can actively choose between the various start templates. To branch to the selected page, click on `go to`. The option `main menu` is proposed by default, and allows you to return to the general start page `wtstart.htm`. If several connections are open, these are also offered for selection and enable you to branch to the respective host-adapter-specific start templates of these connections.

access URL
The buttons beside `access URL` are used to execute the `send` and `receive` methods. The connection parameters defined below are transferred and activated.

parameters
With `reset`, you can reset all the parameters to the state they had when received from the browser. With `update`, you can send the current parameter values to WebTransactions without communicating via HTTP.

The **connection parameters** section specifies the system object attributes:

URL Enter the URL of the HTTP resource you require.

USER, PASSWORD
The values entered here are used for accessing protected resources.

PROXY, PROXY_PORT
If the connection is to be established via a proxy server, you must specify the appropriate Internet address or symbolic name under `PROXY`. `PROXY_PORT` indicates the port of the proxy server.

PROXY_USER, PROXY_PASSWORD
Any values entered for `PROXY_USER` and `PROXY_PASSWORD` are sent in the event of access via a proxy system.

TIMEOUT_HTTP
Sets the time spent waiting for a response from the HTTP host adapter.

SSL_CERT_FILE
Here, enter the name of the file containing the client-side certificate to be used. If the file name is not entered as absolute, it will be relative to the base directory.

SSL_KEY_FILE
Here, enter the name of the file containing the private key of the certificate (see CERT_FILE). If the file name is not entered as absolute, it will be relative to the base directory.

SSL_PASSPHRASE
Here, enter the passphrase to be used with the private key.

SSL_PROTOCOL
Here, select the SSL or TLS version you wish to activate. OpenSSL supports SSL protocol versions 2 and 3, and TLS protocol version 1. Possible values:

- `SSLv2`
- `SSLv3`
- `TLSv1`
- `All` (default)

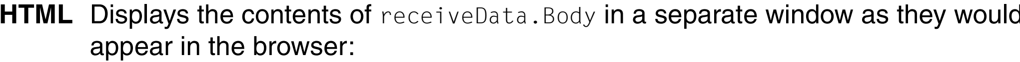
You can enter several protocols but these should be separated by blank spaces.

The **host objects** section displays the `sendData` host object, and allows you to modify its `ContentType` and `Body` attributes.

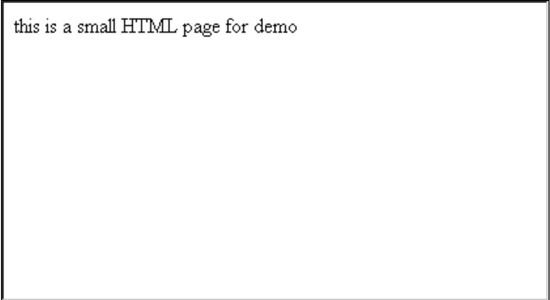
If data has already been received, i.e. `receiveData` exists, the content type of `receiveData` is also output and this section contains two additional buttons:

Text Displays the contents of `receiveData.Body` in a separate window in the form of HTML source code:

```
      <HTML>
  <TITLE>
    Small HTML page to be displayed
  </TITLE>
  <BODY>
    this is a small HTML page for demo
  </BODY>
</HTML>
```



HTML Displays the contents of `receiveData.Body` in a separate window as they would appear in the browser:



The **receiveData.Header** field also shows the header field returned.

3.4.2 Simple start template (StartTemplateHTTP.htm)

The HTTP host adapter comes with a simple WTML document containing prototypes of all statements used to define the connection parameters and of the host object attributes. This template can be customized in order to quickly produce WTScripts for accessing HTTP resources.

This template can be used either as a start template (e.g. for a single-step transaction or in the middle of a session. The following example contains detailed comments to help you identify the meaning of the individual sections.

Template format

The start template begins with a series of HTML comments which indicate, among other things, the format of the URL used to call the start template:

```
<wtRem>
   -----------------------------------------------------------------------
--
   StartTemplateHTTP
   -----------------------------------------------------------------------
--
   may be called by the url:
      http://<host>/<urlprefix>/WTPublish.exe/<basedirectory>?<starttemplate>
   (e.g.  http://localhost/cgi-
bin/WTPublish.exe/c:/base_http?StartTemplateHTTP )
   this document also may be included for an HTTP call while running a
   WebTransactions application:
      <wtInclude name="StartTemplateHTTP">
```

```
   ---------------------------------------------------------------------------
--
   Copyright (c) by Fujitsu Technology Solutions GmbH, 2010
</wtRem>
```

The actual content of this start template consists of an OnCreate script:

```
...
<wtoncreatescript>
<!--
 ...
//-->
</wtoncreatescript>
```

Code of the StartTemplateHTTP.htm template

The OnCreate script is divided into three sections:

The first section creates a new communication object and thus implicitly a private system object. It also activates the HTTP host adapter (`open`).

```
<wtoncreatescript>
<!--
  // *****************************************************************
  // global part of starttemplate which is not dependent of protocol type.
  // these attributes are not relevant, if you use this template as include
  // for a running session.
  // *****************************************************************

  // If a specific style or language is required, you have to set the
following
  // attributes to a suitable value. Default value is empty for forms
directory.

  // WT_SYSTEM.STYLE = "";
  // WT_SYSTEM.LANGUAGE = "";

  // Application timeout and user timeout are set to standard values.

  // WT_SYSTEM.TIMEOUT_APPLICATION = "120";
  // WT_SYSTEM.TIMEOUT_USER        = "600";

  // If a template should be displayed after TIMEOUT_USER and before
  // terminating the session you have to set the TIMEOUT_FORMAT attribute
  // to that specific template name.

  // WT_SYSTEM.TIMEOUT_FORMAT   = "";
```

```
  // Now we have to create a new communication object. This is mandatory
also.
  // Further we use the private wt_system object
(WT_HOST.<myHandle>.WT_SYSTEM).

  // ****************************************************************
  // specific part of starttemplate for protocol type HTTP
  // ****************************************************************
  // A new communication object has to be created.
  // The name "myHTTPConn" of the object may be changed to any valid symbol.
  host = new WT_Communication("myHTTPConn");

  // Further the private system object (host.WT_SYSTEM) is used.
  host_system                          = host.WT_SYSTEM;

  // The call of open method enables the HTTP adapter
  host.open("HTTP");
```

The second section contains the assignments for the attributes of the private system object. These must be set to the appropriate values.

```
  // The URL specifies the desired HTTP resource
  // general format is
  // [http[s]:][//][user[:password]@]hostname[:port][/pathinfo[?query]]

     host_system.URL                   = "myURL";
  //   = "localhost/wtadm.htm"  // example for GETting a static file
  //   =
"//localhost/Scripts/WTPublish.exe/d:/inetpub/wwwroot/basedir/?wtstartHTTP";
                                // example for calling cgi program with
method GET
  //   =
"//localhost/Scripts/WTPublish.exe/d%3A%5Cinetpub%5Cwwwroot%5Cbasedir?wtstart
HTTP";
                                // same example but with escape sequences for
:(=%3A) and \(=%5C)
                                // maybe necessary for HTTP daemon
  //   = "//localhost/Scripts/WTPublish.exe/startup";
                                // same example using metod POST
                                // corresponds with content of sendData
object

  // For restricted HTTP resources user authorization has to be done.

  // host_system.USER                  = "";
  // host_system.PASSWORD              = "";
```

```
  // Address and port of HTTP proxy might be specified.

  // host_system.PROXY                    = "";
  // host_system.PROXY_PORT               = "";

  // For restricted PROXY access user authorization has to be done.

  // host_system.PROXY_USER               = "";
  // host_system.PROXY_PASSWORD           = "";

  // Timeout for HTTP may be determined.

  // host_system.TIMEOUT_HTTP             = "60";

  // if SSL shall be used (see URL: 'https: ...'), the following attributes
may configure the SSL connection
  // host_system.SSL_PROTOCOL             = 'All'; // SSLv2, SSLv3, TLSv1 or
All, multiple values may be defined separated by space
  // host_system.SSL_CERT_FILE            = '';    // file name of certifcate
  // host_system.SSL_KEY_FILE             = '';    // file name of matching key,
may be omitted, if SSL_CERT_FILE contains both: certificate and key
  // host_system.SSL_PASSPHRASE           = '';    // passphrase if needed for
the certifcate

  // For invocation of POST method host object sendData has to be defined.

  // host.sendData                        = new Object();
  // host.sendData.ContentType            = "application/x-www-form-urlencoded";
  // host.sendData.Body                   =
"WT_SYSTEM_BASEDIR=d%3A%5Cinetpub%5Cwwwroot%5Cbasedir&WT_SYSTEM_FORMAT=wtstar
tHTTP";
            // see comment on system attribute URL
```

Finally, the third section contains the `send` and `receive` method calls.

```
  // Now the HTTP resource is requested.
  host.send();
  host.receive();

  // The response receiveData has to be analysed.
  // document.write( host.receiveData.Body );

  // Close HTTP communication
  host.close();

  // If this was the whole job, terminate the session
```

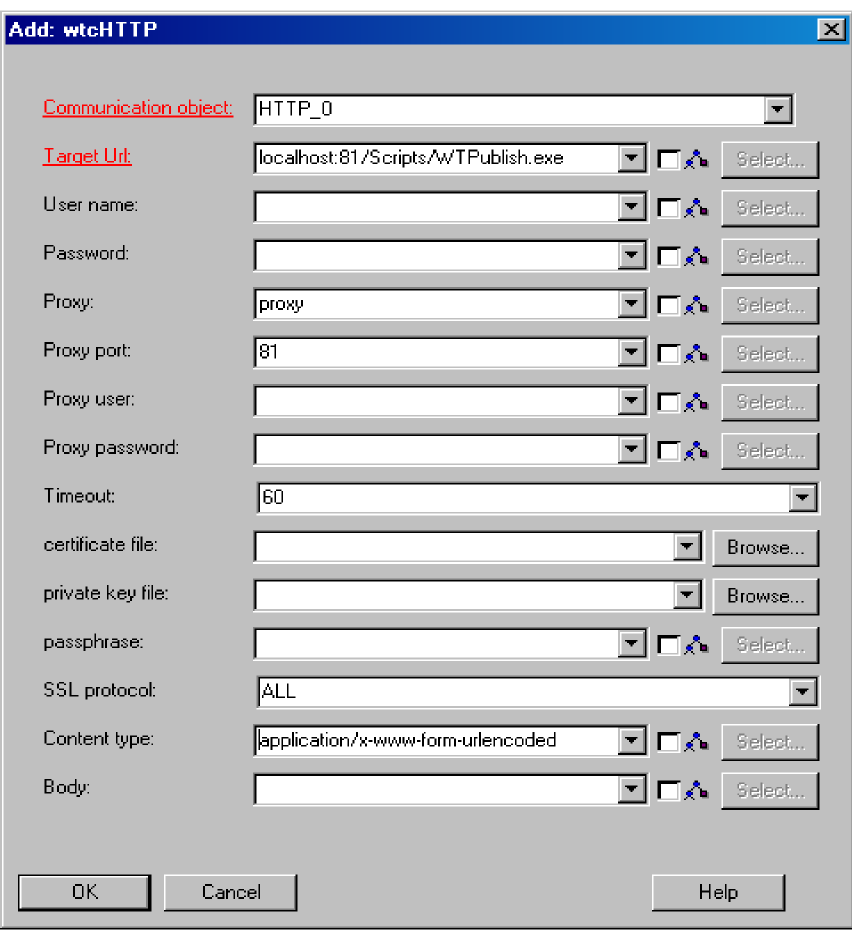
```
  // exitSession();

//-->
</wtoncreatescript>
```

3.5 Creating a new HTTP communication object (wtcHTTP)

The WTBean `wtcHTTP` is supplied in order to enable you to create a new HTTP communication object in a template and thus establish a connection to an HTTP server. You can also use this WTBean to open multiple connections in parallel. `wtcHTTP` is an inline WTBean. For more information, see the WebTransactions manual “Concepts and Functions”.

i Before you can access an inline WTBean, there must be a connection to the host application and the template in which you want to insert the WTBean must be open.

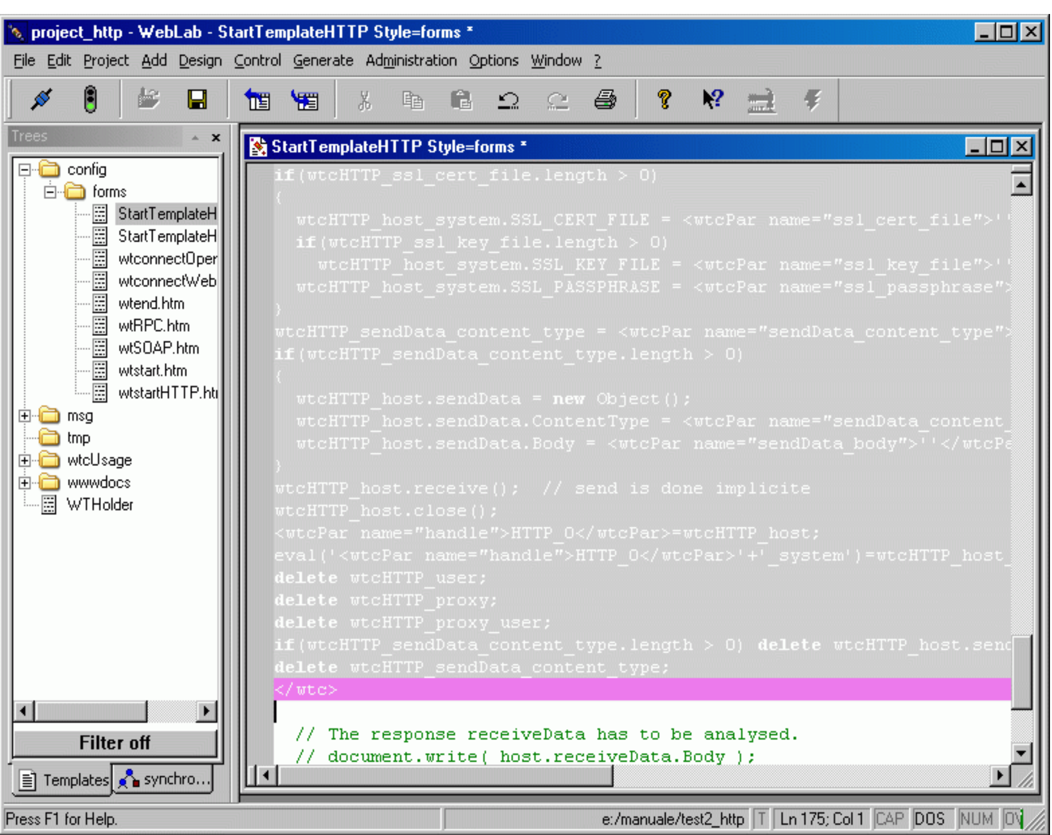
You use the **Add/WTBean/wtcHTTP** command to open the WTBean for editing. WebLab generates the dialog box **Add:wtcHTTP**:



In this dialog box, you can edit the parameters for the communication object that is to be generated. The parameters that you can enter in the input fields of this dialog box correspond to the parameters in the start template, see also section “HTTP-specific start template of the start template set (wtstartHTTP.htm)” on page 36. The mandatory parameters are displayed in red.

When you have entered values for the parameters and clicked **OK** to confirm, the parameters and description file are used to generate the code of the WTBean which is then inserted at the cursor position in the open template.

The WTBean consists of protected and unprotected code areas. The protected areas are displayed with a gray background. You can only work in these areas via the WTBean’s user interface. To do this, choose the **Edit WTBean** command in the context menu of the WTBean’s start line (pink background) (see the WebTransactions manual ‘Concepts and Functions’).



4 Connecting web services via SOAP

This chapter describes how the SOAP protocol (**S**imple **O**bject **A**ccess **P**rotocol) is integrated into WebTransactions:

- Concept underlying SOAP integration in WebTransactions
- `WT_SOAP` class
- `WT_SOAP_HEADER` class

4.1 Concept of SOAP integration in WebTransactions

The integration of the SOAP protocol in WebTransactions is based on the WebTransactions HTTP host adapter as well as the definitions of the SOAP and WSDL (**W**eb **S**ervices **D**escription **L**anguage) protocols.

4.1.1 SOAP (Simple Object Access Protocol)

The XML-based SOAP protocol provides a simple, transparent mechanism for the exchange of structured, typed information between systems within a decentralized, distributed environment.

SOAP provides a modular package model together with mechanisms for data encryption within modules. This permits a simple description of the external interfaces for a remote application that can be accessed in a web (web service).

SOAP is an XML based protocol consisting of three components:

- Envelope specification

  The envelope defines the guidelines that describe:

  – **what** is present in a message,
  – **who** is to process the message and **how** it is to be processed,
  – **whether** the individual data items in the message are optional or have to be specified (mandatory).

  The namespace identifier of the envelope is:

  `"http://schemas.xmlsoap.org/soap/envelope"`

  (Here, the term “namespace” refers to the set of names that are valid in a given context.)

- A set of coding and serialization rules that describes instances of application-specific data types.

  The namespace identifier for serialization is:

  `"http://schemas.xmlsoap.org/soap/encoding"`

- A convention for the representation of remote procedure calls (RPC) and possible responses to these RPCs.

SOAP services are described in WSDL documents.

4.1.2 Describing SOAP services with WSDL

WSDL (**W**eb **S**ervices **D**escription **L**anguage) provides XML language rules for describing the capabilities of web services. Multiple SOAP services can be described in a single WSDL document. These SOAP services may, in turn, be available on a number of different servers. The input and output parameters for the individual services can also be described using WSDL.

The figure below provides an outline illustration of a WSDL document.

WSDL Document

<types>

Description of extended data structures

<message>

Description of the messages and associated parameters which can be sent to the WebService or received by the WebService. If complex data types are used as parameters, <message> contains references to <types>.

<port type>

<operation>

Description of a web service function together with a <message> reference

<binding>

Description of the concrete protocol and data format for the work steps and messages given by a specific port type.

<service>

<port>

Server description (address)

Figure 2: Structure of a WSDL document

4.1.3 UDDI (Universal Description, Discovery and Integration Project)

The **U**niversal **D**escription, **D**iscovery and **I**ntegration Project (UDDI) is a comprehensive, platform.independent system for the documentation, among other things, of web services. All web users have unrestricted access to the UDDI directories in which the web services are documented.

4.1.4 SOAP support in WebTransactions

WebTransactions provides client-side access to SOAP services via the HTTP protocol using the `WT_SOAP` class that is written in WTML. Consequently, only the HTTP host adapter and the `WT_SOAP` class are required for WebTransactions SOAP support. You can call Web services in accordance with the SOAP standards V1.1 and V1.2.

SOAP services are described in WSDL. WebTransactions can read the WSDL from a local file or load it from the network. When a WSDL `WT_SOAP` object is created, the corresponding service is made available as a proxy method for this object. If the proxy method has call parameters then proxy objects will be created for these parameters. When the proxy method is executed, the query message is sent to the SOAP server via the HTTP host adapter and the response message is received via the same path before being transformed into a WTScript.

The figure below illustrates this process.

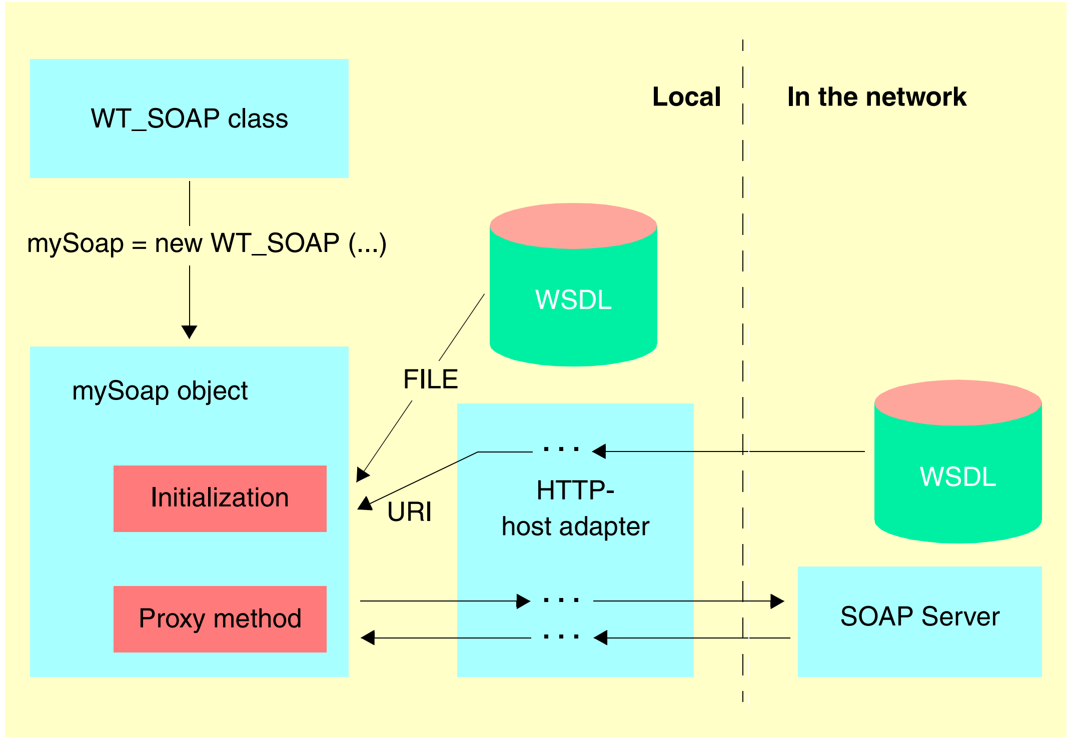


Figure 3: Client-side SOAP support using the WT_SOAP class

If you want to test a web service, you can use WebLab to generate a default interface for the web service (**Generate/Templates/for WebServices**). You can call this template from the general start template `wtstart.htm`. For more detailed information on starting a web service, see the WebTransactions manual “Web Frontend for Web-Services”.

SOAP support is a standard component of all WebTransactions supply units. During creation of a base directory for the HTTP host adapter (see chapter “Creating a base directory” on page 15), a corresponding reference to the `wtSOAP.htm` is created in the installation directory.

4.2 WT_SOAP - client-side class

The `WT_SOAP` class is implemented in the `wtSOAP.htm` template.
The template contains the constructor of the `WT_SOAP` class which generates an instance of the class from a WSDL file of a web service. This instance makes proxy methods available which are used to call an operation. In addition to the proxy methods, there are a whole series of other methods which enable setting of a `WT_SOAP` class object.

The `WT_SOAP` class uses WebTransactions exception handling (see the WebTransactions manual “Template Language”). If you use the `WT_SOAP` class then it is advisable to call the methods associated with this class (including the constructor) from within a `try` block and then handle any errors in a `catch()` block. If you do not do this, any errors present will be perpetuated right through to the user interface.

Once the `WT_SOAP` class has been instantiated, certain attributes and methods are available immediately as `WT_SOAP` object attributes and methods. However, most `WT_SOAP` object methods are used internally, i.e. during the instantiation or execution of proxy methods (see page 62). Consequently, the following sections describe only those methods that are components of the user interface.

The following special characteristics concerning SOAP functionality should be borne in mind when the `WT_SOAP` class is used:

- The SOAP header is not automatically generated from the WSDL. However, you can add a header to the message. To do this, insert the header in `WT_SOAP.envelope.header` before calling the method or create the header of the `WT_SOAP_HEADER` class. This is then taken over automatically.
- The simple data types `ID` and `IDREF` are not supported. The corresponding data can be transferred as redundant.
- Only messages which use “SOAP binding” and which are sent via `http` are supported.
- SOAP body encoding is only supported as specified in

  `"http://schemas.xmlsoap.org/soap/envelope"`

  If a different schema is used for data serialization then the `WT_SOAP` class can only be used as the basis for a separate class.
- Only data descriptions in XML schema (XSD) are automatically supported.

4.2.1 Structure of a WT_SOAP object

This section explains the structure of a WT_SOAP object with the aid of an example in the WebLab object tree. The WT_SOAP object is generated from the WSDL document described in section "Example: WSDL document" on page 58.

The individual components of a WT_SOAP object are explained in detail in the proximate sections (as of page 60).

4.2.1.1 Representation in the WebLab object tree

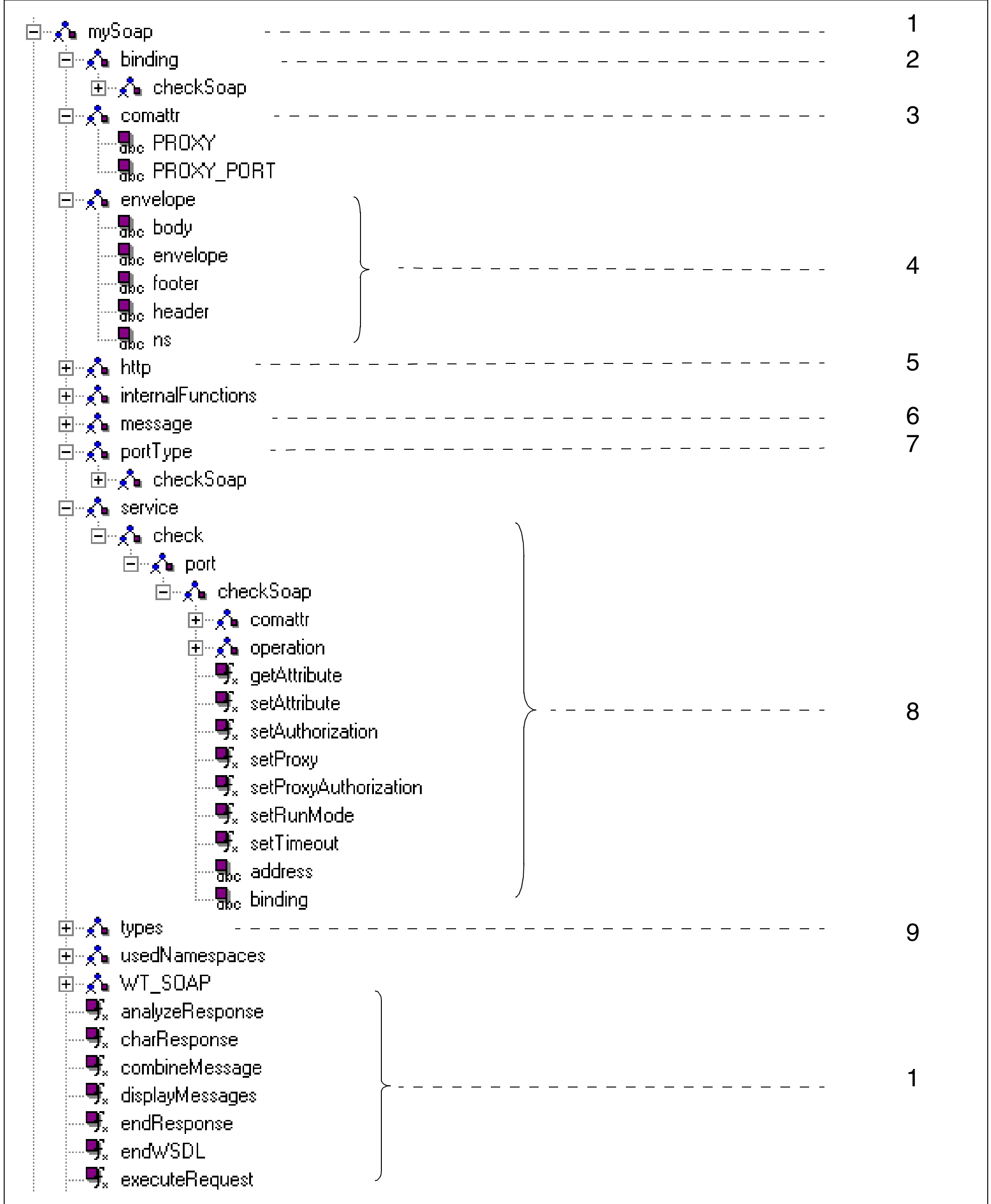


Figure 4: Representation of a WT_SOAP object in the WebLab object tree

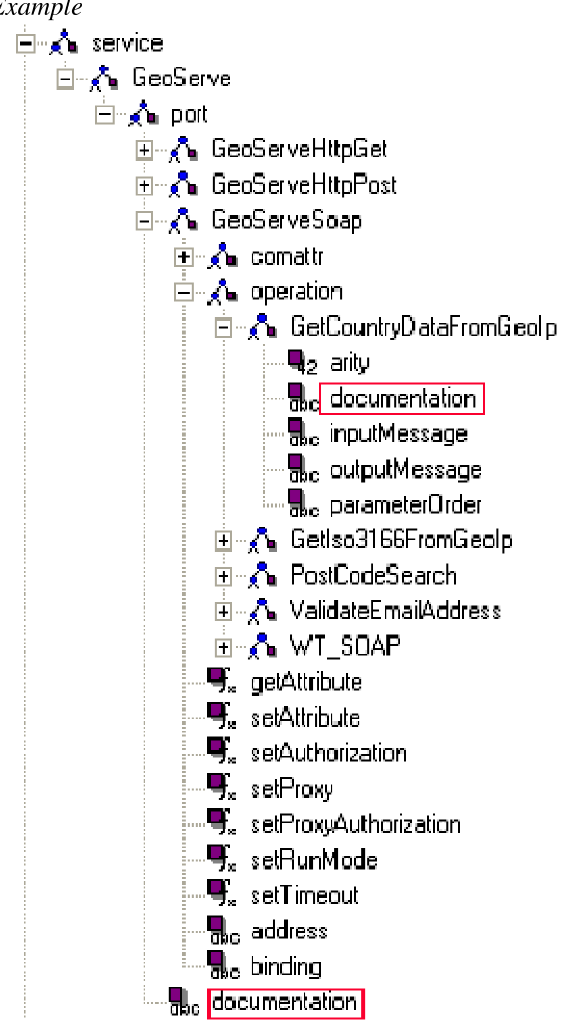
Explanation (see also section “WT_SOAP attributes” on page 79)

1. mySoap object after instantiation. The methods defined under WT_SOAP can be used in connection with instances of the WT_SOAP class and therefore also with mySoap (see sections “WT_SOAP class object methods” on page 65 and “Methods for configuring access to the WT_SOAP_COM_FUNCTIONS subclass” on page 75).
2. Map of the <binding> section from the WSDL document.
3. commattr object containing the attributes that control HTTP access.
4. The envelope object contains the envelope, header and body attributes which in turn contain the text segments used to construct the SOAP message when calling a proxy method or the executeRequest method.
5. The http object contains the communication object for the HTTP host adapter.
6. The message object maps the <message> section of the WSDL document.
7. The portType object contains an object for each operation described in the <portType> section of the WSDL (in this case, checkSoap).
8. The service object contains an object for each service provided by the web service. There is a separate <service>...</service> section in the WSDL for each service (in our example this is: check). check contains a port object and, below this, information from the WSDL document <port> section (in our example: checkSoap). checkSoap in turn contains the objects commattr, operation and adress as well as a number of methods that can be used to perform port-specific operations. The most important object is operation since this contains the proxy methods used to call the web service operations. The comattr object contains the port-specific attributes for HTTP access that are used when the proxy method is called. adress contains the web service URI.

   When defining a service and an operation in the WSDL, you can use a documentation element to insert comments. This information is taken over into the WT_SOAP structure. It is helpful to provide this type of information when generating user interfaces for web services. For this reason, the comments are stored at the following locations:
   - If a documentation element is found within the service tag in the WSDL, under WT_SOAP.service.<*name of service*>.port.documentation.
   - If a corresponding comment is found in the operation tags in the WSDL, under the name of the operation.

   A documentation element may contain text, any required XML structure, or both. In practice, only simple texts are found. As a result, only the text content of the documentation element is taken over as a string into the WT_SOAP structure.

Example



9. The `types` object is used for storing the data types described in the WSDL document.

Web services on different hosts

A web service can be provided on different hosts. In these cases, the `<service>` section of the WSDL document contains several `<port>` sections. Because the various servers generally use different parameters, (proxy etc.), this is reflected in the structure of the `WT_SOAP` object under `service`/***servicename***/`port`. In this case, therefore, the `port` object is assigned several objects with differing port-specific settings.

By the same token, proxy methods for calling the web service which are provided under `service`/***servicename***/`port`/***portname***/`operation` in the object tree must be present several times. This is, however, not strictly redundant since each *portname* is connected to a different internet address and thus any call on a different port communicates with a different server on the internet.

The methods and attributes for setting hosts and access rights for access to the web service are available both under the `WT_SOAP` object and under `service`/***servicename***/`port`/***portname***. This allows you to make the settings either globally or on a port specific-basis.

4.2.1.2 Example: WSDL document

The section below shows a web service used to check the spelling of a text. This WDSL document is the basis for the object tree shown on page 54. The WTScript in the section “Example: Checking the spelling of a text” on page 83 uses this WSDL and calls the web service.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:s="http://www.w3.org/2001/XMLSchema"
   xmlns:s0="http://ws.cdyne.com/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
   xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
   targetNamespace="http://ws.cdyne.com/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://ws.cdyne.com/">
      <s:element name="CheckTextBody">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="BodyText" type="s:string" />
            <s:element minOccurs="0" maxOccurs="1" name="LicenseKey" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="CheckTextBodyResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="DocumentSummary"
                       nillable="true" type="s0:DocumentSummary" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:complexType name="DocumentSummary">
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="unbounded" name="MisspelledWord"
                     type="s0:Words" />
          <s:element minOccurs="0" maxOccurs="1" name="ver" type="s:string" />
          <s:element minOccurs="0" maxOccurs="1" name="body" type="s:string" />
          <s:element minOccurs="1" maxOccurs="1" name="MisspelledWordCount"
                     type="s:int" />
        </s:sequence>
      </s:complexType>
      <s:complexType name="Words">
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="unbounded" name="Suggestions"
                     type="s:string" />
          <s:element minOccurs="0" maxOccurs="1" name="word" type="s:string" />
          <s:element minOccurs="1" maxOccurs="1" name="SuggestionCount" type="s:int" />
        </s:sequence>
```

```
      </s:complexType>
   <s:element name="DocumentSummary" nillable="true" type="s0:DocumentSummary" />
      <s:element name="string" nillable="true" type="s:string" />
    </s:schema>
  </types>
  <message name="CheckTextBodySoapIn">
    <part name="parameters" element="s0:CheckTextBody" />
  </message>
  <message name="CheckTextBodySoapOut">
    <part name="parameters" element="s0:CheckTextBodyResponse" />
  </message>
  <portType name="checkSoap">
    <operation name="CheckTextBody" parameterOrder="parameters">
      <input message="s0:CheckTextBodySoapIn" />
      <output message="s0:CheckTextBodySoapOut" />
    </operation>
  </portType>
  <binding name="checkSoap" type="s0:checkSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
    <operation name="CheckTextBody">
      <soap:operation soapAction="http://ws.cdyne.com/CheckTextBody"
                      style="document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>   </binding>
  <service name="check">
    <port name="checkSoap" binding="s0:checkSoap">
      <soap:address location="http://ws.cdyne.com/SpellChecker/check.asmx" />
    </port>
  </service>
</definitions>
```

4.2.2 Constructor for the WT_SOAP class

The constructor analyses the received WSDL file and from this creates a WTScript object. The constructor also creates a communication object named WT_SOAP_first free number under WT_HOST. All HTTP accesses, including the loading of the WSDL document as URI (**U**niform **R**esource **I**dentifier) are performed via this communication object.

WT_SOAP()
WT_SOAP(*init_document*)
WT_SOAP(*init_document*, *http_proxy_host*)
WT_SOAP(*init_document*, *http_proxy_host*, *http_proxy_port*)

Return value

Instance of the `WT_SOAP` class for SOAP support.

Parameters

init_document
: Specifies the URI or the file containing the WSDL document. File names must be relative to the base directory.

http_proxy_host
: Specifies the PROXY host that was used on loading the WSDL document and is also to be used by the HTTP host adapter for the actual SOAP request.

http_proxy_port
: Specifies the PROXY port that was used on loading the WSDL document and is also to be used by the HTTP host adapter for the actual SOAP request.

Example

```
//Initialize WT_SOAP-Object from file (file is in
                     <basedir>/wsdl/check.wsdl)
mySoap = new WT_SOAP('wsdl/check.wsdl', 'proxy','81');
```

Constructor call without parameters

You must call the constructor without arguments in those cases where other attributes in addition to *http_proxy_host* and *http_proxy_port* are required for the loading of the WSDL document when initialization is performed. After this you will be able to use the `WT_SOAP_COM_FUNCTIONS` class methods (see page 75) to make all the necessary settings. After this, initialize the object using the `WT_SOAP.initFromWsdlUri` method (see page 65).

Example

```
mySoap = new WT_SOAP();
mySoap.setProxy ('proxy.company.com','81');
mySoap.setProxyAuthorization('puser','ppass');
mySoap.initFromWsdlUri('http://url');
```

Exceptions

In the event of errors, the constructor will output the exceptions `SOCKET`, `HTTP`, `FILE` or `WSDL` (see section “Exceptions” on page 77).

4.2.3 Proxy methods

Initializing a `WT_SOAP` object in order to call a web service via interpretation of a WSDL document dynamically generates methods. Methods generated dynamically in this way are created on the basis of the WSDL object structure under:

`service`/***servicename***/`port`/***portname***/`operation`

(See “Representation in the WebLab object tree” on page 54.)

If the web service provides several functions with the same name then these functions will be created as array. The individual methods are then accessed as elements of this array.

Proxy method calls and parameter transfers

In order to abbreviate the notation required to call proxy methods, you can create references like this example below.

Example

```
myOperations =
mySoap.service.check.port.checkSoap.operation;
myOperations.CheckTextBody(. . .);
```

Parameters are passed as follows when a proxy method is called:

– Parameters which have a simple data type can be passed “by value” as arguments when the proxy method is called:

```
myOperations.SendMail('recipients@domain','sender@domain,'subject',
                                                'this is the Message');
```

– For complex data types, specially structured proxy objects are created as global objects under the `proxyObjects` global object; these objects can be filled with real data and transferred as arguments at calls. The name of this proxy object is constructed from the name of the message to be set (see the attribute `inputMessage`) and the parameter name.

– Attributes in a message or response are stored in the object `wt_attributes`.
This has the following implications for the user interface:

  – The `wt_attributes` object containing the attributes may be present anywhere in the proxy objects.

    If the element is a simple data type then it is converted into the corresponding object data type (string, boolean, number) to make it possible to incorporate the attributes.

Example

The input parameter of a method for translating text - with the source and target language in attributes - looks like this:

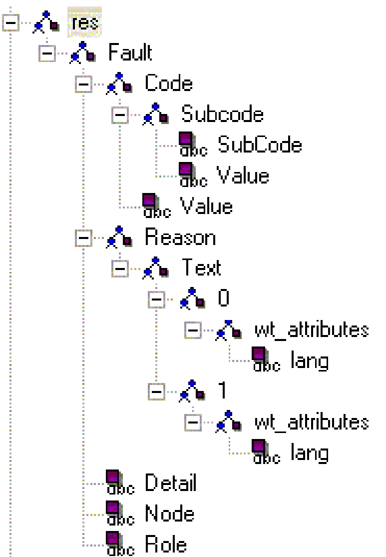
```
⊟ proxyObjects.GetTranslation_Translate
     Text                                   Text to translate
   ⊟ wt_attributes
        destLang                            de
        sourceLang                          en
```

This causes the generation of the following message:

```
….
      <wt_tns0:GetTranslation>
         <Translate destLang="de" sourceLang="en">
            <Text>Text to translate</Text>
         </Translate>
      </wt_tns0:GetTranslation>
…
```

– When the response is analyzed, attributes are again stored under the object `wt_attributes`.

Example l



Generating the SOAP message without sending it

If the web service requires a function which is not supported by the `WT_SOAP` class then it will call the proxy method in the `PREPARE` mode (see section "setRunMode method" on page 66). In this case the proxy method prepares the HTTP message but does not send it. You can manipulate the data under `WT_SOAP.envelope`, edit the message and then send it using `WT_SOAP.executeRequest`. The result can be converted into a WTScript data structure using `WT_SOAP.anlyseResponse`.

For an example, see section "setRunMode method" on page 66.

Setting HTTP headers

When you execute a proxy method, the HTTP headers `Content-Type` and `SOAPAction` are generated automatically.

If you want to add further HTTP headers to the message, you can do this using the function `setHTTPHeader()`. When you have called this function, this header is added to every subsequent message.

```
setHTTPHeader (name, value)
setHTTPHeader (headerObj)
```

name Name of the header field

value Value of the header field

headerObj
Object with the attributes *name* and *value* which contain the name and value of the required header field.

Return value from a proxy method

A proxy method returns an object that contains the response from the SOAP service:

- If the operation is successful then the proxy method returns a structure that corresponds to the `<output message=`*outputmessage*`>` definition in the associated WSDL document.
- If an error occurs, the web service should return a fault message which is converted into the appropriate structure. The structure contains the mandatory attribute `faultcode` and the optional attributes `faultstring`, `faultfactor` and `detail`.

  If the web service sends a message whose structure does not match that of a fault message, then the return value of the proxy method will be structured in the same way as the returned message.

Exceptions

If an error occurs, the proxy method will output the exceptions `SOCKET`, `HTTP` or `PARAMETER` (see section “Exceptions” on page 77).

4.2.4 WT_SOAP class object methods

The methods described in this section are available immediately in the instance after instantiation of the `WT_SOAP` class. These methods and all `WT_SOAP_COM_FUNCTIONS` class methods (see section “Methods for configuring access to the WT_SOAP_COM_FUNCTIONS subclass” on page 75) can be called from a `WT_SOAP` class object.

4.2.4.1 initFromWSDLUri method

Sometimes it is not possible for the constructor to transfer the remote WSDL when creating an instance of the `WT_SOAP` class. This situation may arise, for example, when additional parameters are required at the HTTP interface in order to transfer the WSDL file. In these cases, you can use the `initFromWSDLUri` method to perform the analysis of the WSDL file.

```
initFromWsdlUri(uri)
```

Return value

No return value

Parameter

uri Specifies the URI (**U**niform **R**esource **I**dentifier) of the WSDL document that contains the description of the web service.

Exceptions

In the event of errors, `initFromWsdlUri` will output the exceptions `SOCKET`, `HTTP` or `WSDL` (see section “Exceptions” on page 77).

Example

```
mySoap = new WT_SOAP();
mySoap.setProxy ('proxy.company.com','81');
mySoap.setProxyAuthorization('puser','ppass');
mySoap.initFromWsdlUri('http://url');
```

4.2.4.2 setRunMode method

The `setRunMode` method sets the mode for performing the proxy method (see section “Proxy methods” on page 62).

```
setRunMode("mode")
```

Return value

No return value

Parameter

mode Specifies the proxy method mode.

You can enter the following values for *mode*:

- `PREPARE`:
  A proxy method called after this mode prepares a data structure for HTTP access but does not execute the access.
- `DOCUMENT`
  The proxy method parameters will be understood as an XML document and will be inserted without modifications directly after `<body>` in the SOAP message. The result will not be converted to a WTScript data type but will be output as a `string`.
- `RUN`
  This resets the mode, i.e. the proxy method will run the web service. The parameters must be transferred as WTScript data types.

Exceptions

The `setRunMode` method does not output exceptions.

Example: setRunMode("PREPARE")

```
try{
mySoap = new WT_SOAP('wsdl/check.wsdl','proxy','81');
}
catch (e)
{
//do something
}
myOperations =
mySoap.service.check.port.checkSoap.operation;
mySoap.setRunMode('prepare'); //Just build the message, do not send it
//Fill Parameter with values
```

```
proxyObjects.CheckTextBodySoapIn_parameters.LicenceKey="0";
proxyObjects.CheckTextBodySoapIn_parameters.BodyText = "This is a sample text";
try{
//call proxy method
    myOperations.CheckTextBody(proxyObjects.CheckTextBodySoapIn_parameters);
}
catch (e)
{
}
//Set additional HTTP-header
mySoap.setHTTPHeader( 'Additional-Header-Field','something');
try {
    mySoap.executeRequest();//Send Request
    myRet=mySoap.analyzeResponse(); //and convert answer into a WTScript-Object
}
catch (e)
{
}
```

4.2.4.3 executeRequest method

The `executeRequest` method executes the current request. The message is constructed of the attributes `envelope`, `header`, `HeaderBlocks` and `body` of the instance `envelope` object and sent via the HTTP host adapter (see section “WT_SOAP attributes” on page 79). You will need this method together with the `setRunMode("PREPARE")` method which prepares a web service message but does not send it (see section “setRunMode method” on page 66).

```
executeRequest()
```

Return value

No return value

Exceptions

If an error occurs, `executeRequest()` will output the exceptions `SOCKET`, `HTTP` or `PARAMETER` (see section “Exceptions” on page 77).

4.2.4.4 executeGetRequest method

In SOAP V1.2, you are recommended to use the HTTP `GET` method if information is simply to be fetched without any header being sent. As usual, the response is returned as a SOAP envelope.

To make this possible, `WT_SOAP` provides the method `executeGetRequest`.

`WT_SOAP` calls the specified URL.
If the server responds with a SOAP message then the response is analyzed and used to generate a WTScript object in the same way as when a proxy method is executed.
If the server does not respond with a SOAP message but, for example, with an HTML document then the content of this document is returned.

```
executeGetRequest(url)
```

url URL containing the complete request

Exceptions

If you attempt to set an HTTP header that has already been set by `WT_SOAP` then the method outputs a `USAGE` exception.

4.2.4.5 analyseResponse method

The `analyseResponse` method analyses the response to the last request (i.e. the contents of `<WT_SOAP>.http.receiveData.Body`) and constructs a WTScript data structure from the result. You will need this method together with the `setRunMode("PREPARE")` method (this prepares a web service message but does not send it) and the `executeRequest` method (this sends a message) (see section “setRunMode method” on page 66).

```
analyseResponse()
```

Return value

WTScript data structure

Exceptions

The `analyseResponse` method does not output exceptions.

4.2.4.6 setSOAPVersion method

Since there is as yet no WSDL version offering SOAP V1.2 support that has been approved by W3C, web services continue to be described in WSDL version 1.1 files. As a result, `WT_SOAP` is not able to identify whether a message is to be generated for SOAP version 1.1. or 1.2 on the basis of the WSDL file.
The method `setSOAPVersion` defines the format in which messages are to be exchanged.

```
setSOAPVersion(<version>)
```

Return value

The set version

Parameter

<version>

The following values can be specified for *version*:

1.1	Generates messages in the old SOAP format and awaits responses in the old SOAP format. This value is set by default.
1.2	Generates messages in SOAP version 1.2 format and awaits responses in the same format.

If you specify a different value then the default value `1.1` is set.

Exceptions

The method `setSOAPVersion` does not output any exceptions.

Example

```
mySoap = new WT_SOAP('wsdl/test1.wsdl');  //Create WT_SOAP object
mySoap.setSOAPVersion("1.2");             //Set Version
```

4.2.4.7 addHeader method

The `addHeader` method sets the specified header block for the following messages. All the defined headers are recorded in the array `WT_SOAP.envelope.HeaderBlocks`. When a message is generated, XML is generated from the objects and entered in the header.

Any added headers remain valid for all subsequent proxy method calls. If they are no longer needed, you must remove them with `removeAllHeaders`.

```
addHeader(wtsoapHeader1[,wtsoapHeader2[,…]])
```

Return value

Index of the first added header.

Parameters

wtsoapHeader1,
wtsoapHeader2,…
Objects of the `WT_SOAP_HEADER` class that are to be added to the message.

4.2.4.8 removeAllHeaders method

The `removeAllHeaders` method deletes all the headers.

```
removeAllHeaders()
```

4.2.4.9 getHeaderObjects method

If the response to a SOAP message contains information in the SOAP header, you can, for example, use the `getHeaderObjects` method to access the headers.

The `getHeaderObjects` method returns an object array in which each object is an instance of the `WT_SOAP_HEADER` class and represents an element in the SOAP header.

```
getHeaderObjects()
```

Example

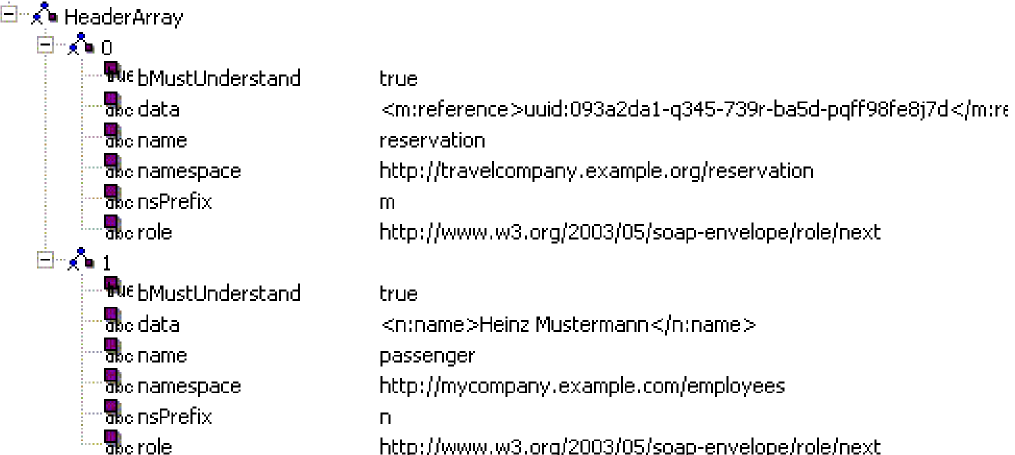
A response from a web service contains the following header:

```
<env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
          env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
          env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <n:name>John Smith</n:name>
  </n:passenger>
 </env:Header>
```

The call

```
myObjTree= mySoap.getHeaderObjects();
```

returns the following data structure



4.2.4.10 getHeaderObjectTree method

If the response to a SOAP message contains information in the SOAP header then you can, for example, use the `getHeaderObjectTree` method to access the headers.

The `getHeaderObjectTree` method returns a WTScript object which represents the entire header. The object is generated in the same way as a proxy method's response object.

```
getHeaderObjectTree()
```

Example

A response from a web service contains the following header:

```
<env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
          env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
          env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <n:name>John Smith</n:name>
  </n:passenger>
 </env:Header>
```

The call

```
myObjTree= mySoap.getHeaderObjectTree();
```

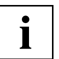
returns the following data structure:

4.2.4.11 createProxysWithPrefix method

Operations that provide a web service may possess both input parameters that have to be specified at call time as well as input parameters that are not essential. You can use the `createProxysWithPrefix` method to make this information evident in the names of the proxy objects and their attributes.

To preserve the structure of the proxy objects, this information is stored as a prefix in the attribute name. An `m_` in front of the name indicates a mandatory attribute while an `o_` indicates an optional attribute. This prefix plays no role in the generation of messages.

The `createProxysWithPrefix` method activates or deactivates this mode.



The method must be called before the WDSL is analyzed. For this reason, the constructor must first be called without parameters and the function `initFromWsdl()` must be used

---

`createProxysWithPrefix (`*bMode*`)`

Parameter

bMode Boolean value.

true	The proxy objects are generated with prefixes.
false	The proxy objects are generated without prefixes. Default setting.

Example

```
mySoap = new WT_SOAP();
mySoap.setProxy ('proxy.company.com','81');
mySoap.setProxyAuthorization('puser','ppass');
mySoap.createProxyWithPrefix(true);
mySoap.initFromWsdlUri('http://url');
```

- proxyObjects
  - m_clonePortlet_clonePortlet
    - m_portletContext
      - o_extensions
      - m_portletHandle
      - o_portletState
    - m_registrationContext
      - o_extensions
      - m_registrationHandle
      - o_registrationState
    - m_userContext
      - o_extensions
      - o_profile
      - o_userCategories
      - m_userContextKey

4.2.5 Methods for configuring access to the WT_SOAP_COM_FUNCTIONS subclass

The `WT_SOAP_COM_FUNCTIONS` class methods are immediately available under the `WT_SOAP` object and also available under each port. The following methods operate globally or current port-specific only; the type of operation used by the method depends on the object used to call the method.

4.2.5.1 setAuthorization method

If a web service is protected, use the `setAuthorization` method to set the user authorization for access to the web service. These values will be taken over by the attributes `USER` and `PASSWORD` of the HTTP host adapter and when the proxy method is called by the HTTP host adapter will be sent together with the proxy method (see section “Overview” on page 18).

```
setAuthorization(user, password)
```

Return value

No return value

Parameters

user Specifies the user to be transferred.

password
Specifies the password for the user *user*.

4.2.5.2 setProxy method

The `setProxy` method defines the proxy host and proxy port to be used when the WSDL document is loaded and for the actual SOAP request by the HTTP host adapter. `setProxy` therefore sets the `PROXY` and `PROXY_PORT` attributes for the host adapter in question.

```
setProxy(http_proxy_host, http_proxy_port)
```

Return value

No return value

Parameters

http_proxy_host
Specifies the value for the `PROXY` attribute.

http_proxy_port
Specifies the value for the `PROXY_PORT` attribute.

4.2.5.3 setProxyAuthorization method

The `setProxyAuthorization` method assigns a user authorization permitting access to the proxy host which the HTTP host adapter is to use to access the web service. `setProxyAuthorization` sets the attributes `PROXY_USER` and `PROXY_PASSWORD` of the HTTP host adapter.

```
setProxyAuthorization(http_proxy_user, http_proxy_password)
```

Return value

No return value

Parameters

http_proxy_user
Specifies the value for the `PROXY_USER` attribute.

http_proxy_password
Specifies the value for the `PROXY_PASSWORD` attribute.

4.2.5.4 setTimeout method

The `setTimeout` method specifies the timeout value for the host adapter by setting the `HTTP_TIMEOUT` attribute for the HTTP host adapter.

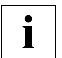
```
setTimeout (timeout)
```

Return value

No return value

Parameter

timeout Specifies the timeout value for the HTTP host adapter.



You should note that the attribute `TIMEOUT_HTTP` cannot be greater than `TIMEOUT_APPLICATION`. This means that you may also have to set the attribute `TIMEOUT_APPLICATION`.

4.2.6 Exceptions

In the event of errors that stop the SOAP service from responding, an exception occurs and the proxy method does not return a result. Such errors may be caused, for example, by parameter errors on the proxy method call or failed connections to the proxy or SOAP server.

Exception objects have the following structure:

`type` Either a SoapError if the error was recognized only by the WT_SOAP class or one of the other WebTransactions error types.

`soapCode`

Contains the SOAP error code. The following values are possible:

- `SOCKET`
  Error accessing the web service (access impossible).

  Possible causes:
  no network connection, incorrect `PROXY` server, incorrect `PROXY_PORT` etc.

- `HTTP`
  Error accessing the web service (e.g. access possible but `HTTP` signals a fault).

  Possible causes:
  `PROXY_AUTHORIZATION` is required; service cannot be found on the addressed system, etc.

- `FILE`
  The constructor was unable to find the WSDL file specified.
- `WSDL`
  The WSDL document contains elements that could not be interpreted correctly.
- `PARAMETER`
  The structure of the proxy method transfer parameters does not match that required by the WSDL.
- `PARSE`
  A syntax failure was found during execution.
- `WT_SOAP_HEADER`
  An error occurred while generating an element of class `WT_SOAP_HEADER`.
- `VERSION_MISMATCH`
  If you send a message in the wrong format to a web service that expects messages in SOAP V1.2 format then the server responds with a special error code. If `WT_SOAP` receives this type of response when executing a proxy method or the `ExecuteRequest` method then it outputs the exception `VERSION_MISMATCH` .This has the following meaning:
  A message was sent to a web service in the wrong format. If present, the versions of SOAP supported by the service are output for future changes.

`soapText`

Contains an explanatory text describing the error that has occurred.

The `soapCode` and `soapText` attributes only exist if:
`type == 'soapError'`

4.2.7 WT_SOAP attributes

The list below describes the attributes required for the `WT_SOAP` class. For more information on `WT_SOAP`, see section “Concept of SOAP integration in WebTransactions” on page 47.

`service`
: This is creates a `WT_SOAP` class object as an attribute. `service` contains an object for each service provided by the web service. This is used as an interface for the proxy method call.

`envelope`
: This is creates a `WT_SOAP` class object as an attribute. `envelope` contains the `envelope`, `header`, `HeaderBlocks` and `body` attributes which in turn contain the text segments used to construct the SOAP message when calling a proxy method or the `executeRequest` method.

  The `envelope` attribute in turn contains the following attributes:

  `envelope`
  : contains the `envelope` tag with all attributes. The attributes are supplied at instantiation. When a message is sent, the terminating tag is created at the end of the `body`.

  `header`
  : contains the content of the SOAP header; after instantiation the header is empty. `header` is bracketed by the matching `header` tags when the message is sent.

  `HeaderBlocks`
  : if you have used the `addHeader()` function to add header blocks then these headers are stored here and inserted in the header following the content of `header`.

  `body`
  : contains the contents of the SOAP body and is created at the proxy method call. When the message is sent, the matching `body` tag is generated.

`http`
: Contains a reference to the HTTP communication object used. The value is usually supplied by `WT_SOAP`. Developers can use `http` for special cases.

The list below gives the attributes which you can use to find out information about the WSDL:

`binding`
: Map of the `<binding>` section in the WSDL document.

`comattr`
: Contains the attributes controlling HTTP access (`http.WT_SYSTEM`). These attributes are stored here by `WT_SOAP_COM_FUNCTIONS` class methods (see section “Methods for configuring access to the WT_SOAP_COM_FUNCTIONS subclass” on page 75).

`message`
: Map of the `<message>` section of the WSDL document.

`portType`
: Map of the `<portType>` section of the WSDL document (for internal use). This contains one or more WSDL objects that mirror operations described in the WSDL `<portType>` section.

`types`
: This is used for the storage of complex data types that are described in the WSDL document.

4.2.8 Data types for the SOAP request in SOAP body

The SOAP data types are mapped to WTScript data types as specified in the table below. The data types are from the XSD schema.

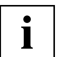
Complex data types

SOAP data type	WTScript data type
– SOAP data type `array` – Element from the `<types>` section of the WSDL with the attribute `maxOccurs > 1`.	Array
All data types which in the `<types>` section are defined with the `complexType` element.	Object

Simple data types

SOAP data type	WTScript data type
anyURI	string
base64Binary	string
binary	not currently supported
boolean	boolean
byte	number
date	string
dateTime	string
decimal	number
double	number
duration	string
ENTITIES	not currently supported
ENTITY	string
float	number
gDay	string
gMonth	string
gMonthDay	string
gYear	string
gYearMonth	string
gYearMonth	string
hexBinary	string
ID	not currently supported
IDREF	not currently supported
IDREFS	not currently supported
int	number
integer	number
language	string
long	number
Name	string
NCName	string
negativeInteger	number
NMTOKEN	string
NMTOKENS	string

SOAP data type	WTScript data type
nonNegativeInteger	number
nonPositiveInteger	number
normalizedstring	string
NOTATION	not currently supported
positiveInteger	number
QName	string
recurringDuration	string
short	number
string	string
time	string
timeDuration	string
token	string
unsignedByte	number
unsignedInt	number
unsignedLong	number
unsignedShort	number



All data is converted to string form when the message is sent using the `toString` method.

The conversion of these values is not checked. For example, no check is performed to determine whether integer values are passed to the proxy method in the case of operations that require the SOAP decimal data type.

4.2.9 Example: Checking the spelling of a text

In the following WTScript, use the WSDL shown in section “Example: WSDL document” on page 58. Call the web service which is used to spell check texts. The result is shown in a table.

```
<wtinclude name="wtSOAP">
<wtoncreatescript>
<!--
  try {

    mySoap = new WT_SOAP("wsdl/check.wsdl",
     'proxy.my_company.net','81');

    obj= new Object();
    obj.BodyText = "This is the latest speling chek";
    obj.LicenseKey = 0;
    myResult = mySoap.service.check.port.checkSoap.operation.CheckTextBody(obj);
      }
    catch( e )
  {
    document.write( '<P>Exception: ', e);
exitTemplate();
  }
//-->
</wtoncreatescript>

<wtrem>Evaluating result</wtrem>
<h3>Original Text</h3>
##myResult.CheckTextBodyResponse.DocumentSummary.body#
<h3>Misspelled Words</h3>
<table border="1">
<tr>
<th>word</th> <th> Possible corrections</th>
</tr>
<wtoncreatescript>
<!--
 for (i=0;i<myResult.CheckTextBodyResponse.DocumentSummary.MisspelledWord.length;i++)
 {
     document.write("<tr><td>");
     document.write
(myResult.CheckTextBodyResponse.DocumentSummary.MisspelledWord[i].word);
     document.write("</td><td>");

document.write(myResult.CheckTextBodyResponse.DocumentSummary.MisspelledWord[i].Suggest
ions.toString().slice(1,-1).replace(/,/g," ").replace(/"/g,""));
     document.write("</td></tr>");
 }
//-->
</wtoncreatescript>
</table>
```

4.3 WT_SOAP_HEADER - class for support of SOAP headers

In SOAP version 1.2, headers are of greater importance because of the so-called SOAP processing model. The main idea behind this model is that multiple hosts may be involved in the execution of a web service and that these may play different roles: The body of a message is intended for the last host (ultimateReceiver) in the chain. In contrast, the headers are used to send information to all the hosts, including the intermediate hosts. This mechanism can therefore be used, for example, to perform transactions involving web services or to implement security measures.

For a detailed description of how the headers are to be handled by the individual hosts and the role played by the individual header attributes, see the SOAP Version 1.2 documentation, section 2 “SOAP Processing Model”.

The `WT_SOAP_HEADER` class is used to generate these headers in the correct form.

It represents an XML element within the `SOAP` header block.

Objects in the `WT_SOAP_HEADER` class possess the attributes `name`, `data`, `role`, `namespace`, `nsPrefix` and `bRelay`. You can subsequently modify the individual properties of this class by accessing these elements directly.

4.3.1 Constructor of the WT_SOAP_HEADER class

The `WT_SOAP_HEADER` class has only one constructor:

```
WT_SOAP_HEADER (name,data[,encodingStyle[,bMustUnderstand[,
                role[,namespace[,nsPrefix[,bRelay]]]]]])
```

Parameters

name Name of the header element. You must specify a valid XML tag name here. A valid XML tag name starts with a letter or an underscore. This may be followed by digits, letters, underscores, hyphens or periods.

data Content of the header element. Specify a well-formed XML text.

encodingStyle

Here you can specify the value of the `encodingStyle` attribute for a SOAP header block. You can specify any URI that indicates how the header element is coded: The following constants for values defined in SOAP are possible:

SOAP1.1	Coding as in SOAP 1.1
SOAP1.2	Coding as in SOAP1.2
none	No indication of coding.

If this parameter is missing or an empty string is specified then this attribute is ignored when the message is generated

bMustUnderstand

may have the following values:

true	The host that was addressed with `role` must understand the header. If you do not specify `role` or enter an empty string for `role` then the attribute is omitted on message generation. The omission of the attribute has the same effect as setting it to `ultimateReceiver`.
false	It is not essential for the header to be understood. Default setting

role

SOAP V1.2 defines roles for the host for which the header is intended. The role is specified as a string in the form of an URL. If you do not specify `role` or enter an empty string for `role` then the attribute is omitted.
You can specify keywords for the roles that are predefined in SOAP V1.2:

none	The header may not be processed by any host that is a SOAP node. Such hosts may only view the header.
next	The header must be processed by the next host.
ultimate Receiver	The header must be processed by the host that executes the web service. Default setting
anyUri	If the host possesses a role defined by the web service, enter the URI of this role here.

namespace

You enter a header's namespace in this parameter.

nsPrefix

Prefix assigned to the header element. If you do not specify anything here or if the parameter is omitted then a prefix is generated internally. Prefixes used internally by WT_SOAP start with `wt_`. Do **not** specify any such prefixes here.

A valid prefix for an XML tag starts with a letter or an underscore. This can be followed by numbers, letters, underscores, hyphens and dots.

bRelay

Here you can specify `true` or `false` depending on whether or not a header is to be passed on if the intermediate host (in accordance with the SOAP processing model) for which the header was intended is unable to process it.

Examples

Example 1 below generates a header that contains a transaction number:

```
myHead1= new WT_SOAP_HEADER("transaction ",
                "5",
                "http://example.com/encoding ",
                true,
                "",
                "http://thirdparty.example.org/transaction");
```

The defined header is taken over into the message as follows:

```
<SOAP-ENV:Header>
  <wt_tns0:transaction
      SOAP-ENV:mustUnderstand="true"
      SOAP-ENV:encodingStyle="http://example.com/encoding"
      xmlns:wt_tns0="http://thirdparty.example.org/transaction">
          5
  </wt_tns0:transaction >
</SOAP-ENV:Header><env:Header>
   <wt_tns0:transaction
           xmlns:wt_hns0="http://thirdparty.example.org/transaction"
           env:encodingStyle="http://example.com/encoding"
           env:mustUnderstand="true" >5</<wt_tns0:transaction>
 </env:Header>
```

Example 2 below generates a header that contains elements with reservation numbers and dates. Since these elements are also specified qualified with the same namespace as the header element itself, you must also specify the prefix that is to be used:

```
myHead1=new WT_SOAP_HEADER("transactionreservation",
    '<m:reference>uuid:q345-739r-ba5d-pqff98fe8j7d</m:reference>
     <m:dateAndTime>2001-11-29T13:36:50.000-05:00</m:dateAndTime>',
"",
     true,
     "Next",
     "http://travelcompany.example.org/reservation",
     "m");
```

The defined header is taken over into the message as follows:

```
<SOAP-ENV:Header>
  <m:reservation
    SOAP-ENV:mustUnderstand="true"
    SOAP-ENV:role="Next"
    xmlns:m="http://travelcompany.example.org/reservation">
     <m:reference>uuid:q345-739r-ba5d-pqff98fe8j7d</m:reference>
     <m:dateAndTime>2001-11-29T13:36:50.000-05:00</m:dateAndTime>
  </m:reservation>
</SOAP-ENV:Header><env:Header>
   <m:reservation
     xmlns:m="http://travelcompany.example.org/reservation"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
    <m:reference>uuid:q345-739r-ba5d-pqff98fe8j7d</m:reference>
    <m:dateAndTime>2001-11-29T13:36:50.000-05:00</m:dateAndTime>
  </m:reservation>
 </env:Header>
```

Exceptions

The constructor outputs an exception if both `bMustUnderstand` and `bRelay` have the value `true` since these two values are mutually exclusive as specified.

5 Examples

This chapter provides an overview of the options provided by WebTransactions for accessing an HTTP server.

5.1 Using an existing CGI script

This example shows how to use an HTTP `POST` method call (send data to an HTTP server) to call a CGI script. The CGI script (`www.deepThought.mt/cgi-bin/verify.exe`) expects two parameters: a question (`question`) and an answer (`answer`) which must be analyzed.

For the sake of simplicity, the caller provides an object `sendFORM` containing the required parameters in the form of attributes, e.g. `sendFORM.question = 'sense+of+universe'`.

The send function `sendForm` creates an object `sendData` with the appropriate attributes `ContentType` and `Body` of the required HTTP message. For each attribute of `sendFORM`, the name and value are appended to the `Body` attribute as a name/value pair:

```
function sendForm()
{
  this.sendData = new Object();
  this.sendData.ContentType = 'application/x-www-form-urlencoded';
  this.sendData.Body = '';
  for( attr in this.sendFORM )
  {
    this.sendData.Body += ( this.sendData.Body ? '&' : '' ) +
                          attr + '=' + this.sendFORM [attr];
  }
  return this.send();
}
```

The universal receive function `receiveForm` interprets the body of the received HTTP message and defines a pattern for searching for HTML input fields (in the format `<INPUT TYPE="TEXT" NAME="`*fieldname*`" VALUE="`*fieldvalue*`">`). The name and value are enclosed in brackets. Based on this pattern, the system searches the `Body` attribute of the `receiveData` object. For each input field found, an attribute of the `receiveFORM` object is created with the name of the input field and the received value.

```
function receiveForm()
{
  if( ! this.receive() )
    return null;
  this.receiveFORM = new Object();
  fieldPattern =
    /<INPUT.*+TYPE=.*TEXT.+NAME=\W*(\w+).+VALUE=["'](\w*)["'].*>/ig
  while( field = fieldPattern.exec( this.receiveData.Body ) )
  {
    this.receiveFORM[field[1]] = field[2];
  }
  return this;
}
```

The communication object is generated:

```
host = new WT_Communication('myHTTP');
host.open('HTTP');
```

These methods are now inserted in our communication object.

```
host.sendForm = sendForm;
host.receiveForm = receiveForm;
```

The HTTP request can thus be formulated as follows:

```
host.sendFORM = new Object();
host.sendFORM.question = 'sense+of+universe';
host.sendFORM.answer = 42;
host.WT_SYSTEM.URL = www.deepThought.mt/cgi-bin/verify.exe;
host.sendForm();
host.receiveForm();
...
```

Processing can now continue as required with the attributes received by the `receiveForm` call and stored in the `receiveFORM` object (input fields of the answer).

5.2 Using information from the Web

This example shows how to use the HTTP host adapter to retrieve up-to-date information from the WWW for use in the WebTransactions application. In each session, the following function retrieves the current exchange rate for the US dollar and the Japanese Yen from a web site, and stores this information in the system object attributes `WT_SYSTEM.dollar_factor` and `WT_SYSTEM.yen_factor`.

```
<wtoncreatescript>
    <!--
    /* Function for retrieving the dollar and Yen rates */
    function getEuroToDollarYen ()
    {
        var i,j;
        // To be executed only once per session
        if (!WT_SYSTEM.dollar_factor)
        {
            // If the communication object does not yet exist, it is created
            if (!WT_HOST.HTTPCON)
                db = new WT_Communication("HTTPCON");
            else
                db = WT_HOST.HTTPCON;
            // Prepare the HTTP connection
            db.open ("HTTP");
            db.WT_SYSTEM.URL =
                "http://waehrungen.onvista.de/devisenkurse.html";
            db.WT_SYSTEM.PROXY = "proxy.my_company.net";
            db.WT_SYSTEM.PROXY_PORT = "81";
            db.WT_SYSTEM.TIMEOUT_HTTP = "10";
            // Send the HTTP request and analyze the result
            if (db.receive())
            {
                // Search for the dollar rate
                var cellBegin = '<td align="right">';
                i = db.receiveData.Body.indexOf('Euro-US Dollar');
                //The current dollar rate is located in the 3rd table column
                for (z=0; z<3;z++)
                  i = db.receiveData.Body.indexOf(cellBegin,i) +
cellBegin.length;
                j = db.receiveData.Body.indexOf(' ',i);
                help=db.receiveData.Body.substring(i,j);
                // Save the dollar rate in numeric format
                WT_SYSTEM.dollar_factor =
                    db.receiveData.Body.substring(i,j).replace(/,/,".") * 1;
                if (WT_SYSTEM.dollar_factor == NaN)
                    delete WT_SYSTEM.dollar_factor;
                // And while we’re here, search for the Yen rate
```

```
                i = db.receiveData.Body.indexOf('Euro-Japanischer Yen');
                for (z=0; z<3;z++)
                  i = db.receiveData.Body.indexOf(cellBegin,i) +
cellBegin.length;
                j = db.receiveData.Body.indexOf(' ',i);
                WT_SYSTEM.yen_factor =
                    db.receiveData.Body.substring(i,j).replace(/,/,".") * 1;
                if (WT_SYSTEM.yen_factor == NaN)
                    delete WT_SYSTEM.yen_factor;
            }
                // Conclude the procedure
                db.close();
        }
    }
    getEuroToDollarYen();
    //-->
</wtoncreatescript>
##WT_SYSTEM.dollar_factor#
<br>
##WT_SYSTEM.yen_factor#
```

5.3 Communicating via HTTP and processing with WT_Filter

This section describes a WebTransactions application for an HTTP host adapter taking the WebTransactions client API `WT_RPC` for `WT_REMOTE` supplied with the product as an example. For detailed information on using the `WT_RPC` class, please refer to the WebTransactions manual “Client APIs for WebTransactions”, where you will also find an example of how to use the class. This section concentrates on the definition of the `WT_RPC` class as an example of how to communicate via HTTP and process the messages received using the filters supplied with the product.

5.3.1 Basic concept of the WT_RPC class

The HTTP host adapter allows you to integrate any information provided by web servers into WebTransactions. In this context, the WebTransactions application acts as a client of a Web application. At the same time, it also functions as a server providing information on the basis of the HTTP protocol. In addition to the HTML interface, WebTransactions offers a second server interface, `WT_REMOTE`. This interface enables clients to control a WebTransactions application remotely.

As basic WebTransactions mechanisms, the `WT_REMOTE` interface and the HTTP host adapter allow the distribution of applications via the network. It is possible for one WebTransactions application to delegate certain tasks to another WebTransactions application. The basic mechanisms allow the client to define method calls in the form of XML documents, for example, and send them to the remote WebTransactions application. The application, in turn, returns the result in an XML document, which can be converted back into the corresponding data objects in the client application. The `WT_RPC` class is used for handling the creation of the method call in the client, the submission of the remote call, and the formatting of the result.

The `WT_RPC` class supports the remote invocation of WebTransactions functions. The address (URL and base directory for WebTransactions) of the remote WebTransactions application and the names of the remote functions are transferred to a `WT_RPC` object. The remote functions can then be called as methods of the `WT_RPC` object.

The figure below shows an example of this. The client requires the document `WT_RPC.htm`, which contains the definition of the `WT_RPC` class. This class is used to define a connection and make known the remote function `f` in the document `calc.htm`. The local method `rtw.f` can now be called as a proxy for the remote function `f`.

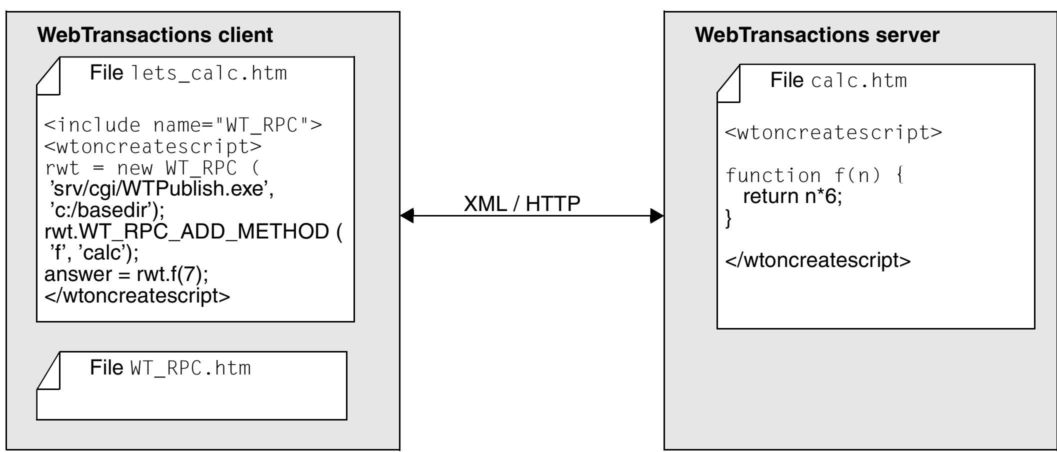


Figure 5: Functionality of the `WT_RPC` class

5.3.2 Implementation of the WT_RPC class

The following sections describe the implementation of the constructor and methods of the `WT_RPC` class as supplied in WebTransactions. For an example of how to use this class, please refer to the WebTransactions manual “Client APIs for WebTransactions”.

The WT_RPC constructor

The `WT_RPC` constructor creates a new communication object with the name `WT_RPC_n`, where `n=0,1,2...` and represents the smallest index for which a corresponding object does not yet exist. This communication object is used to manage the HTTP connection to the remote WebTransactions application. It contains the methods `WT_RPC_OPEN`, `WT_RPC_CLOSE`, `WT_RPC_INVOKE`, and `WT_RPC_ADD_METHOD`, which are described below. Any connection data specified for the remote WebTransactions application is stored in the `WT_URL` and `WT_BASEDIR` attributes, and the remote WebTransactions application is started:

```
// constructor for the remote procedure object //////////////
function WT_RPC( urlOfWebTA, basedir )
{
  // create new communication object WT_RPC_n
  var i;
  for ( i = 0; WT_HOST[ 'WT_RPC_' + i ] != null; i++ );
  this.WT_COM_OBJECT = new WT_Communication( 'WT_RPC_' + i );
  // define methods
  this.WT_RPC_OPEN       = WT_RPC_OPEN;
  this.WT_RPC_CLOSE      = WT_RPC_CLOSE;
  this.WT_RPC_INVOKE     = WT_RPC_INVOKE;
  this.WT_RPC_ADD_METHOD = WT_RPC_ADD_METHOD;
  // start remote WebTransactions application
  if( urlOfWebTA && basedir )
    this.WT_RPC_OPEN( urlOfWebTA, basedir );
  else
    this.WT_CONNECTED = false;
}
```

WT_RPC_OPEN method - start a remote WebTransactions application

This method starts a remote WebTransactions session and prepares the communication object for method calls. If all actions are performed successfully, the function returns with `true`. Otherwise, it returns with `false`.

The following actions are carried out.

- Any active remote WebTransactions application is terminated and the connection is closed.
- If parameters are specified, these are transferred to the `WT_URL` and `WT_BASEDIR` attributes.
- If an address for the remote WebTransactions application has not been made available (by transferring parameters from this or an earlier call), the method is terminated with a negative return code.
- Otherwise, the HTTP host adapter is activated for the communication object by calling the `com.open` method.
- The `URL` attribute is constructed to start the remote WebTransactions application.
- The `WT_REMOTE` action `START_SESSION` requests the remote WebTransactions application to output its session parameters.
- The `sendData` object is deleted in order to ensure that the `GET` method is executed.
- Using `send` and `receive`, access takes place via the network.
- If this action is performed successfully (`HTTP_RETURN_CODE` is set to 200 and XML data is received), the `sendData` object can be prepared for the subsequent method calls.

  This should involve a multisection HTTP message.

  – The first section contains the session data used by WebTransactions to locate the desired remote session. This includes the MIME type `application/x-www-form-urlencoded`. The required values can be obtained by analyzing the response from the preceding call using `WT_Filter`.

  – The second section contains the actual usable information. This is encoded in the form of an XML document and thus has the MIME type `text/xml`.

```
// method to start new remote session and connect to it //////////////
function WT_RPC_OPEN( urlOfWebTA, basedir )
{
  // terminate existing remote WebTransactions application
  if( this.WT_CONNECTED )
    this.WT_RPC_CLOSE();

  // store new connection definition if redefined
```

```
    if( urlOfWebTA && basedir )
    {
      this.WT_URL = urlOfWebTA;
      this.WT_BASEDIR = basedir;
    }

    // return immediately if connection parameters missing
    if( ! ( this.WT_URL && this.WT_BASEDIR ) )
      return this.WT_CONNECTED = false;
    // start remote session
    var com = this.WT_COM_OBJECT;
    com.open( 'HTTP' );
    com.WT_SYSTEM.URL =
      this.WT_URL + '/startup?'+
      'WT_SYSTEM_BASEDIR=' + this.WT_BASEDIR +
      '&WT_REMOTE=START_SESSION';
    if( com.sendData )
      delete com.sendData;
    com.send();
    com.receive();
    if( !WT_SYSTEM.ERROR && com.WT_SYSTEM.HTTP_RETURN_CODE == 200
        && com.ReceiveDate.ContentType='text/xml' )
    {
      //store session parameters
      var remoteWtSystem;
      WT_Filter.XMLToDataObject(com.receiveData.Body,'remoteWtSystem');
      this.WT_SESSION_PARAMS ='WT_SYSTEM_BASEDIR=' + this.WT_BASEDIR +
       '&WT_SYSTEM_FORMAT_STATE=IGNORE'+
       '&WT_SYSTEM_SESSION=' + remoteWtSystem.SESSION +
       '&WT_SYSTEM_SIGNATURE=' + remoteWtSystem.SIGNATURE;
      // prepare sendData
      com.sendData = new Array();
      com.sendData[0] = new Object();
      com.sendData[0].ContentType =
        'application/x-www-form-urlencoded';
      com.sendData [0].Body =
        this.WT_SESSION_PARAMS + '&WT_REMOTE=PROCESS_COMMANDS';
      com.sendData[1] = new Object();
      com.sendData[1].ContentType = 'text/xml';
      // prepare communication object for method invocation
      com.WT_SYSTEM.URL = this.WT_URL;
      this.WT_CONNECTED = true;
    }
    else
      this.WT_CONNECTED = false;
    return this.WT_CONNECTED;
  }
```

WT_RPC_CLOSE method - terminate a remote WebTransactions application

This method terminates a remote WebTransactions application and deactivates the HTTP communication module. The following actions are carried out:

- The `URL` attribute is constructed for accessing the remote WebTransactions application.
- The `WT_REMOTE` action `EXIT_SESSION` terminates the remote WebTransactions application.
- The `sendData` object is deleted in order to ensure that the `GET` method is executed.
- Using `send` and `receive`, access takes place via the network.
- Finally, the `close` method is called to release the resources of the HTTP communication module.

```
// terminate remote session and close connection ///////////
function WT_RPC_CLOSE()
{
  if( this.WT_CONNECTED )
  {
    var com = this.WT_COM_OBJECT;
    com.WT_SYSTEM.URL = this.WT_URL + '?' +
                        this.WT_SESSION_PARAMS +
                        '&WT_REMOTE=EXIT_SESSION';
    if( com.sendData )
      delete com.sendData;
    com.send();
    com.receive();
    com.close();
    this.WT_CONNECTED = false;
  }
}
```

WT_RPC_INVOKE method - call a remote function

This method calls a remote function with the name *functionName*. It involves the creation of an appropriate XML document using the `methodCallToXML` filter. In `functionName` the filter receives the name of the method, in `codeBase` the name of the WTML document to be implemented on the remote machine, and in `argArray` an array of the desired parameters for the method call. This is sent to the remote WebTransactions application and executed. The result is an XML document, which is then converted into a WTML script data object using the `XMLToDataObject` filter and returned. If there is no connection to a remote WebTransactions application, the return value is `null`.

```
// prepare XML document for method call and send it to remote session
function WT_RPC_INVOKE( functionName, codeBase, argArray )
{
  if( this.WT_CONNECTED )
  {
    var returnValue;
    var com = this.WT_COM_OBJECT;
    com.sendData[1].Body =
      WT_Filter.methodCallToXML( functionName, argArray, codeBase );
    com.send();
    com.receive();
    WT_Filter.XMLToDataObject( com.receiveData.Body, 'returnValue' );
    return returnValue;
  }
  else
    return null;
}
```

WT_RPC_ADD_METHOD method - define a remote function

As well as calling remote functions using the `WT_RPC_INVOKE` method in which the name, the WTML document to be implemented, and a parameter array are specified as arguments, it must also be possible to invoke remote functions directly by means of a local method of the `WT_RPC` object with the same name. Such proxy methods can be created using the `WT_RPC_ADD_METHOD` method. `WT_RPC_ADD_METHOD` creates a new function which calls `WT_RPC_INVOKE` with the appropriate function and document name. The parameters of this new function are forwarded to `WT_RPC_INVOKE` in an array. The new function object is stored in the `WT_RPC` object with the name *functionName*.

```
// bind function to WT_RPC object //////////////////
function WT_RPC_ADD_METHOD( functionName, codeBase )
{
  this[functionName] = new Function(
   '{return this.WT_RPC_INVOKE ("' + functionName + '", "' +
   codeBase + '", arguments );}' );
}
```

6 Appendix

The appendix provides

- a brief overview of the HTTP error messages as defined in RFC 2068
- WSDL schema

6.1 HTTP error messages

This chapter lists the HTTP error messages that may occur when connecting to an HTTP server.

Overview as defined in RFC 2068

The table below provides a brief overview of the HTTP error codes in HTTP/ 1.1, their names in RFC 2086, and their meaning:

Code / name	Meaning
`1xx`	Informative response codes (only in HTTP/ 1.1)
`200 OK`	Request successful
`201 Created`	Request executed, new contents generated
`202 Accepted`	Request accepted but not yet completed
`203 Non-Authoritative Information`	Metadata modified by third party
`204 No Content`	Successful but no contents
`205 Reset Content`	Contents modified successfully
`206 Partial Content`	GET operation partially successful
`300 Multiple Choices`	Localized versions available
`301 Moved Permanently`	Moved
`302 Moved Temporarily`	Moved temporarily
`303 See Other`	Available under another URI

Table 6: HTTP error codes as defined in RFC 2068

Code / name	**Meaning**
304 Not Modified	Conditional access successful but resources not modified
305 Use Proxy	Access only via proxy
400 Bad Request	Request in invalid syntax
401 Unauthorized	Authentication required
402 Payment Required	Reserved for future extensions
403 Forbidden	Access denied
404 Not Found	Requested resource not found
405 Method Not Allowed	Requested method not permitted for specified resource
406 Not Acceptable	Response entities do not match request
407 Proxy Authentication Required	Authentication via proxy required
408 Request Timeout	Timeout for client request
409 Conflict	Conflict in resources status
410 Gone	Resource not available
411 Length Required	Content length must be specified in the header
412 Precondition Failed	Condition in request header not fulfilled
413 Request Entity Too Large	Request entity too long
414 Request-URI Too Long	Request URI longer than permitted by server
415 Unsupported Media Type	Invalid request format for required method
500 Internal Server Error	Unexpected HTTP server error
501 Not Implemented	Required functionality not available
502 Bad Gateway	Invalid response from superordinate server
503 Service Unavailable	Server cannot process request at the moment
504 Gateway Timeout	Timeout in response from superordinate server
505 HTTP Version Not Supported	Unsupported HTTP version of request

Table 6: HTTP error codes as defined in RFC 2068

6.2 WSDL Schema

The following WSDL schema shows the specification for WSDL 1.1 as provided by the World Wide Web Consortium (W3C). Sections of the schema which are not supported by the WT_SOAP class are printed in blue.

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        targetNamespace="http://schemas.xmlsoap.org/wsdl/"
        elementFormDefault="qualified">
   <element name="documentation">

   <complexType mixed="true">
         <choice minOccurs="0" maxOccurs="unbounded">
            <any minOccurs="0" maxOccurs="unbounded"/>
         </choice>
         <anyAttribute/>
      </complexType>
   </element>
   <complexType name="documented" abstract="true">
      <sequence>
         <element ref="wsdl:documentation" minOccurs="0"/>
      </sequence>
   </complexType>
   <complexType name="openAtts" abstract="true">
      <annotation>
         <documentation>
         This type is extended by  component types
         to allow attributes from other namespaces to be added.
         </documentation>
      </annotation>
      <sequence>
         <element ref="wsdl:documentation" minOccurs="0"/>
      </sequence>
      <anyAttribute namespace="##other"/>
   </complexType>
   <element name="definitions" type="wsdl:definitionsType">
      <key name="message">
         <selector xpath="message"/>
         <field xpath="@name"/>
      </key>
      <key name="portType">
         <selector xpath="portType"/>
         <field xpath="@name"/>
      </key>
      <key name="binding">
         <selector xpath="binding"/>
         <field xpath="@name"/>
      </key>
      <key name="service">
         <selector xpath="service"/>
         <field xpath="@name"/>
```

```
        </key>
       <key name="import">
              <selector xpath="import"/>
              <field xpath="@namespace"/>
        </key>
        <key name="port">
           <selector xpath="service/port"/>
           <field xpath="@name"/>
        </key>
     </element>
  <complexType name="definitionsType">
        <complexContent>
           <extension base="wsdl:documented">
              <sequence>
                 <element ref="wsdl:import" minOccurs="0" maxOccurs="unbounded"/>
                 <element ref="wsdl:types" minOccurs="0"/>
                 <element ref="wsdl:message" minOccurs="0" maxOccurs="unbounded"/>
                 <element ref="wsdl:portType" minOccurs="0" maxOccurs="unbounded"/>
                 <element ref="wsdl:binding" minOccurs="0" maxOccurs="unbounded"/>
                 <element ref="wsdl:service" minOccurs="0" maxOccurs="unbounded"/>
                 <any namespace="##other" minOccurs="0" maxOccurs="unbounded">
                    <annotation>
                       <documentation>to support extensibility elements </documentation>
                    </annotation>
                 </any>
              </sequence>
              <attribute name="targetNamespace" type="uriReference" use="optional"/>
              <attribute name="name" type="NMTOKEN" use="optional"/>
           </extension>
        </complexContent>
    </complexType>
     <element name="import" type="wsdl:importType"/>
     <complexType name="importType">
        <complexContent>
     <extension base="wsdl:documented">
     <attribute name="namespace" type="uriReference" use="required"/>
        <attribute name="location" type="uriReference" use="required"/>
     </extension>
    </complexContent>
    </complexType>
     <element name="types" type="wsdl:typesType"/>
     <complexType name="typesType">
        <complexContent>
     <extension base="wsdl:documented">
     <sequence>
     <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
     </extension>
    </complexContent>
    </complexType>
     <element name="message" type="wsdl:messageType">
        <unique name="part">
```

```
        <selector xpath="part"/>
        <field xpath="@name"/>
     </unique>
  </element>
  <complexType name="messageType">
     <complexContent>
  <extension base="wsdl:documented">
  <sequence>
  <element ref="wsdl:part" minOccurs="0" maxOccurs="unbounded"/>
 </sequence>
     <attribute name="name" type="NCName" use="required"/>
  </extension>
 </complexContent>
 </complexType>
  <element name="part" type="wsdl:partType"/>
  <complexType name="partType">
     <complexContent>
  <extension base="wsdl:openAtts">
  <attribute name="name" type="NMTOKEN" use="optional"/>
     <attribute name="type" type="QName" use="optional"/>
     <attribute name="element" type="QName" use="optional"/>
  </extension>
 </complexContent>
 </complexType>
  <element name="portType" type="wsdl:portTypeType"/>
  <complexType name="portTypeType">
     <complexContent>
  <extension base="wsdl:documented">
  <sequence>
  <element ref="wsdl:operation" minOccurs="0" maxOccurs="unbounded"/>
 </sequence>
     <attribute name="name" type="NCName" use="required"/>
  </extension>
 </complexContent>
 </complexType>
  <element name="operation" type="wsdl:operationType"/>
  <complexType name="operationType">
     <complexContent>
  <extension base="wsdl:documented">
     <choice>
        <group ref="wsdl:one-way-operation"/>
        <group ref="wsdl:request-response-operation"/>
        <group ref="wsdl:solicit-response-operation"/>
        <group ref="wsdl:notification-operation"/>
     </choice>
     <attribute name="name" type="NCName" use="required"/>
  </extension>
 </complexContent>
 </complexType>
  <group name="one-way-operation">
     <sequence>
        <element ref="wsdl:input"/>
```

```
      </sequence>
   </group>
   <group name="request-response-operation">
      <sequence>
         <element ref="wsdl:input"/>
         <element ref="wsdl:output"/>

         <element ref="wsdl:fault" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
   </group>
   <group name="solicit-response-operation">
      <sequence>
         <element ref="wsdl:output"/>
         <element ref="wsdl:input"/>
         <element ref="wsdl:fault" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
   </group>
   <group name="notification-operation">
      <sequence>
         <element ref="wsdl:output"/>
      </sequence>
   </group>
   <element name="input" type="wsdl:paramType"/>
   <element name="output" type="wsdl:paramType"/>
   <element name="fault" type="wsdl:faultType"/>
   <complexType name="paramType">
      <complexContent>
   <extension base="wsdl:documented">
   <attribute name="name" type="NMTOKEN" use="optional"/>
      <attribute name="message" type="QName" use="required"/>
   </extension>
  </complexContent>
  </complexType>
   <complexType name="faultType">
      <complexContent>
   <extension base="wsdl:documented">
   <attribute name="name" type="NMTOKEN" use="required"/>
      <attribute name="message" type="QName" use="required"/>
   </extension>
  </complexContent>
  </complexType>
   <complexType name="startWithExtensionsType" abstract="true">
      <complexContent>
   <extension base="wsdl:documented">
   <sequence>
   <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
   </extension>
  </complexContent>
  </complexType>
   <element name="binding" type="wsdl:bindingType"/>
   <complexType name="bindingType">
      <complexContent>
```

```
      <extension base="wsdl:startWithExtensionsType">
      <sequence>
      <element name="operation" type="wsdl:binding_operationType" minOccurs="0"
                                                                    maxOccurs="unbounded"
   />
     </sequence>
         <attribute name="name" type="NCName" use="required"/>
         <attribute name="type" type="QName" use="required"/>
      </extension>
     </complexContent>
     </complexType>
      <complexType name="binding_operationType">
         <complexContent>
      <extension base="wsdl:startWithExtensionsType">
      <sequence>
      <element name="input" type="wsdl:startWithExtensionsType" minOccurs="0"/>
         <element name="output" type="wsdl:startWithExtensionsType" minOccurs="0"/>
         <element name="fault" minOccurs="0" maxOccurs="unbounded">
            <complexType>
               <complexContent>
      <extension base="wsdl:startWithExtensionsType">
      <attribute name="name" type="NMTOKEN" use="required"/>
            </extension>
     </complexContent>
     </complexType>
         </element>
     </sequence>
         <attribute name="name" type="NCName" use="required"/>
      </extension>
     </complexContent>
     </complexType>
      <element name="service" type="wsdl:serviceType"/>
      <complexType name="serviceType">
         <complexContent>
      <extension base="wsdl:documented">
      <sequence>
      <element ref="wsdl:port" minOccurs="0" maxOccurs="unbounded"/>
         <any namespace="##other" minOccurs="0"/>
     </sequence>
         <attribute name="name" type="NCName" use="required"/>
      </extension>
     </complexContent>
     </complexType>
      <element name="port" type="wsdl:portType"/>
      <complexType name="portType">
         <complexContent>
      <extension base="wsdl:documented">
      <sequence>
      <any namespace="##other" minOccurs="0"/>
     </sequence>
         <attribute name="name" type="NCName" use="required"/>
         <attribute name="binding" type="QName" use="required"/>
```

```
      </extension>
    </complexContent>
    </complexType>
    <attribute name="arrayType" type="string"/>
</schema>
```

Glossary

A term in ->*italic* font means that it is explained somewhere else in the glossary.

active dialog

In the case of active dialogs, WebTransactions actively intervenes in the control of the dialog sequence, i.e. the next ->*template* to be processed is determined by the template programming. You can use the ->*WTML* language tools, for example, to combine multiple ->*host formats* in a single ->*HTML* page. In this case, when a host ->*dialog step* is terminated, no output is sent to the ->*browser* and the next step is immediately started. Equally, multiple interactions between the Web ->*browser* and WebTransactions are possible within **one and the same** host dialog step.

array

->*Data type* which can contain a finite set of values of one data type. This data type can be:
- ->*scalar*
- a ->*class*
- an array

The values in the array are addressed via a numerical index, starting at 0.

asynchronous message

In WebTransactions, an asynchronous message is one sent to the terminal without having been explicitly requested by the user, i.e. without the user having pressed a key or clicked on an interface element.

attribute

Attributes define the properties of ->*objects*.
An attribute can be, for example, the color, size or position of an object or it can itself be an object. Attributes are also interpreted as ->*variables* and their values can be queried or modified.

Automask template

A WebTransactions *->template* created by WebLab either implicitly when generating a base directory or explicitly with the command **Generate Automask**. It is used whenever no format-specific template can be identified. An Automask template contains the statements required for dynamically mapping formats and for communication. Different variants of the Automask template can be generated and selected using the system object attribute `AUTOMASK`.

base directory

The base directory is located on the WebTransactions server and forms the basis for a *->WebTransactions application*. The base directory contains the *->templates* and all the files and program references (links) which are necessary in order to run a WebTransactions application.

BCAM application name

Corresponds to the openUTM generation parameter `BCAMAPPL` and is the name of the *–>openUTM application* through which *–>UPIC* establishes the connection.

browser

Program which is required to call and display *->HTML* pages. Browsers are, for example, Microsoft Internet Explorer or Mozilla Firefox.

browser display print

The WebTransactions browser display print prints the information displayed in the *->browser*.

browser platform

Operating system of the host on which a ->*browser* runs as a client for WebTransactions.

buffer

Definition of a record, which is transmitted from a ->*service*. The buffer is used for transmitting and receiving messages. In addition there is a specific buffer for storing the ->*recognition criteria* and for data for the representation on the screen.

capturing

To enable WebTransactions to identify the received *->formats* at runtime, you can open a *->session* in *->WebLab* and select a specific area for each format and name the format. The format name and *->recognition criteria* are stored in the *->capture database*. A *->template* of the same name is generated for the format. Capturing forms the basis for the processing of format-specific templates for the WebTransactions for OSD and MVS product variants.

capture database

The WebTransactions capture database contains all the format names and the associated *->recognition criteria* generated using the *->capturing* technique. You can use *->WebLab* to edit the sequence and recognition criteria of the formats.

CGI

(**C**ommon **G**ateway **I**nterface)
Standardized interface for program calls on *->Web servers*. In contrast to the static output of a previously defined*->HTML* page, this interface permits the dynamic construction of HTML pages.

class

Contains definitions of the *->properties* and *->methods* of an *->object*. It provides the model for instantiating objects and defines their interfaces.

class template

In WebTransactions, a class template contains valid, recurring statements for the entire object class (e.g. input or output fields). Class templates are processed when the *->evaluation operator* or the `toString` method is applied to a *->host data object*.

client

Requestors and users of services in a network.

cluster

Set of identical *->WebTransactions applications* on different servers which are interconnected to form a load-sharing network.

communication object

This controls the connection to an *->host application* and contains information about the current status of the connection, the last data to be received etc.

conversion tools

Utilities supplied with WebTransactions. These tools are used to analyze the data structures of *->openUTM applications* and store the information in files. These files can then be used in WebLab as *->format description sources* in order to generate WTML templates and *->FLD files*.
COBOL data structures or IFG format libraries form the basis for the conversion tools. The conversion tool for DRIVE programs is supplied with the product DRIVE.

daemon

Name of a process type in Unix system/POSIX systems which runs in the background and performs no I/O operations at terminals.

data access control

Monitoring of the accesses to data and *->objects* of an application.

data type

Definition of the way in which the contents of a storage location are to be interpreted. Each data type has a name, a set of permitted values (value range), and a defined number of operations which interpret and manipulate the values of that data type.

dialog

Describes the entire communication between browser, WebTransactions and *->host application*. It will usually comprise multiple *->dialog cycles*. WebTransactions supports a number of different
types of dialog.
- ->*passive dialog*
- ->*active dialog*
- ->*synchronized dialog*
- ->*non-synchronized dialog*

dialog cycle

Cycle that comprises the following steps when a ->*WebTransactions application* is executed:
- construct an ->*HTML* page and send it to the *->browser*
- wait for a response from the browser
- evaluate the response fields and possibly send them to the*->host application* for further processing

A number of dialog cycles are passed through while a *->WebTransactions application* is executing.

distinguished name

The Distinguished Name (DN) in ->*LDAP* is hierarchically organized and consists of a number of different components (e.g. “country, and below country: organization, and below organization: organizational unit, followed by: usual name”). Together, these components provide a unique identification of an object in the directory tree.

Thanks to this hierarchy, the unique identification of objects is a simple matter even in a worldwide directory tree:
- The DN "Country=DE/Name=Emil Person" reduces the problem of achieving a unique identification to the country DE (=Germany).
- The DN "Organization=FTS/Name=Emil Person" reduces it to the organization FTS.
- The DN "Country=DE/Organization=FTS/Name=Emil Person" reduces it to the organization FTS located in Germany (DE).

document directory

->*Web server* directory containing the documents that can be accessed via the network. WebTransactions stores files for download in this directory, e.g. the WebLab client or general start pages.

Domain Name Service (DNS)

Procedure for the symbolic addressing of computers in networks. Certain computers in the network, the DNS or name server, maintain a database containing all the known host names and *IP numbers* in their environment.

dynamic data

In WebTransactions, dynamic data is mapped using the WebTransactions object model, e.g. as a *->system object*, host object or user input at the browser.

EHLLAPI

Enhanced **H**igh-**L**evel **L**anguage **API**
Program interface, e.g. of terminal emulations for communication with the SNA world. Communication between the transit client and SNA computer, which is handled via the TRANSIT product, is based on this interface.

EJB

(**E**nterprise **J**ava**B**ean)
This is a Java-based industry standard which makes it possible to use in-house or commercially available server components for the creation of distributed program systems within a distributed, object-oriented environment.

entry page

The entry page is an *->HTML page* which is required in order to start a ->*WebTransactions application* This page contains the call which starts WebTransactions with the first *->template*, the so-called start template.

evaluation operator

In WebTransactions the evaluation operator replaces the addressed *->expressions* with their result (object attribute evaluation). The evaluation operator is specified in the form ##expression#.

expression

A combination of *->literals*, *->variables*, operators and expressions which return a specific result when evaluated.

FHS

Format **H**andling **S**ystem
Formatting system for BS2000/OSD applications.

field

A field is the smallest component of a service and element of a ->*record* or ->*buffer*.

field file (*.fld file)

In WebTransactions, this contains the structure of a ->*format* record (metadata).

filter

Program or program unit (e.g. a library) for converting a given ->*format* into another format (e.g. XML documents to ->*WTScript* data structures).

format

Optical presentation on alphanumeric screens (sometimes also referred to as screen form or mask).

In WebTransactions each format is represented by a ->*field file* and a ->*template*.

format type

(only relevant in the case of ->*FHS* applications and communication via ->*UPIC*) Specifies the type of format: #format, +format, -format or *format.

format description sources

Description of multiple ->*formats* in one or more files which were generated from a format library (FHS/IFG) or are available directly at the ->*host* for the use of “expressive” names in formats.

function

A function is a user-defined code unit with a name and ->*parameters*. Functions can be called in ->*methods* by means of a description of the function interface (or signature).

holder task

A process, a task or a thread in WebTransactions depending on the operating system platform being used. The number of tasks corresponds to the number of users. The task is terminated when the user logs off or when a time-out occurs. A holder task is identical to a ->*WebTransactions session*.

host

The computer on which the- >*host application* is running.

host adapter

Host adapters are used to connect existing ->*host applications* to WebTransactions. At runtime, for example, they have the task of establishing and terminating connections and converting all the exchanged data.

host application

Application that is integrated with WebTransactions.

host control object

In WebTransactions, host control objects contain information which relates not to individual fields but to the entire ->*format*. This includes, for example, the field in which the cursor is located, the current function key or global format attributes.

host data object

In WebTransactions, this refers to an ->*object* of the data interface to the ->*host application*. It represents a field with all its field attributes. It is created by WebTransactions after the reception of host application data and exists until the next data is received or until termination of the ->*session*.

host data print

During WebTransactions host data print, information is printed that was edited and sent by the ->*host application*, e.g. printout of host files.

host platform

Operating system of the host on which the ->*host applications* runs.

HTML

(**H**yper**t**ext **M**arkup **L**anguage)
See ->*Hypertext Markup Language*

HTTP

(**H**ypertext **T**ransfer **P**rotocol)
This is the protocol used to transfer ->*HTML* pages and data.

HTTPS

(**H**ypertext **T**ransfer **P**rotocol **S**ecure)
This is the protocol used for the secure transfer of ->*HTML* pages and data.

hypertext

Document with links to other locations in the same or another document. Users click the links to jump to these new locations.

Hypertext Markup Language

(**H**yper**t**ext **M**arkup **L**anguage)
Standardized markup language for documents on the Web.

Java Bean

Java programs (or ->*classes*) with precisely defined conventions for interfaces that allow them to be reused in different applications.

KDCDEF

openUTM tool for generating *->openUTM applications*.

LDAP

(**L**ightweight **D**irectory **A**ccess **P**rotocol)
The X.500 standard defines DAP (Directory Access Protocol) as the access protocol. However, the Internet standard “LDAP” has proved successful specifically for accessing X.500 directory services from a PC.
LDAP is a simplified DAP protocol that does not support all the options available with DAP and is not compatible with DAP. Practically all X.500 directory services support both DAP and LDAP. In practice, interpretation problems may arise since there are various dialects of LDAP. The differences between the dialects are generally small.

literal

Character sequence that represents a fixed value. Literals are used in source programs to specify constant values (“literal” values).

master template

WebTransactions template used to generate the Automask and the format-specific templates.

message queuing (MQ)

A form of communication in which messages are not exchanged directly, rather via intermediate queues. The sender and receiver can work at separate times and locations. Message transmission is guaranteed regardless of whether or not a network connection currently exists.

method

Object-oriented term for a ->*function*. A method is applied to the ->*object* in which it is defined.

module template

In WebTransactions, a module template is used to define ->*classes*, ->*functions* and constants globally for a complete ->*session*. A module template is loaded using the `import()` function.

MT tag

(**M**aster **T**emplate tag)
Special tags used in the dynamic sections of *->master templates*.

multitier architecture

All client/server architectures are based on a subdivision into individual software components which are also known as layers or tiers. We speak of 1-tier, 2-tier, 3-tier and multitier models. This subdivision can be considered at the physical or logical level:

– We speak of logical software tiers when the software is subdivided into modular components with clear interfaces.
– Physical tiers occur when the (logical) software components are distributed across different computers in the network.

With WebTransactions, multitier models are possible both at the physical and logical level.

name/value pair

In the data sent by the *->browser*, the combination, for example, of an *->HTML* input field name and its value.

non-synchronized dialog

Non-synchronized dialogs in WebTransactions permit the temporary deactivation of the checking mechanism implemented in ->*synchronized dialogs*. In this way, ->*dialogs* that do not form part of the synchronized dialog and have no effect on the logical state of the ->*host application* can be incorporated. In this way, for example, you can display a button in an ->*HTML* page that allows users to call help information from the current host application and display it in a separate window.

object

Elementary unit in an object-oriented software system. Every object possesses a name via which it can be addressed, *->attributes*, which define its status together with the ->*methods* that can be applied to the object.

openUTM

(**U**niversal **T**ransaction **M**onitor)
Transaction monitor from Fujitsu Technology Solutions, which is available for BS2000/OSD and a variety of Unix platforms and Windows platforms.

openUTM application

A ->*host application* which provides services that process jobs submitted by ->*clients* or other ->*host applications*. openUTM responsibilities include transaction management and the management of communication and system resources. Technically speaking, the UTM application is a group of processes which form a logical unit at runtime.
openUTM applications can communicate both via the client/server protocol ->*UPIC* and via the emulation interface (9750).

openUTM-Client (UPIC)

The openUTM-Client (UPIC) is a product used to create client programs for openUTM. openUTM-Client (UPIC) is available, for example, for Unix platforms, BS2000/OSD platforms and Windows platforms.

openUTM program unit

The services of an ->*openUTM application* are implemented by one or more openUTM program units. These can be addressed using transaction codes and contain special openUTM function calls (e.g. KDCS calls).

parameter

Data which is passed to a ->*function* or a ->*method* for processing (input parameter) or data which is returned as a result of a function or method (output parameter).

passive dialog

In the case of passive dialogs in WebTransactions, the dialog sequence is controlled by the *->host application*, i.e. the host application determines the next *->template* which is to be processed. Users who access the host application via WebTransactions pass through the same dialog steps as if they were accessing it from a terminal. WebTransactions uses passive dialog control for the automatic conversion of the host application or when each host application format corresponds to precisely one individual template.

password

String entered for a ->*user id* in an application which is used for user authentication (->*system access control*).

polling

Cyclical querying of state changes.

pool

In WebTransactions, this term refers to a shared directory in which WebLab can create and maintain *->base directories*. You control access to this directory with the administration program.

post

To send data.

posted object (`wt_Posted`)

List of the data returned by the *->browser*. This *->object* is created by WebTransactions and exists for the duration of a *->dialog* cycle.

process

The term “process” is used as a generic term for process (in Solaris, Linux and Windows) and task (in BS2000/OSD).

project

In the WebTransactions development environment, a project contains various settings for a ->Web*Transactions application*. These are saved in a project file (suffix `.wtp`). You should create a project for each WebTransactions application you develop, and always open this project for editing.

property

Properties define the nature of an ->*object*, e.g. the object “Customer” could have a customer name and number as its properties. These properties can be set, queried, and modified within the program.

protocol

Agreements on the procedural rules and formats governing communications between remote partners of the same logical level.

protocol file

- openUTM-Client: File into which the openUTM error messages as are written in the case of abnormal termination of a conversation.
- In WebTransactions, protocol files are called trace files.

roaming session

->*WebTransactions sessions* which are invoked simultaneously or one after another by different ->*clients*.

record

A record is the definition of a set of related data which is transferred to a ->*buffer*. It describes a part of the buffer which may occur one or more times.

recognition criteria

Recognition criteria are used to identify ->*formats* of a ->*terminal application* and can access the data of the format. The recognition criteria selected should be one or more areas of the format which uniquely identify the content of the format.

scalar

->*variable* made up of a single value, unlike a ->*class*, an ->*array* or another complex data structure.

service (openUTM)

In ->*openUTM,* this is the processing of a request using an *->openUTM application*. There are dialog services and asynchronous services. The services are assigned their own storage areas by openUTM. A service is made up of one or more ->*transactions*.

service application

->*WebTransactions session* which can be called by various different users in turn.

service node

Instance of a ->*service*. During development and runtime of a ->*method* a service can be instantiated several times. During modelling and code editing those instances are named service nodes.

session

When an end user starts to work with a ->*WebTransactions application* this opens a WebTransactions session for that user on the WebTransactions server. This session contains all the connections open for this user to the ->*browsers*, special ->*clients* and ->*hosts*.
A session can be started as follows:

- Input of a WebTransactions URL in the browser.
- Using the `START_SESSION` method of the `WT_REMOTE` client/server interface.

A session is terminated as follows:

- The user makes the corresponding input in the output area of this ->*WebTransactions application* (not via the standard browser buttons).
- Whenever the configured time that WebTransactions waits for a response from the ->*host application* or from the ->*browser* is exceeded.
- Termination from WebTransactions administration.
- Using the `EXIT_SESSION` method of the `WT_REMOTE` client/server interface.

A WebTransactions session is unique and is defined by a ->*WebTransactions application* and a session ID. During the life cycle of a session there is one ->*holder task* for each WebTransactions session on the WebTransactions server.

SOAP

(originally **S**imple **O**bject **A**ccess **P**rotocol)
The *->XML* based SOAP protocol provides a simple, transparent mechanism for exchanging structured and typecast information between computers in a decentralized, distributed environment.
SOAP provides a modular package model together with mechanisms for data encryption within modules. This enables the uncomplicated description of the internal interfaces of a ->*Web-Service*.

style

In WebTransactions this produces a different layout for a *->template*, e.g. with more or less graphic elements for different*->browsers*. The style can be changed at any time during a *->session*.

synchronized dialog

In the case of synchronized dialogs (normal case), WebTransactions automatically checks whether the data received from the web browser is genuinely a response to the last ->*HTML* page to be sent to the ->*browser*. For example, if the user at the web browser uses the **Back** button or the History function to return to an "earlier" HTML page of the current *->session* and then returns this, WebTransactions recognizes that the data does not correspond to the current *->dialog cycle* and reacts with an error message. The last page to have been sent to the browser is then automatically sent to it again.

system access control

Check to establish whether a user under a particular ->*user ID* is authorized to work with the application.

system object (`wt_System`**)**

The WebTransactions system object contains *->variables* which continue to exist for the duration of an entire *->session* and are not cleared until the end of the session or until they are explicitly deleted. The system object is always visible and is identical for all name spaces.

TAC

See *->transaction code*

tag

->HTML, *->XML* and *->WTML* documents are all made up of tags and actual content. The tags are used to mark up the documents e.g. with header formats, text highlighting formats (bold, italics) or to give source information for graphics files.

TCP/IP

(**T**ransport **C**ontrol **P**rotocol/**I**nternet **P**rotocol)
Collective name for a protocol family in computer networks used, for example, in the Internet.

template

A template is used to generate specific code. A template contains fixed information parts which are adopted unchanged during generation, as well as variable information parts that can be replaced by the appropriate values during generation.
A template is a *->WTML* file with special tags for controlling the dynamic generation of a *->HTML* page and for the processing of the values entered at the *->browser*. It is possible to maintain multiple template sets in parallel. These then represent different ->*styles* (e.g. many/few
graphics, use of Java, etc.).
WebTransactions uses different types of template:

- ->*Automask templates* for the automatic conversion of the ->*formats* of MVS and OSD applications.
- Custom templates, written by the programmer, for example, to control an *->active dialog.*
- Format-specific templates which are generated for subsequent post-processing.
- Include templates which are inserted in other templates.
- *->Class templates*
- ->*Master templates* to ensure the uniform layout of fixed areas on the generation of the Automask and format-specific templates.
- Start template, this is the first template to be processed in a WebTransactions application.

template object

->Variables used to buffer values for a *->dialog cycle* in WebTransactions.

terminal application

Application on a ->*host* computer which is accessed via a 9750 or 3270 interface.

terminal hardcopy print

A terminal hardcopy print in WebTransactions prints the alphanumeric representation of the *->format* as displayed by a terminal or a terminal emulation.

transaction

Processing step between two synchronization points (in the current operation) which is characterized by the ACID conditions (**A**tomicity, **C**onsistency, **I**solation and **D**urability). The intentional changes to user information made within a transaction are accepted either in their entirety or not at all (all-or-nothing rule).

transaction code/TAC

Name under which an openUTM service or *->openUTM program unit* can be called. The transaction code is assigned to the openUTM program unit during configuration. A program unit can be assigned several transaction codes.

UDDI

(**U**niversal **D**escription, **D**iscovery and **I**ntegration)
Refers to directories containing descriptions of *->Web services*. This information is available to web users in general.

Unicode

An alphanumeric character set standardized by the International Standardisation Organisation (ISO) and the Unicode Consortium. It is used to represent various different types of characters: letters, numerals, punctuation marks, syllabic characters, special characters and ideograms. Unicode brings together all the known text symbols in use across the world into a single character set. Unicode is vendor-independent and system-independent. It uses either two-byte or four-byte character sets in which each text symbol is encoded. In the ISO standard, these character sets are termed UCS-2 (Universal Character Set 2) or UCS-4. The designation UTF-16 (Unicode Transformation Format 16-bit), which is a standard defined by the Unicode Consortium, is often used in place of the designation UCS-2 as defined in ISO. Alongside UTF-16, UTF-8 (Unicode Transformation Format 8 Bit) is also in widespread use. UTF-8 has become the character encoding method used globally on the Internet.

UPIC

(**U**niversal **P**rogramming **I**nterface for **C**ommunication)
Carrier system for openUTM clients which uses the X/Open interface, which permity CPI-C client/server communication between a CPI-C-Client application and the openUTM application.

URI

(**U**niform **R**esource **I**dentifier)
Blanket term for all the names and addresses that reference objects on the Internet. The generally used URIs are*->URLs*.

URL

(**U**niform **R**esource **L**ocator)
Description of the location and access type of a resource in the *->Internet*.

user exit

Functions implemented in C/C++ which the programmer calls from a *->template*.

user ID

User identification which can be assigned a password (->*system access control*) and special access rights (->*data access control*).

variable

Memory location for variable values which requires a name and a ->*data type*.

visibility of variables

->*Objects* and ->*variables* of different dialog types are managed by WebTransactions in different address spaces. This means that variables belonging to a ->*synchronized dialog* are not visible and therefore not accessible in a ->*asynchronous dialog* or in a dialog with a remote application.

web server

Computer and software for the provision of ->*HTML* pages and dynamic data via ->*HTTP*.

web service

Service provided on the Internet, for example a currency conversion program. The SOAP protocol can be used to access such a service. The interface of a web service is described in ->*WSDL*.

WebTransactions application

This is an application that is integrated with ->*host applications* for internet/intranet access. A WebTransactions application consists of:

– a ->*base directory*
– a start template
– the ->*templates* that control conversion between the ->*host* and the ->*browser*.
– protocol-specific configuration files.

WebTransactions platform

Operating system of the host on which WebTransactions runs.

WebTransactions server

Computer on which WebTransactions runs.

WebTransactions session

See ->*session*

WSDL

(**W**eb **S**ervice **D**efinition **L**anguage)
Provides ->*XML* language rules for the description of ->*web services*. In this case, the web service is defined by means of the port selection.

WTBean

In WebTransactions *->WTML* components with a self-descriptive interface are referred to as WTBeans. A distinction is made between inline and standalone WTBeans:

- An inline WTBean corresponds to a part of a WTML document
- A standalone WTBean is an autonomous WTML document

A number of WTBeans are included in of the WebTransactions product, additional WTBeans can be downloaded from the WebTransactions homepage *ts.fujitsu.com/products/software/openseas/webtransactions.html*.

WTML

(**W**eb**T**ransactions **M**arkup **L**anguage)
Markup and programming language for WebTransactions *->templates*. WTML uses additional *->WTML tags* to extend *->HTML* and the server programming language *->WTScript*, e.g. for data exchange with *->host applications*. WTML tags are executed by WebTransactions and not by the *->browser* (serverside scripting).

WTML tag

(**W**eb**T**ransactions **M**arkup **L**anguage-Tag)
Special WebTransactions tags for the generation of the dynamic sections of an *->HTML* page using data from the*->host application*.

WTScript

Serverside programming language of WebTransactions. WTScripts are similiar to client-side Java scripts in that they are contained in sections that are introduced and terminated with special tags. Instead of using *->HTML*-`SCRIPT` tags you use *->WTML-Tags*: `wtOnCreateScript` and `wtOnReceiveScript`. This indicates that these scripts are to be implemented by WebTransactions and not by the *->browser* and also indicates the time of execution. OnCreate scripts are executed before the page is sent to the browser. OnReceive scripts are executed when the response has been received from the browser.

XML

(e**X**tensible **M**arkup **L**anguage)
Defines a language for the logical structuring of documents with the aim of making these easy to exchange between various applications.

XML schema

An XML schema basically defines the permissible elements and attributes of an XML description. XML schemas can have a range of different formats, e.g. DTD (**D**ocument **T**ype **D**efinition), XML Schema (W3C standard) or XDR (**X**ML **D**ata **R**educed).

Abbreviations

BO	**B**usiness **O**bject
CGI	**C**ommon **G**ateway **I**nterface
DN	**D**istinguished **N**ame
DNS	**D**omain **N**ame **S**ervice
EJB	**E**nterprise **J**ava**B**ean
FHS	**F**ormat **H**andling **S**ystem
HTML	**H**yper**t**ext **M**arkup **L**anguage
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
HTTPS	**H**yper**t**ext **T**ransfer **P**rotocol **S**ecure
IFG	**I**nteraktiver **F**ormat **G**enerator
ISAPI	**I**nternet **S**erver **A**pplication **P**rogramming **I**nterface
LDAP	**L**ightweight **D**irectory **A**ccess **P**rotocol
LPD	**L**ine **P**rinter **D**aemon
MT-Tag	**M**aster-**T**emplate-Tag
MVS	**M**ultiple **V**irtual **S**torage
OSD	**O**pen **S**ystems **D**irection
SGML	**S**tandard **G**eneralized **M**arkup **L**anguage
SOAP	**S**imple **O**bject **A**ccess **P**rotocol

SSL	**S**ecure **S**ocket **L**ayer
TCP/IP	**T**ransport **C**ontrol **P**rotocol/**I**nternet **P**rotocol
Upic	**U**niversal **P**rogramming **I**nterface for **C**ommunication
URL	**U**niform **R**esource **L**ocator
WSDL	**W**eb **S**ervices **D**escription **L**anguage
wtc	**W**eb**T**ransactions **C**omponent
WTML	**W**eb**T**ransactions **M**arkup **L**anguage
XML	e**X**tensible **M**arkup **L**anguage

Related publications

WebTransactions manuals

You can download all manuals from the Web address *http://manuals.ts.fujitsu.com*.

WebTransactions
Concepts and Functions
Introduction

WebTransactions
Template Language
Reference Manual

WebTransactions
Client APIs for WebTransactions
User Guide

WebTransactions
Connection to openUTM Applications via UPIC
User Guide

WebTransactions
Connection to OSD Applications
User Guide

WebTransactions
Connection to MVS Applications
User Guide

WebTransactions
Web Frontend for Web Services
User Guide

Other publications

SOAP Version 1.2
Part 1: Messaging Framework
http://www.w3.org/TR/soap12-part1/

SOAP Version 1.2
Part 2: Adjuncts
http://www.w3.org/TR/soap12-part2

Web Services Description Language (WSDL) 1.1
http://www.w3.org/TR/wsdl

Index

A
active dialog 109, 112
addHeader (WT_SOAP class) 70
analyseResponse (WT_SOAP class) 68
architecture
  WebTransactions 9
array 109
asynchronous message 109
attribute 109
  of WT_SOAP 79
automask template 110

B
base data type 109
base directory 110
  converting to a new version 16
  creating 15
BCAM application name 110
BCAMAPPL 110
binding 80
Body attribute 24
browser 110
browser display print 110
browser platform 110
buffer 110

C
call
  proxy method 62
capture database 111
capturing 110
CGI (Common Gateway Interface) 111
CGI script, using (example) 89
class 111
  templates 111
  WT_RPC 93
  WT_SOAP 51, 52
  WT_SOAP_COM_FUNCTIONS 75
  WT_SOAP_HEADER 84
client 111
close 23
cluster 111
comattr 80
communication object 111
  activate 21
  connection parameters 46
  creating 45
  end 23
COMMUNICATION_FILE_NAME (system object attribute) 18
COMMUNICATION_FILE_TYPE (system object attribute) 18
configuration
  HTTP access 75
connect
  web service via SOAP 47
connection
  opening multiple 45
constructor
  WT_SOAP_HEADER 84
Content-Type (HTTP header field) 18
ContentType (HTTP header) 64
ContentType attribute 24
conversion tools 111
createProxysWithPrefix (WT_SOAP class) 73

D
daemon 111
data
  dynamic 113

data access control 112
data type 112
  SOAP object 80
dialog 112
  active 112
  non-synchronized 112, 117
  passive 112, 118
  synchronized 112, 121
  types 112
dialog cycle 112
distinguished name 112
document directory 113
documentation 55
Domain Name Service (DNS) 113

E
EHLLAPI 113
EJB 113
entry page 113
envelope 79
envelope (a SOAP message) 48
evaluation operator 113
exception object 77
  soapCode 78
executeGetRequest (WT_SOAP class) 68
executeRequest (WT_SOAP class) 67
expression 113

F
FHS 113
field 114
field file 114
FILE 78
filter 114
  ready-made 35
  user exits 34
  WTScript functions 34
first template see start template
fld file 114
format 114
  #format 114
  *format 114
  +format 114
  -format 114
format description source 114
format type 114
function 114

G
generate SOAP message
  proxy method 64
getHeaderObjects (WT_SOAP class) 70
getHeaderObjectTree (WT_SOAP class) 72

H
Header attribute 24
header field
  Authorization 25, 26, 29
  Content-Length 26, 30
  Content-Type 27, 30
  Host 25, 26, 29
  Proxy-Authorization 25, 26, 29
  sequence 33
  User-Agent 25, 26, 29
  user-defined 34
holder task 114
host 114
host adapter 114
host application 115
host control object 115
host data object 115
host data print 115
host object 24
host object attribute
  Body 24
  ContentType 24
  Header 24, 32
host platform 115
HTML 115
HTTP 77, 115
http 79
HTTP error messages 101
HTTP header
  proxy method 64
HTTP raw data, process 34
HTTP server
  accessing from WebTransactions 89
HTTP_RETURN_CODE (system object

attribute) 18, 22
HTTPS 115
hypertext 115
Hypertext Markup Language (HTML) 115

I
initFromWSDLUri (WT_SOAP class) 65
inline WTBean 125
installation
    WebTransactions 34

J
Java Bean 116

K
KDCDEF 116

L
LDAP 116
literals 116

M
master template 116, 122
    tag 116
message 80
message queuing 116
method 116
    addHeader (WT_SOAP class) 70
    analyseResponse (WT_SOAP class) 68
    createProxysWithPrefix (WT_SOAP class) 73
    executeGetRequest (WT_SOAP class) 68
    executeRequest (WT_SOAP class) 67
    getHeaderObjects (WT_SOAP class) 70
    getHeaderObjectTree (WT_SOAP class) 72
    initFromWSDLUri (WT_SOAP class) 65
    Proxy methods 62
    removeAllHeaders (WT_SOAP class) 70
    setAuthorization (WT_SOAP class) 75
    setProxy (WT_SOAP class) 76
    setProxyAuthorization (WT_SOAP class) 76
    setRunMode (WT_SOAP class) 66
    setSOAPVersion (WT_SOAP class) 69
    setTimeout (WT_SOAP class) 77
    WT_RPC_ADD_METHOD 100
    WT_RPC_CLOSE 98
    WT_RPC_INVOKE 99
    WT_RPC_OPEN 96
METHOD (system object attribute) 18
module template 116
MT tag 116
multitier architecture 117

N
name/value pair 117
namespace identifier 48
non-synchronized dialog 112, 117

O
object 117
open 21
openUTM 117
    application 117
    Client 118
    program unit 118
    service 120
operations 112

P
PARAMETER 78
parameter 118
parameter transfer
    proxy method 62
PARSE 78
passive dialog 112, 118
password 118
PASSWORD (system object attribute) 19
polling 118
pool 118
portType 80
posted object 118
posting 118
process 119
project 119
property 119
protocol 119
protocol file 119
PROXY (system object attribute) 19

proxy method 62
call and parameter transfer 62
generating a SOAP message 64
HTTP header 64
return value 64
PROXY_PASSWORD (system object attribute) 19
PROXY_PORT (system object attribute) 19
PROXY_USER (system object attribute) 19
proxyObjects 62

R
receive 22
storing data 18
recognition criteria 119
record 119
record structure 114
remote function
calling 99
defining 100
removeAllHeaders (WT_SOAP class) 70
return
value for proxy method 64

S
scalar 119
send 21
send query 21
sequence
header fields 33
service 79
service (openUTM) 120
service node 120
session 120
WebTransactions 120
setAuthorization (WT_SOAP class) 75
setHTTPHeader() 64
setProxy (WT_SOAP class) 76
setProxyAuthorization (WT_SOAP class) 76
setRunMode (WT_SOAP class) 66
setSOAPVersion (WT_SOAP class) 69
setTimeout (WT_SOAP class) 77
Simple Object Access Protocol see SOAP
SOAP 120
body 80
client-side support 51
envelope 48
integration in WebTransactions 47
request 80
service 50
service described with WSDL 48
web service connecting 47
SOAP data type 80
SOAP object
representation in object tree of WebLab 54
SOAPAction 64
soapCode (exception object) 78
SOCKET 77
SSL_CERT_FILE (system object attribute) 19
SSL_KEY_FIL (system object attribute) 19
SSL_PASSPHRASE (system object attribute) 19
SSL_PROTOCOL (system object attribute) 19
standalone WTBean 125
start template 122
simple 40
wtstartOSD.htm 36
start template set 36
StartTemplateHTTP.htm 40
code 41
store message 18
structure
WSDL document 49
WT_SOAP object 53
style 121
synchronized dialog 112, 121
system access control 121
system object 121
interaction between attribute and methods 21
OSD-specific attributes 17
system object attribute
HTTP_RETURN_CODE 18
PASSWORD 19
PROXY 19
PROXY_PASSWORD 19
PROXY_PORT 19
PROXY_USER 19

SSL_CERT_FILE 19
SSL_KEY_FIL 19
SSL_PASSPHRASE 19
SSL_PROTOCOL 19
TIMEOUT_HTTP 20
URL 20
USER 20

T
TAC 123
tag 121
TCP/IP 121
template 122
class 111
master 122
object 122
start 122
terminal application 122
terminal hardcopy printing 122
Thread 114
TIMEOUT_HTTP (system object attribute) 20, 22
transaction 122
transaction code/TAC 123
types 80

U
UDDI 50, 123
Unicode 123
Universal Description, Discovery and Integration Project see UDDI
update
base directory 16
UPIC 123
URI 123
URL 123
URL (system object attribute) 20
USER (system object attribute) 20
user exits 123
as filters 34
user ID 124
UTM see openUTM

V
value range of a data type 112
variable 124
VERSION_MISMATCH 78
visibility 124

W
web information
using (example) 91
web server 124
web service 124
connecting via SOAP 47
on different hosts 57
Web Services Description Language see WSDL
WebLab
object tree of WT_SOAP 54
WebTransactions
architecture 9
session 120
WebTransactions application 124
starting (remote) 96
terminating (remote) 98
WebTransactions platform 124
WebTransactions server 124
WSDL 48, 78, 124
document structure 49
wt_attributes 62
WT_RPC class 93
constructor 95
implementing 95
WT_RPC_ADD_METHOD 100
WT_RPC_CLOSE 98
WT_RPC_INVOKE 99
WT_RPC_OPEN 96
WT_SOAP 51, 52
attribute 79
constructor 60
proxy methods 62
WT_SOAP method
addHeader 70
analyseResponse 68
createProxysWithPrefix 73
executeGetRequest 68
executeRequest 67
getHeaderObjects 70
getHeaderObjectTree 72

- initFromWSDLUri 65
- removeAllHeaders 70
- setAuthorization 75
- setProxy 76
- setProxyAuthorization 76
- setRunMode 66
- setSOAPVersion 69
- setTimeout 77

WT_SOAP object, structure 53
WT_SOAP_COM_FUNCTIONS 75
WT_SOAP_HEADER 78, 84
- constructor 84

WTBean 125
- wtcHTTP 45

wtcHTTP 45
WTML 125
WTML tag 125
WTScript 125
WTScript filter 34
wtstartOSD.htm 36
WWW browser 110
WWW server 124

X

XML 125
XML schema 125