User Guide - English

FUJITSU

WebTransactions V7.5

Client APIs for WebTransactions

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

Contents

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 client access in WebTransactions

The diagram below shows the architecture of client access in WebTransactions. This manual deals with the items shown here in yellow:

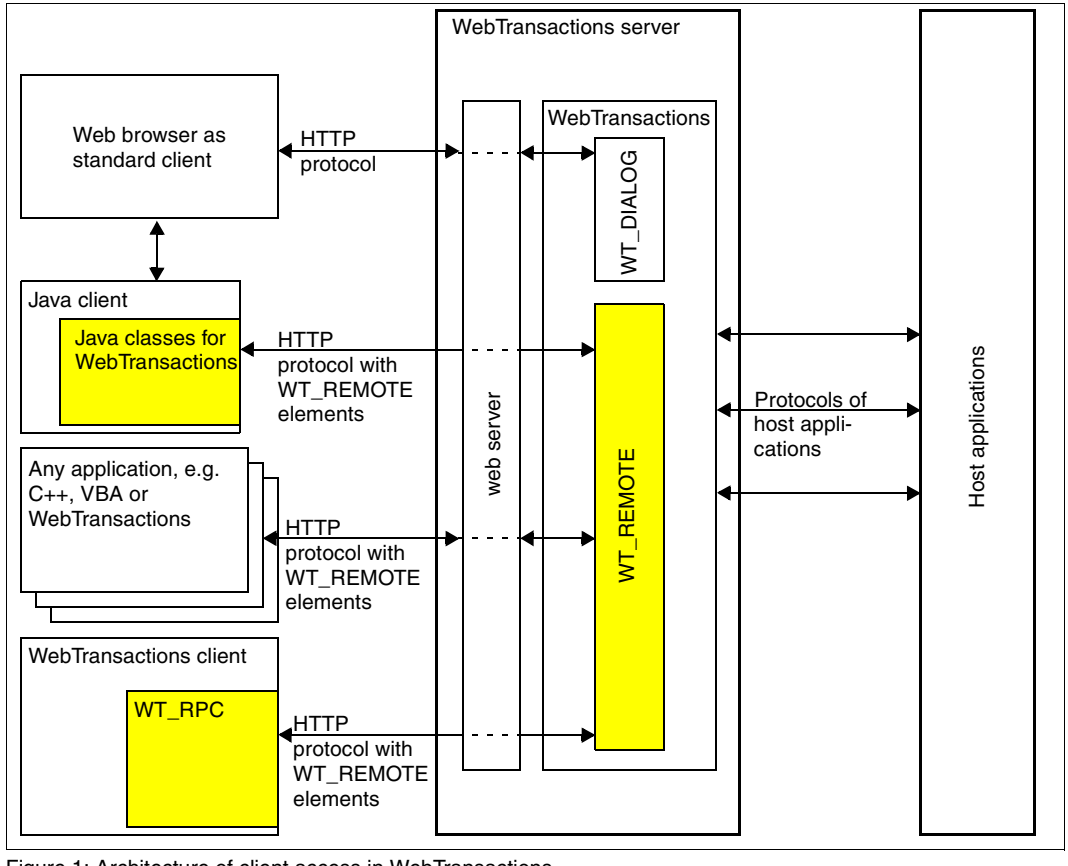

Figure 1: Architecture of client access in WebTransactions

Web browser

The Web browser previously served as the standard client. Using the Web browser, host applications that were formerly only available via terminal emulations can be operated via a Web interface. In this case, the client only accesses the elements of the Web pages created by the WebTransactions application.

WebTransactions server

The WebTransactions server is the computer which runs the WebTransactions application and the web server for client access to this application.

web server

In principle, all clients use the HTTP protocol to access the WebTransactions application via the web server. If the client is a Web browser, the inquiries are forwarded to `WT_DIALOG` which assumes control of the end user session.

If the HTTP messages contain `WT_REMOTE` elements, the inquiries are forwarded to the `WT_REMOTE` interface of WebTransactions and are processed there.

WT_REMOTE

`WT_REMOTE` is an open interface of WebTransactions for all types of clients. It is thus possible to access the resources (objects and methods) of WebTransactions applications from any programs and thereby use their functionality in other applications. The only prerequisite for this is that the client is capable of sending multi-part HTTP messages.

WebTransactions client

The WebTransactions client is a computer running a WebTransactions application which accesses another WebTransactions application.

WT_RPC

`WT_RPC` is a programming interface of WebTransactions for distributed WebTransactions applications. In this case, communication with the web server and hence utilization of the `WT_REMOTE` interface are performed internally by `WT_RPC` and are thus transparent to the programmer. This means that WebTransactions applications can access other WebTransactions applications without major programming effort.

Java client

`WTJavaClient` classes can be used to write Java applets and Java programs for access to WebTransactions. The methods of the Java classes are modeled on the calls of the `WT_REMOTE` interface.

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 that applies to all the supply units and describes the possibilities of the HTTP host adapter:

  Access to Dynamic Web Contents

  This manual describes:

  – How you can use WebTransactions to access a HTTP server and use its resources.

  – The integration of SOAP (Simple Object Access Protocol) protocols in WebTransactions and the connection of web services via SOAP.

- 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 aimed at anyone who creates clients for WebTransactions applications or who wishes to distribute functional elements of WebTransactions applications on a number of servers.

The individual chapters describe the necessary protocols and interfaces for carrying out these tasks.

This manual provides all the client-specific information supplementary to the introductory WebTransactions manual “Concepts and Functions” and the WebTransactions reference manual “Template Language”.

1.5 New features

You will find an overview of all the changes in WebTransactions V7.5 in the WebTransactions manual “Concepts and Functions”

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
CR LF	Representation for “line feed” in the HTTP examples
►	Prompt telling you to do something.
⇨	Refers to detailed information

2 Client concept of WebTransactions

The client concept described here allows for the flexible implementation of WebTransactions as a supplier of data for all types of clients.

2.1 The Web browser as standard client

The structure of the client server access to WebTransactions appliances is fundamentally different from the usual dialog-oriented WebTransactions applications with a Web browser as standard client. In order to make the difference clearer, the standard case will be again shown below, that means how WebTransactions transfers host applications to interactive applications in the WWW. Using a Web browser available on most platforms, the HTTP protocol can be used to access interactive applications that were formerly only available via terminal emulations.

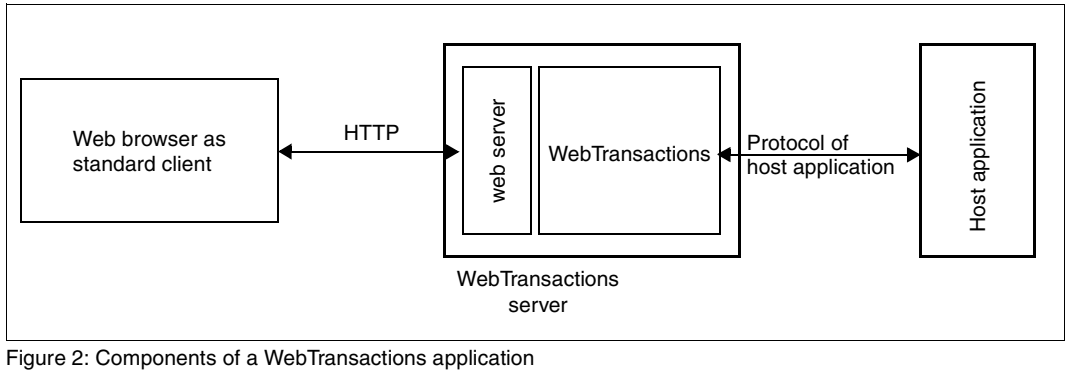

Figure 2: Components of a WebTransactions application

The browser sends its inquiries to WebTransactions via the web server. It can display the HTML pages created by the WebTransactions server, and the browser user can communicate with the host application via the WebTransactions application in accordance with the specifications in the templates. The user can only exert direct influence on the process of the WebTransactions application insofar as this is permitted by the WebTransactions application (e.g. a button to close the session).

2.2 The WT_REMOTE interface

The open client interface `WT_REMOTE` enables all types of clients to directly access the resources of a WebTransactions application. Special incapsulations of this interface (`WT_RPC` and the `WTJavaClient` classes) represent WebTransactions resources using proxies, and thereby enable access as though these were part of the client application.

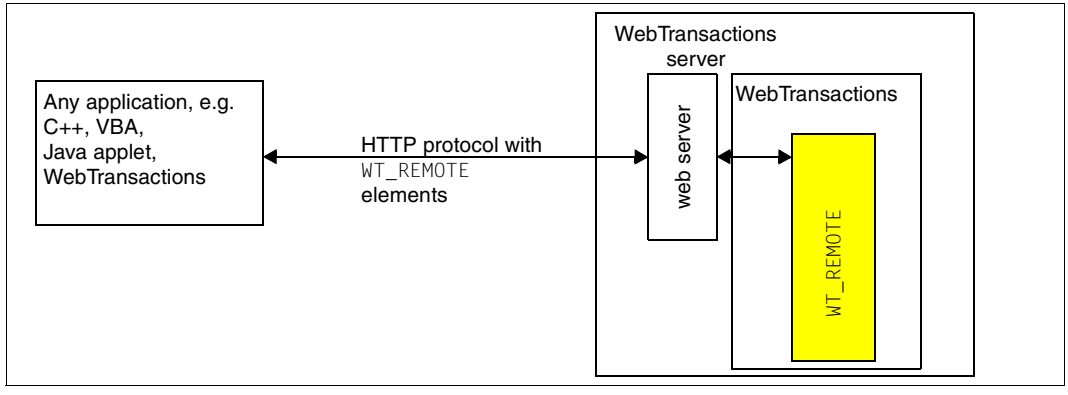

Figure 3: The `WT_REMOTE` interface for client access

Here, an application of any type sends inquiries to the web server in the form of multi-part HTTP messages in a special format. The web server forwards these messages to the `WT_REMOTE` interface of WebTransactions, where they are interpreted and processed.

The main difference from the standard client is that direct access is possible to the remote application in the form of a remote procedure call, and not only access to the web pages of the WebTransactions application. Client applications are allowed the following accesses:

- Start a WebTransactions session
- Execute a command in the WebTransactions session:
  - send data to the WebTransactions session
  - receive data from the WebTransactions session
  - create WebTransactions objects
  - call WebTransactions methods
- Close a WebTransactions session

In this way clients can actively influence a WebTransactions application and control this application remotely, or use the functionality of a WebTransactions application for their own purposes.

Standard libraries (described in the following two sections) are available for the most important application scenarios (Java applets and WebTransactions itself as client). For all other clients, please see the detailed description of the `WT_REMOTE` interface in the chapter “Appendix: The WT_REMOTE interface” on page 61.

2.3 The WT_RPC class library for WebTransactions clients

Using the client/server protocol `WT_REMOTE`, any application capable of sending HTTP multi-part messages can obtain access to a WebTransactions application. For WebTransactions applications as clients, there is also a user-friendly interface in the form of a class library called `WT_RPC`, which enables uncomplicated communication using this protocol.

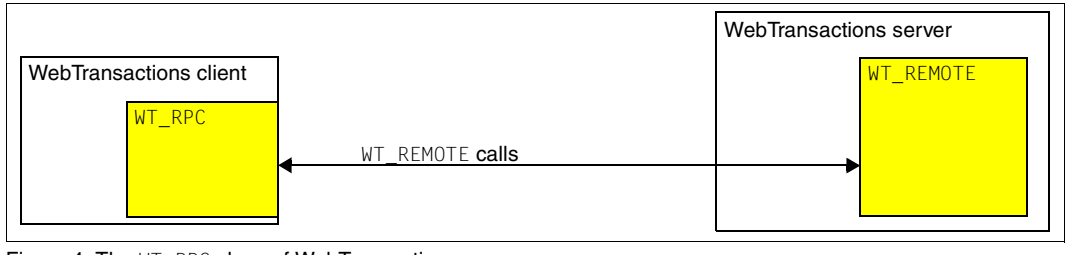

Figure 4: The `WT_RPC` class of WebTransactions

The `WT_RPC` class library provides a range of methods for communication with WebTransactions server applications via the `WT_REMOTE` interface, without having to consider the technical details of HTTP communication. Although communication takes place via the web server as before, this is transparent to the programmer on the client side. `WT_RPC` thus represents a high-level interface.

This interface offers the following functionality:

- starting and closing a remote WebTransactions application
- calling remote methods
- local definition of remote methods so that they can be used in the same way as local methods

This functionality is all that is needed to implement distributed WebTransactions applications simply and effectively.

2.4 Class library for Java clients

The `WTJavaClient.jar` class library can be used to write applets and Java programs which access the data of a WebTransactions application using the methods of the predefined classes. The classes of `WTJavaClient.jar` are based on JDK V1.1. The corresponding methods use the interface `WT_REMOTE` internally but hide it behind an object interface that is much easier to use.

A Java client for WebTransactions comprises various classes:

– user-defined and predefined classes based on JDK V1.1 which contain the on-screen graphical representation of the program and the application logic

– `WTJavaClient` classes, responsible for connecting to a WebTransactions session and for data exchange

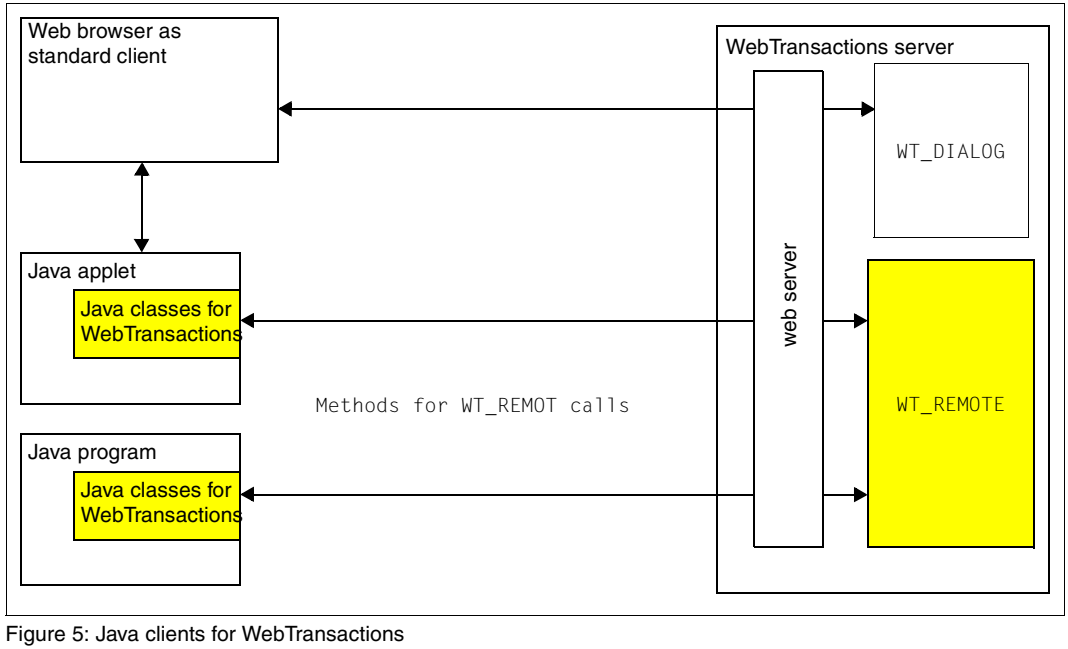

Figure 5: Java clients for WebTransactions

An applet differs from a Java application in that the top class must be derived from the `java.applet.Applet` class. This is not the case for a Java application, which must simply have a `main` method. Different security concepts also apply.

The `WTJavaClient.jar` class library contains three classes:

– `WTSession` for starting and closing WebTransactions sessions and for restarting a new or existing session

– `WTObject` for representing the data of a WebTransactions session, for establishing data structures, and for setting and querying objects

– `WTObjectRemoteAccess` for the actual data exchange with a WebTransactions session, for instantiating objects in the WebTransactions session, and for calling methods in the WebTransactions session

2.4.1 Applet for WebTransactions access

The applet call is defined in a template and is sent to the browser as part of the HTML page. The browser starts the applet automatically and thereby loads the applet class and the Java classes for WebTransactions on the client computer.

The `WTJavaClient` classes must be located on the computer also running the web server and WebTransactions, because an applet is only permitted to establish one connection to the computer from which it was loaded. During installation, the `WTJavaClient.jar` archive is therefore stored in the document directory of the web server under the `webtav75` directory.

When the applet is to set up a connection to a WebTransactions session, the session parameters must be transferred with the PARAM tag.

Example

```
<APPLET NAME="CLIPBOOK"
          CODE="clipBook.class"
          ARCHIVE="/webtav75/JavaDemo/java/clipBook.jar"
          WIDTH="516" HEIGHT="207">
    <PARAM NAME="SERVER"     VALUE="##WT_SYSTEM.CGI.SERVER_NAME#">
    <PARAM NAME="SERVERPORT" VALUE="##WT_SYSTEM.CGI.SERVER_PORT#">
    <PARAM NAME="HREF"       VALUE="##WT_SYSTEM.HREF#">
    <PARAM NAME="LANGUAGE"   VALUE="##WT_SYSTEM.LANGUAGE#">
    <PARAM NAME="TRACELEVEL" VALUE="1">
  </APPLET>
```

With these values, the applet can attach to the existing connection with the WebTransactions session (`attach` method) and can access the data of this WebTransactions application. In addition, an applet itself can start a WebTransactions application (`open` method).

Demo Java applications which clarify this procedure are supplied with WebTransactions. During installation, these applications are installed in the document directory of the web server under the subdirectory `webtav75/JavaDemo`, if the **WebTransactions Demo Applications** option has been selected. A demo application is a notebook function which can make notes during the entire course of the WebTransactions session after a non-synchronous call, displays the existing notes, and saves these notes when the session is closed.

2.4.2 Java program for WebTransactions access

The Java program itself can start a WebTransactions application using the `open` method. The Java program and the `WTJavaClient` classes are located on the client computer which accesses the WebTransactions application.

2.4.3 Data exchange between the Java client and WebTransactions

The Java client uses the `upload` method to transfer data to the WebTransactions session. Depending on the content, this data is created as new global variables or as attributes of `WT_SYSTEM` or `WT_HOST` in the WebTransactions session.

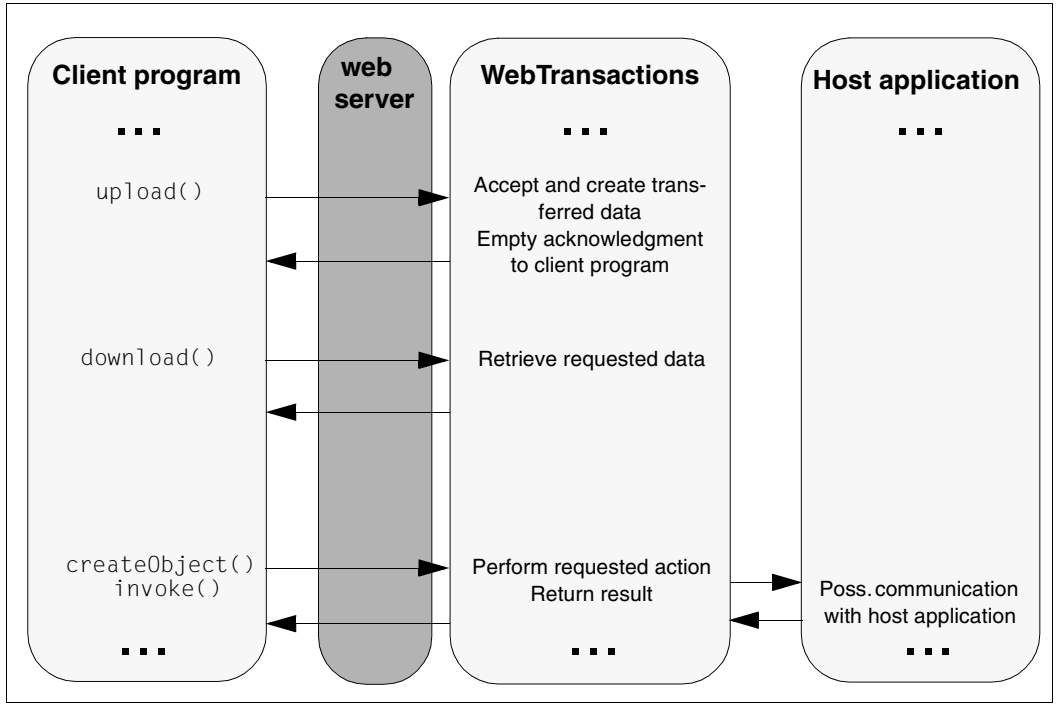

Figure 6: Data exchange between Java client and WebTransactions

Conversely, the Java client can also use the `download` method to query the data of the addressed WebTransactions session. In addition, the `CreateObject` method can be used to execute integrated or user-defined constructors, and the `invoke` method used to execute methods and functions in the WebTransactions session. The result of such a call is returned to the client program.

3 The WT_RPC class

An object of the `WT_RPC` class represents a connection to a remote WebTransactions application, which was implemented on the server side via the `WT_REMOTE` interface. The `WT_RPC` class is defined in the delivered template `wtRPC.htm`, which is saved in your base directory in the subdirectory `config/forms`. You have to include `wtRPC.htm` in your template to create objects of the the `WT_RPC` class. An example for an application of this class can be found in chapter “Example: Distributed WebTransactions application with WT_RPC” on page 55.

3.1 Constructor

The `WT_RPC` constructor creates a new `WT_RPC` object via which the `WT_REMOTE` calls to a remote WebTransactions application can be processed.

`WT_RPC()`
`WT_RPC(`*urlOfWebTA*`,` *basedir*`)`

If the constructor is called without arguments, only the communication object is created. If the remote WebTransactions application is also to be started, the following arguments must be specified:

urlOfWebTA
: URL of the `WTPublish` program on the computer whose WebTransactions application is to be started. *urlOfWebTA* must refer to the program `WTPublish.exe` or `WTPublishISAPI.dll` on the remote machine
(e.g. `http://remoteMachine/cgi-bin/WTPublish.exe`).

basedir
: String with the base directory of the remote WebTransactions application.

If the call was successful and if the remote WebTransactions application could be started, the `WT_CONNECTED` attribute of the new `WT_RPC` object is assigned the value `true`. Otherwise, it is defined as `false`.

3.2 Attributes

An object of the `WT_RPC` class has three attributes that can be addressed. These attributes are generally set and used by the methods, but can also be used by the client application for checking purposes.

```
WT_URL
WT_BASEDIR
WT_CONNECTED
```

WT_URL
: Contains the URL for calling the WebTransactions application. This URL is set by the `open` call or the constructor call, provided a URL was specified as an argument in the call.

WT_BASEDIR
: Contains the base directory of the remote WebTransactions application. This base directory is parameterized by the `open` call or the constructor call, provided a base directory was specified as an argument in the call.

WT_CONNECTED
: Contains the status of the connection to the remote WebTransactions application. This status is set to `true` by the `open` call, provided the connection was established successfully. Otherwise, the status is set to `false`.

3.3 Methods

A description of the methods of the `WT_RPC` class follows.

3.3.1 open method

The `open` method starts a remote a WebTransactions application or creates a connection with a current WebTransactions application.

`open()`
`open(`*urlOfWebTA*`)`
`open(`*urlOfWebTA*`,` *basedir*`)`
`open(`*urlOfWebTA*`,` *basedir*`,` *session*`,` *signature*`)`

---

urlOfWebTA
: URL of the `WTPublish` program on the computer whose WebTransactions application is to be started. *urlOfWebTA* must refer to the program `WTPublish.exe` or `WTPublishISAPI.dll` on the remote machine
(e.g. `http://remoteMachine/cgi-bin/WTPublish.exe`).

If the method is called without any parameters, the values of the `WT_URL` and `WT_BASEDIR` attributes of the `WT_RPC` object are used to establish the connection; these values must be transferred beforehand to the constructor or to a preceding `open` call.

basedir
: Base directory of the remote WebTransactions application. If the method is called without the *basedir* parameter, the value of the `WT_BASEDIR` attribute of the `WT_RPC` object is used to establish the connection; this value must be transferred beforehand to the constructor of the `WT_RPC` class or to a preceding `open` call.

session, *signature*
: If the `open` method is used additionally to transfer for the parameters *session* and *signature* the values of the attributes `WT_SYSTEM.SESSION` and `WT_SYSTEM.SIGNATURE` of a remote current WebTransactions application, a connection is established with this WebTransactions application. No new session is started in this case.

The values of the transferred parameters are stored in the `WT_URL` and `WT_BASEDIR` attributes of the `WT_RPC` object. If the remote WebTransactions application is started successfully, the `WT_CONNECTED` attribute is set to `true`; otherwise, it is set to `false`. The value of this attribute is also returned by the method as the result. If the parameters *session* and *signature* are not specified, it is tried to start a new remote session in all cases. If in this case there is already a connection for the current object the previously connected remote application is terminated (see section “close method” on page 26)

3.3.2 close method

The `close` method ends the remote WebTransactions application which is connected with this `WT_RPC` object. The `WT_CONNECTED` attribute of the object is set to `false`.

```
close()
```

3.3.3 invoke method

The `invoke` method calls a function in the remote WebTransactions application.

`invoke(`*name*`,` *codeBase*`,` *argArray*`)`

name Specifies the name of the function in the remote WebTransactions application

codeBase
Specifies the WTML document containing the function definition

argArray
An array whose elements are transferred to the remote function as arguments

The method returns the result of the remote function. If no connection is established to a remote WebTransactions application, the value `null` is returned.

Example

A remote function `add`, which is defined in the WTML document `calc.htm`, calculates the sum of all its parameters and returns this value as the result. A `WT_RPC` object `rwt` has already been created for the remote WebTransactions application by a preceding constructor call. The following call then returns the result `42` in the `answer` variable.

```
answer = rwt.invoke( 'add', 'calc', new Array( 1, 2, 3, 4, 32 ));
```

3.3.4 addMethod Method

Calling a remote function using the `invoke` method is somewhat laborious, because the defined document must always be specified again and the parameters must be transferred in an array.

Calling the `addMethod` method defines a new method of the underlying `WT_RPC` object as a representative for the remote function. The remote function can now be called as a method of the `WT_RPC` object.

`addMethod(`*name*`,` *codeBase*`)`

---

name Specifies the name of the function in the remote WebTransactions application

codeBase
Specifies the WTML document containing the function definition

Example

A local representative method can be defined by the following call for the `add` function described above in the section “invoke method” on page 26. The remote function can then be called using this representative method.

```
rwt.addMethod( 'add', 'calc.htm' );
answer = rwt.add( 1, 2, 3, 4, 32 );
```

3.4 Developing distributed applications with WT_RPC

If you want to develop distributed WebTransactions applications, it is advisable to observe the following sequence:

1. Define functionality

   First define the functionality of the WebTransactions application. This functionality should be made available in the form of functions, i.e. you should define an API for the desired functionality. To simplify access to the functions at a later stage, the API should be provided in a separate template:

   ```
   //myAPI
   function turnover(company) {...}
   ```

   To simplify the test and subsequent distribution, it is advisable to provide the entire functionality as methods of an object:

   ```
   myAPI = new Object();
   myAPI.turnover = turnover;
   ```

2. Test the functionality locally

   The functions should first be tested locally in the same WebTransactions application. This means that you can use the full functionality of WebLab during the test. You simply have to write a test template which accesses the functions. This test template includes the functions you have defined and implements an interface for the test:

   Turnover:

   ```
   ##myAPI.turnover(WT_POSTED.c)#
   ```

   Continue testing until your functions are working satisfactorily.

3. Distribute the WebTransactions application

   In this step you must perform two tasks:

   - Distribute your documents to two WebTransactions applications. One application contains the templates which provide functionality (server), while the other application contains the templates you used for the test (client).
   - In the test template on the client, replace the API object with an object of the `WT_RPC` class and define the methods on the server as methods of this object:

     ```
     myAPI = new WT_RPC(...);
     myAPI.addMethod('turnover',...);
     ```

   When you specify the URL and base directory of your WebTransactions application in the constructor of the `WT_RPC` object in your client application, your client application accesses the WebTransactions application remotely and you have distributed your application.

4 The com.siemens.webta Java package

Java classes are combined into packages. The `com.siemens.webta` package is supplied for communication with WebTransactions. During installation, this package is stored as the `WTJavaClient.jar` archive both in the subdirectory `lib` in the WebTransactions installation directory and in the `webtav75` directory under the web server document directory. The `com.siemens.webta` package contains the following classes:

– `WTSession`, which provides methods for establishing a connection to the WebTransactions application.

– `WTObject`, which provides methods for object representation of the remote `WTObject` data of an active WebTransactions session.

– `WTObjectRemoteAccess`, which provides methods for exchanging data with the WebTransactions application.

In order to work with the `WTJavaClient` classes of WebTransactions, the path under which the classes can be accessed must be added to the Java environment variable `CLASSPATH`. In the source program itself, the package must be made known to your Java program using the `import` statement.

Example

```
import com.siemens.webta.*;
```

4.1 WTSession class

The `WTSession` class contains the basic methods for communication with WebTransactions. An object of the `WTSession` class references a remote WebTransactions session. With an object of this class a new WebTransactions session can be started or the object can be connected with an existing session. For this purpose you enter the address in the construktor. Use the `open` or `attach` methods to establish whether a new or existing WebTransactions session should be used.

Internally, the methods use the interface `WT_REMOTE`, which is described in chapter “Appendix: The WT_REMOTE interface” on page 61.

```
public class WTSession
```

4.1.1 Constructors

Use a constructior to create a new object for access to a remote WebTransactions session. Here you transfer the information that is required for addressing a session.

Two cases can be distinguished:

- A new WebTransactions session is created and used. In this case it is sufficient to specify the WebTransactions application by entering the server and base directory.
- An already existing WebTransactions setting should be used. For addressing here, the session Id and the signature of this session are required. They can be transferred via the parameter `href`.

This distinction is not definitive because the decision to invoke a new or existing session can be set not until using the `open` or `attach` methods. The `attach` method enables the session Id and the signature to be entered at a later date to invoke the current session. But you can also use the `open` method and thus ignore any session parameters previously transferred with `href`.

Use in applets

There is a special constructor for use in applets that contains only the applet object and determines all required entries from the correspondingly named parameters. Both cases, i.e. new and already existing WebTransactions sessions are equally supported.

4.1.1.1 WTSession for a new WebTransactions session

`WTSession` creates a new `WTSession` object to be used for a new connection to a WebTransactions application.

```
WTSession(String protocol, String server, int serverPort,
          String WTScriptName, String basedir)
WTSession(String server, int serverPort, String WTScriptName, String basedir)
WTSession(String protocol, String server, String WTScriptName, String basedir)
```

protocol
: Protocol for the connection; this parameter can be omitted if the HTTP protocol is used.
Possible values: http, https
Default: http

server
: Internet address or symbolic name of the computer running the WebTransactions application.

serverPort
: Port for the HTTP connection; default:

  80 if http is specified for *protocol*

  443 if https is specified for *protocol*

WTScriptName
: Path for the WTPublish call (e.g. `/cgi-bin/WTPublish.exe` or `/scripts/WTPublishISAPI.dll`).

basedir
: Base directory of the WebTransactions application.

i Please note that you must specify either the parameter *protocol* or *serverPort*. Both parameters must not be omitted simultaneously.

Example

```
WTSession osd1=new WTSession("http", "111.222.111.222", 8080,
                             "/cgi-bin/WTPublish.exe",
                             "c:\\WebTABase\\osd_test");
```

4.1.1.2 WTSession for an already existing WebTransactions session

`WTSession` creates a new `WTSession` object to be attached to an already existing WebTransactions session.

`WTSession(String` *protocol*`, String` *server*`, int` *serverPort*`, String` *href*`)`
`WTSession(String` *server*`, int` *serverPort*`, String` *href*`)`
`WTSession(String` *protocol*`, String` *server*`, String` *href*`)`
`WTSession(String` *server*`, String` *href*`)`

---

protocol
: Protocol for the connection; this parameter can be omitted if the HTTP protocol is used.
Possible values: http, https
Default: http

server
: Internet address or symbolic name of the computer running the WebTransactions application.

serverPort
: Port for the HTTP connection; default:

80 if http is specified for *protocol*

443 if https is specified for *protocol*

href
: Relative URL for the WebTransactions application (corresponds to the values of the `HREF` or `HREF_ASYNC` attribute of the global system object).

Example

```
string href="cgi-bin/WTPublish.exe?WT_SYSTEM_BASEDIR=c:/myBase&
      WT_SYSTEM_FORMAT=myStart&WT_SYSTEM_SESSION=E-43585543569&
      WT_SYSTEM_SIGNATURE=1242545991206130607";
WTSession osd1=new WTSession("http", "rechner1", 8080, href);
```

4.1.1.3 WTSession for an applet

`WTSession` creates a new `WTSession` object for a new or existing connection to a WebTransactions session in an applet.

`WTSession(Applet` *app*`)`

---

app Current applet

This constructor can only be used in a Java applet if the parameters required to establish a connection to a WebTransactions application are transferred in the HTML page via parameters. Please note that HTML is not case-sensitive.

The following parameters can be specified with an applet using the PARAM tag in a WTML template:

- Connection parameters that must always be specified:

  protocol Protocol for the connection; this parameter can be omitted if the HTTP protocol is used.
  Possible values: http, https
  Default: http

  server Internet address or symbolic name of the computer running the WebTransactions application.

  serverPort Port for the HTTP connection; default:

  80 if http is specified for *protocol*

  443 if https is specified for *protocol*

- Connection parameters for a **new** WebTransactions session:

  WTScriptName
  Path for the WTPublish call (e.g. `/cgi-bin/WTPublish.exe` or `/scripts/WTPublishISAPI.dll`).

  basedir Base directory of the WebTransactions application.

- Connection parameters for an **existing** WebTransactions session:

  href Relative URL for the WebTransactions session (corresponds to the values of the `HREF` or `HREF_ASYNC` attribute of the global system object).

The following parameters can also be specified instead of the *href* parameter:

WTScriptName
: Path for the WTPublish call (e.g. `/cgi-bin/WTPublish.exe` or `/scripts/WTPublishISAPI.dll`).

basedir
: Base directory of the WebTransactions application.

session
: Session ID of the current WebTransactions session, corresponding to the `WT_SYSTEM.SESSION` attribute of the global system object.

signature
: Signature of the current WebTransactions session, corresponding to the `WT_SYSTEM.SIGNATURE` attribute of the global system object.

Example

Below is an extract from a WTML template in which an applet is called with the necessary parameters for an existing connection to a WebTransactions session.

```
...
<APPLET   code="Applet1.class"
          archive="/webtav75/WTJavaClient.jar"
          codebase="/applets">

   <PARAM name="protocol" value="http">
   <PARAM name="server" value="111.222.111.222">
   <PARAM name="serverPort" value="8080">
   <PARAM name="WTScriptName" value="/cgi-bin/WTPublish.exe">
   <PARAM name="basedir" value="c:/WebTAbase">
   <PARAM name="session" value="##WT_SYSTEM.SESSION#">
   <PARAM name="signature" value="##WT_SYSTEM.SIGNATURE#">
</APPLET>
...
```

The applet object can then be created with the following constructor call:

```
WTSession myApp = new WTSession(this);
```

4.1.2 Methods

The methods of the `WTSession` class are described below in alphabetical order.

4.1.2.1 attach method

The `attach` method establishes a connection with an existing WebTransactions session. If a session has already been specified in the constructor (through the parameters *href* or *session* and *signature*), `attach` can be used without parameters. Otherwise, you must instance the *session* and *signature* parameters. The `attach` method returns the current `WTSession` object. The `attach` method does not call a WebTransactions session directly. This occurs with the methods of the `WTObjectRemoteAccess` class, see also section "WTObjectRemoteAccess class" on page 49.

Possible exceptions are described in section "Exceptions" on page 41.

```
WTSession attach(String session, String signature) throws
          WTSessionConnectionException, WTSessionParameterException
WTSession attach() throws
          WTSessionConnectionException, WTSessionParameterException
```

session
: Session ID of the current WebTransactions session, corresponding to the `SESSION` attribute of the global system object.

signature
: Signature of the current WebTransactions session, corresponding to the `SIGNATURE` attribute of the global system object.

Example

```
myApp.attach();
```

4.1.2.2 close method

The `close` method closes a WebTransactions session that was opened using the `open` or `attach` methods. It resets the addressing of the WebTransactions application in the underlying object because the WebTransactions session referenced up until now no longer exists. If no session has been started yet, only the addressing of a WebTransactions application is reset. Possible exceptions are described in section “Exceptions” on page 41.

```
void close() throws
          WTSessionConnectionException,
          WTCloseSessionException
```

Example

```
myApp.close();
```

4.1.2.3 open method

The `open` method starts a new WebTransactions session and returns the current `WTSession` object. Addressing of the WebTransactions application was already set with the constructor. The `open` method calls the `close` method implicitly before a new WebTransactions session is started, and calls the `attach` method following a successful start.

If the new session starts with special timeout, language or style settings you can call the corresponding methods before the `open` method. Possible exceptions are described in section “Exceptions” on page 41.

```
WTSession open() throws
          WTSessionConnectionException,
          WTSessionParameterException,
          WTCloseSessionException
```

Example

```
myApp.open();
```

4.1.2.4 setApplTimeout method

The `setApplTimeout` method sets the value of the `TIMEOUT_APPLICATION` system object attribute for the next call of the WebTransactions application. `setApplTimeout` returns the current `WTSession` object.

`WTSession setUserTimeout(int` *applTimeout*`)`

---

applTimeout
: Time span in seconds for responses from the host application within the WebTransactions session.

> **i** Please note that when this method is used there is no communication with the WebTransactions application. The value set for `TIMEOUT_APPLICATION` is buffered and is sent with the next communication method (e.g. `open`).

Example

```
myApp.setApplTimeout(60);
```

4.1.2.5 setLanguage method

The `setLanguage` method sets the value of the `LANGUAGE` system object attribute for the next call of the WebTransactions application. `setLanguage` returns the current `WTSession` object.

`WTSession setLanguage(String` *language*`)`

---

language
: `config` directory of the WebTransactions application, in which the templates for the corresponding interface language are stored.

> **i** Please note that when this method is used there is no communication with the WebTransactions application. The value set for `LANGUAGE` is buffered and is sent with the next communication method (e.g. `open`).

Example

```
myApp.setLanguage("engl");
```

4.1.2.6 setStyle method

The setStyle method sets the value of the `STYLE` system object attribute for the next call of the WebTransactions application. `setStyle` returns the current `WTSession` object.

```
WTSession setStyle (String style)
```

style Subdirectory under the `config` directory of the WebTransactions application, in which the templates for the corresponding interface style are stored.

Please note that when this method is used there is no communication with the WebTransactions application. The value set for `STYLE` is buffered and is sent with the next communication method (e.g. `open`).

Example

```
myApp.setStyle("lay1");
```

4.1.2.7 setTraceLevel method

The `setTraceLevel` method activates the trace function for the `WTJavaClient` object.

```
void setTraceLevel(int traceLevel)
```

traceLevel

Numeric value which determines the trace level. The individual values are defined as variables. The table below provides an overview of which values you can specify and which variable definition corresponds to each value:

Value	Variable definition	Meaning
0	`public static final int traceLevel_Off`	The trace function is deactivated
1	`public static final int traceLevel_WTSession`	Trace of the `WTSession` class
2	`public static final int traceLevel_WTObjectRemoteAccess`	Trace of the `WTObjectRemoteAccess` class
4	`public static final int traceLevel_WTObject`	Trace of the `WTObject` class
8	`public static final int traceLevel_WTXMLHandler`	Trace of the XML parser

Table 1: Trace levels

The various trace options can be combined by adding the numeric values or linking the constants with the logical OR operand.

Example

```
myApp.setTraceLevel(WTSession.traceLevel_WTSession +
                    WTSession.traceLevel_WTObject);
```

If you use the sum of the numeric values for the constants (`WTSession` =1 and `WTObject` =4), the following line is synonymous with the first:

```
myApp.setTraceLevel(5);
```

4.1.2.8 setUserTimeout method

The `setUserTimeout` method sets the value of the `TIMEOUT_USER` system object attribute for the next call of the WebTransactions application. `setUserTimeout` returns the current `WTSession` object.

`WTSession setUserTimeout(int` *userTimeout*`)`

---

userTimeout
: Time span in seconds for responses from the user within the WebTransactions session.

> **i** Please note that when this method is used there is no communication with the WebTransactions application. The value set for `TIMEOUT_USER` is buffered and is sent with the next communication method (e.g. `upload`). As the value does not come into effect until the next time the WebTransactions application is called, until then it has no effect on the current session.

Example

```
myApp.setUserTimeout(360);
```

4.1.3 Exceptions

The `open`, `attach` and `close` methods of the `WTSession` object can trigger the following exceptions:

Exception	Meaning
`WTSessionConnectionException`	This exception is triggered when the connection parameters for a WebTransactions session are not correct. Check the parameters *protocol*, *server*, *serverPort* or *WTScriptName* which you specified in the `WTSession` constructor.
`WTSessionParameterException`	This exception is triggered when the *session* or *signature* parameter is not set for a WebTransactions session. Check these parameters, which you specified in the `WTSession` constructor.
`WTCloseSessionException`	This exception is triggered when the WebTransactions session cannot be closed. The *session* and *signature* parameters are however always reset, which means that this exception can generally be ignored.

Table 2: Exception for `WTSession`

4.1.4 Example

Below is an extract from a Java program.

```
// construct a WTSession object for a remote WebTransactions session
WTSession wtSession = new WTSession("PGTD1234",
                                    80,
                                   "/scripts/WTPublishISAPI.dll",
                                   "c:/basedirs/myAppl");
// set the application and user timeout object for the new
// WebTransactions session
wtSession.setApplTimeout("60").setUserTimeout("3600");
// open a new WebTransactions session for the WTSession object
wtSession.open();
...
```

4.2 WTObject class

The `WTObject` class reflects the main part of the WebTransactions object model. It provides methods for processing objects and thereby supplies the `WTObjectRemoteAccess` class with the objects and methods for data exchange with the remote WebTransactions application

```
public class WTObject
```

4.2.1 Constructor

`WTObject` creates a new `WTObject` object which corresponds to a WebTransactions object. Data types and classes are predefined as Java variables. The value of the object is managed as a character string. If you want to work with the real value, you must convert it into the corresponding data type.

```
WTObject(int objectType )
WTObject(int objectType, String objectValue)
WTObject(int objectType, int objectClass)
WTObject(int objectType, int objectClass, String objectValue)
```

objectType

Data type of the new object. The following values can be specified:

Java variable definition	WTML data type
`public static final int TYPE_UNDEFINED`	`undefined`
`public static final int TYPE_STRING`	`string`
`public static final int TYPE_NUMBER`	`number`
`public static final int TYPE_BOOLEAN`	`boolean`
`public static final int TYPE_OBJECT`	`object`
`public static final int TYPE_FUNCTION`	`function`

Table 3: Java definitions for WTML data types

objectClass

Class of the new object. The following values can be specified:

Java variable definition	WTML class
`public static final int CLASS_UNDEFINED`	`Undefined`
`public static final int CLASS_STRING`	`String`
`public static final int CLASS_NUMBER`	`Number`
`public static final int CLASS_BOOLEAN`	`Boolean`
`public static final int CLASS_OBJECT`	`Object`
`public static final int CLASS_ARRAY`	`Array`
`public static final int CLASS_REGEXP`	`Regexp`
`public static final int CLASS_FUNCTION`	`Function`
`public static final int CLASS_WTHOSTOBJECT`	`WT_Hostobject`
`public static final int CLASS_WTCOMMUNICATION`	`WT_Communication`
`public static final int CLASS_DOCUMENT`	`Document`
`public static final int CLASS_WTUSEREXIT`	`WT_Userexit`
`public static final int CLASS_DATE`	`Date`

Table 4: Java definitions for WTML classes

If you do not specify a class, an object of the `Undefined` class is created.

objectValue

Value of the new object. If you do not assign a value to an object, the object is initialized with the value `null`.

Example

```
WTObject local_obj= new WTObject(WTObject.TYPE_OBJECT,
                                 WTObject.CLASS_OBJECT);
```

4.2.2 Methods

The methods of the `WTObject` class are described below in alphabetical order.

4.2.2.1 getAttribute method

The `getAttribute` method returns the specified attribute of the current object as a `WTObject` object. If the desired attribute does not exist, the value `null` is returned.

`WTObject getAttribute(String` *attributeName*`)`

attributeName
Name of the desired attribute, which can also be specified in the object hierarchy.

Example

```
WTObject local_obj=remote_obj.download("WT_SYSTEM");
WTObject style=local_obj.getAttribute("STYLE");
WTObject script=local_obj.getAttribute("CGI.SCRIPT_NAME");
```

4.2.2.2 getAttributeNames method

The `getAttributeNames` method returns the name of all attributes of the current object in a character string array.

`String [] getAttributeNames()`

Example

```
String[] Attribute=local_obj.getAttributeNames();
```

4.2.2.3 getValueAsString method

The `getValueAsString` method returns the value of the current object as a character string. Please note that you must convert the value into the corresponding data type if you want to work with the real value.

```
String getValueAsString()
```

Example

```
String value=local_obj.getValueAsString();
```

4.2.2.4 getWTClass method

The `getWTClass` method returns the class of the current object as a constant. The constants are explained in the table “Java definitions for WTML classes” on page 43.

```
int getWTClass()
```

Example

```
int objectClass=local_obj.getWTClass();
```

4.2.2.5 getWTType method

The `getWTType` method returns the data type of the current object as a constant. The constants are explained in the table “Java definitions for WTML data types” on page 42.

```
int getWTType()
```

Beispiel

```
int objektTyp=local_obj.getWTType();
```

4.2.2.6 removeAttribute method

The `removeAttribute` method removes the specified attribute from the current object and returns the current object. If the specified attribute does not exist, this method has no effect.

```
WTObject removeAttribute(String attributeName)
```

attributeName
Name of the attribute to be removed.

Example

```
local_obj.removeAttribute("myObj");
```

4.2.2.7 setAttribute method

The `setAttribute` method sets the specified attribute in the current object and returns the set object. If the set attribute does not yet exist in the current object, it is created. If a level does not exist in the specified object hierarchy, `setAttribute` returns the value `null` and the attribute is not set.

```
WTObject setAttribute(String attributeName, WTObject object)
```

attributeName
Name of the attribute to be set or created. The attribute can also be specified in the object hierarchy.

object Description of the attribute as a `WTObject` object.

Example

```
local_obj.setAttribute("STYLE",new WTObject(WTObject.TYPE_STRING,"N"));
```

4.2.2.8 setValue method

The `setValue` method sets the value of the current object. Please note that the value must always be a character string, even if the data type of the object is not a character string.

```
void setValue(String value)
```

value
Value of the current object as a character string.

Example

```
local_obj.setValue("42");
```

4.2.3 Exceptions

No exceptions are defined for this class.

4.2.4 Example

```
...
// create a new WebTransactions remote access object and open a
// WebTransactions session,
// it is supposed that the WTSession parameters are passed as parameters
// to the applet via the <param> tag.
WTObjectRemoteAccess wtSession = new WTObjectRemoteAccess (
                                 new WTSession(this).open());

// create and download WT_SYSTEM object from WebTransactions session
WTObject wt_system = wtSession.download("WT_SYSTEM");
// determine if connection runs via proxy HOST1 and store it in new system
// object
if (wt_system.getAttribute("CGI.REMOTE_HOST").getValueAsString().
equals("HOST1"))
   wt_system.setAttribute("PROXY", new WTObject(WTObject.TYPE_STRING,
"YES"));
else
   wt_system.setAttribute("PROXY", new WTObject(
                          WTObject.TYPE_STRING, "NO"));

// change attribute style of system object to value APPLREMOTE
wt_system.getAttribute("STYLE").setValue("APPLREMOTE");

// upload current wt_system object into WT_SYSTEM object of remote
// WebTransactions session
wtSession.upload(wt_system ,"WT_SYSTEM");

// download all value attributes of host objects of WT_HOST.MYCOM from
// WebTransactions
WTObject wt_host = wtSession.download("WT_HOST.MYCOM..Value");
WTObject my_comm = wt_host.getAttribute("MYCOM");
// shortcut to WT_HOST.MYCOM

// invoke remote function wtmlMeth of template func.htm with parameters
//of two host objects and store the return value in local variable
WTObject[] params = new WTObject[2];
params[0] = my_comm.getAttribute("E_01_001_80.Value");
params[1] = my_comm.getAttribute("E_02_001_80.Value");
String result = (wtSession.invoke(
                "wtmlMeth", params, "func.htm")).getValueAsString();
...
```

4.3 WTObjectRemoteAccess class

The `WTObjectRemoteAccess class` contains methods for remote access to a WebTransactions session. It enables data exchange and the remote implementation of methods and functions. To represent data on the Java page objects of the `WTObject` class are used. The remote session is specified by an object of the `WTSession` class that contains all addressing information.

```
public class WTObjectRemoteAccess
```

4.3.1 Constructor

`WTObjectRemoteAccess` creates a new `WTObjectRemoteAccess` object. The exception is described in the section “Exceptions” on page 41.

```
WTObjectRemoteAccess(WTSession wtSession) throws
          WTSessionNotAttachedException
```

wtSession
: Current `WTSession` object, which points to a WebTransactions session.

Example

```
WTObjectRemoteAccess remote_obj=new WTObjectRemoteAccess(myApp);
```

4.3.2 Methods

Please note that the following methods can only be executed correctly if a connection is open to a WebTransactions session. You can open a connection using the `open` or `attach` method of a `WTSession`; see also section “WTSession class” on page 30.

4.3.2.1 createObject method

The `createObject` method creates a new object in the remote WebTransactions session and returns this object as a `WTObject` object. If the method call fails, the appropriate exception is triggered; see section “Exceptions” on page 54.

```
WTObject createObject(String name, String constructor [,WTObject[] parameters]
          [,String codebase]) throws WTSessionNotAttachedException,
          WTSessionConnectionException, WTXMLParserException,
          WTNoXMLException
```

name Name of the new object.

constructor
Name of the remote constructor with which the object is to be created.

parameters
An array of `WTObjects` objects with the parameters of the constructor. If the constructor does not expect any parameters, this parameter is omitted.

codebase
In the remote WebTransactions application, *codebase* specifies the template in which the constructor is defined. This parameter can be omitted if the constructor is already known in the WebTransactions session, if the constructor is, for example, from the integrated classes, or if the constructor is assigned to the global system object `WT_SYSTEM`.

Example

The following example creates a function that calls the `receive` method without transferring the entire communication object as a return value.

```
...
WTObject[] params=new WTObject[1];
params[0]=new WTObject(WTObject.TYPE_STRING,"{this.receive();}");
remote_obj.createObject("WT_HOST.osd.silentReceive",
                        "Function", params);
```

4.3.2.2 download method

The `download` method transfers the specified object structure from a WebTransactions session to a `WTObject` object and returns this object. If the method call fails, the appropriate exception is triggered; see section “Exceptions” on page 54.

```
WTObject download(String remotePattern) throws
                      WTSessionNotAttachedException,
                      WTSessionConnectionException,
                      WTXMLParserException,
                      WTNoXMLException
```

remotePattern

Name or search string of the remote WebTransactions object. *remotePattern* can be used in the following ways:

remotePattern	**Meaning**
object	Any object with all attributes. If the attributes themselves are also objects, the conversion for these objects is continued recursively.
object`.`	Any object without attributes. The dot at the end means that no attributes of this object are converted.
object`..`	Any object with attributes but without subobjects, since none is specified between the two dots. The dot at the end means that no subattributes of subobjects in *object* are converted.
object`..`*value*	All attributes of the same name one level below an object. All attributes with the name *value* that are contained in objects directly under the data object *object*.
object1`|`*object2* *object*`..`*val1*`|`*val2*	Several objects or attributes under an object. The objects *object1* and *object2* or all attributes with the name *val1* or *val2* that are contained in objects directly under the data object *object*. The character `|` is a stronger link than the character `.`, which means that the following example applies: `WT_HOST|WT_SYSTEM.xyz` returns `WT_HOST.xyz` **and** `WT_SYSTEM.xyz`

Table 5: Possible specifications for the object structure to be loaded

A logical OR operand such as `"WT_SYSTEM | WT_HOST"` is not permitted on the top object level.

Example

```
WTObject wt_system=remote_obj.download("WT_SYSTEM.CGI");
```

4.3.2.3 invoke method

The `invoke` method calls a remote method or function in a WebTransactions session. `invoke` supplies the return value of the executed method or function as `WTObject` or the value `null` if the method or function does not return a value. If the method call fails, the appropriate exception is triggered; see section “Exceptions” on page 54.

```
WTObject invoke(String methodName [, WTObjects[] parameters]
                      [,String codebase]) throws
                      WTSessionNotAttachedException,
                      WTSessionConnectionException,
                      WTXMLParserException,
                      WTNoXMLException
```

methodName
: Name of the method or function, which can also be specified in the object hierarchy, e.g. `host.myMeth`.

parameters
: An array of `WTObject` objects with the parameters of the method or function. This parameter is optional, i.e. the method or function does not need parameters.

codebase
: In the remote WebTransactions application, *codebase* specifies the template in which the method or function to be executed is defined. The template is sought in accordance with the search sequence of WebTransactions; see also the WebTransactions manual “Concepts and Functions”.
This parameter can be omitted if the method or function is already known in the WebTransactions session, if the method is from the integrated classes or the function is a global WTML function, or if the method is assigned to the global system object `WT_SYSTEM`.

Example

```
...
WTObject[] params=new WTObject[2];
params[0]=myApp.getAttribute("E_02_001_80");
params[1]=myApp.getAttribute("E_06_005_85");
remote_obj.invoke("myMeth",params, "\temp1.htm");
```

4.3.2.4 upload method

The `upload` method transfers a `WTObject` object with the specified name to a remote WebTransactions session. `upload` operates additively, which means that objects or attributes that do not yet exist are created. Existing objects or attributes are overwritten.

```
Void upload(WTObject object, String remoteObjectName) throws
                      WTSessionNotAttachedException,
                      WTSessionConnectionException,
                      WTXMLParserException,
                      WTNoXMLException
```

object
: Object to be uploaded to the WebTransactions session.

remoteObjectName
: Name of the uploaded object in the WebTransactions session.

Example

```
remote_obj.upload(wt_system, "WT_SYSTEM");
```

4.3.3 Exceptions

The `invoke`, `createObject`, `download`, `upload` methods and the constructor of the `WTObjectRemoteAccess` object can trigger the following exceptions:

Exception	Meaning
`WTSessionNotAttachedException`	This exception is triggered when the current `WTSession` object does not have a connection to a WebTransactions session. You must first open a connection to a WebTransactions application using the `open` or `attach` method of the `WTSession` object before you can use the methods of the `WTObjectRemoteAccess` object.
`WTSessionConnectionException`	This exception is triggered when the connection parameters are not correct for a new WebTransactions session. Check the parameters *protocol*, *server*, *serverPort* or *WTScriptName*, which you specified in the `WTSession` constructor.
`WTXMLParserException`	This exception is triggered when a WebTransactions error occurs or when the parser has found an XML error in the current document. In the case of a WebTransactions error, the message text is part of the exception message.
`WTNoXMLException`	This exception is triggered when WebTransactions does not send an XML document as a response. The transmitted document is output as part of the exception message.

Table 6: Exceptions of `WTObjectRemoteAccess`

5 Example: Distributed WebTransactions application with WT_RPC

This chapter uses an example to illustrate how to use the interfaces described.

The example describes a filter application for the HTTP adapter using the example of the supplied WebTransactions client `WT_RPC` for `WTRemote`.

5.1 Implementation scenario

The example is based on two physically distant host applications `AppA` and `AppB`, which are to be integrated under a shared Web interface.

The host applications contain different formats `mA1` and `mA2` / `mB1` and `mB2`, which can be used to determine logically equivalent data. In order to process a transaction, the two formats must be run through as follows.

The terminal user logs on using the command `logon user,password` or `login u=user,p=password`. He or she then issues the command `mA1` or `mB1` to access the format for entering the search. A personnel number is entered for `mA1` or `mB1`, and the date of joining is output in the `mA2` or `mB2` format. The aim of the shared Web interface is to ensure uniform access to the date of joining for both formats.

Only a small section of the data presented in the format is relevant to the respective search:

AppA

```
mA1                                   11:53
--------------------------------------

PersNo.: 12345


--------------------------------------
Command SEARCH___
```

```
mA2                                   11:54
--------------------------------------

Name          : Smith
Sex           : 1
Birth date    : 07.05.52
Join date     : 01.01.85


--------------------------------------
Command END____
```

AppB

```
mB1                                   11:53
======================================

pNo : 12345_______
or
name: ___________


======================================
Action SEND____
```

```
mB2                                   11:54
======================================

date of

birth : 05/07/52
join  : 01/01/85


======================================
Action BYE_____
```

Figure 7: The different host applications and their formats

5.2 Technical concept

To enable uniform access to both host applications, an API is defined to process the transactions. WTScript functions are implemented, which perform the dialog with the respective host application. Since the mainframes are in different locations, the data is extensive, and the formats are very different, it is advisable to distribute this application and thereby permit the data to be presummarized. For both hosts, the methods `logon(`*user*`,`*password*`)` and `date_of_joining(`*pers_no*`)` should be made available as WebTransactions applications on different Web*Transaction* computers that are physically close to the respective host, in order to carry out the appropriate tasks. These WebTransactions applications could also be referred to as *mainframe drivers*. The methods of these WebTransactions applications are called by a third WebTransactions integration application and presented in the Web.

Another possible use of these *mainframe drivers* is that they enable a reduced view of the application (only searches on the basis of the date of joining) to be provided for utilization in other applications.

5.3 Implementation of integration application

The search in both applications is implemented as a distributed WebTransactions application. One of the WebTransactions applications `WTSrvA` or `WTSrvB` accesses the `AppA` or `AppB` application directly and provides the functionality as the functions `logon(`*user*`,`*password*`)` and `date_of_joining(`*pers_no*`)`. An WebTransactions integration application `WTInt` accesses these functions via the `WT_RPC` class:

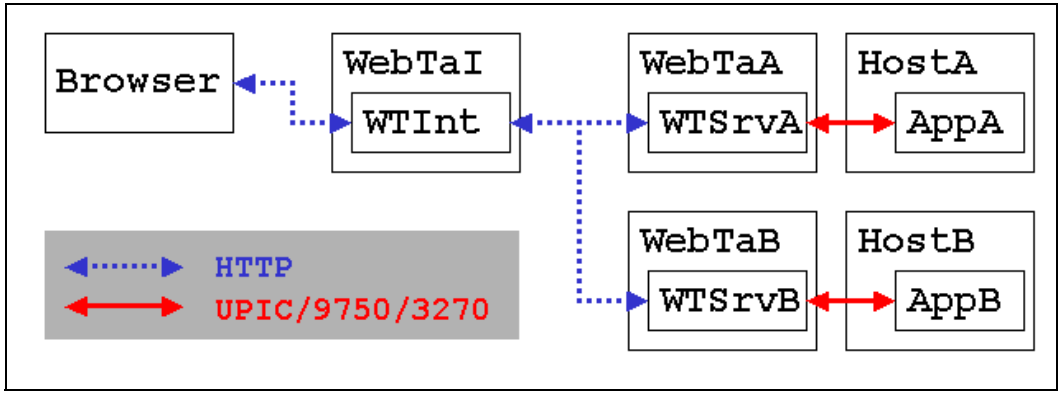

Figure 8: The integration application `WTInt`

WebTransactions applications

The WebTransactions applications `WTSrvA` and `WTSrvB` each contain a WTML document `accHost.htm`, in which the access functions `logon` and `join` are implemented. The documents may be slightly different, as the host applications are not identical. The template for the `AppA` host application could look as follows, for example:

```
function logon(user, password)
{
  host = new WT_Communication("appA");
  host_system = host.WT_SYSTEM;
  host_system.HOST_NAME = "HostA";
  host_system.SYM_DEST = "AppA";
  host.open("OSD");
  host.receive();
  host.E_1_1.Value = 'logon ' + user + ',' + password;
  host.send();
  host.receive();
  return (E_1_1.Value == 'logged in');
}
function date_of_joining(pers_no)
{
  WT_SYSTEM.ERROR = '';
  host = WT_HOST.appA;
  host.E_8_10.Value = 'GOTO mA1';
  host.send();
  host.receive();
  host.E_3_10.Value = persNr;
  host.send();
  host.receive();
  return (WT_SYSTEM.ERROR ? false : E_6_12.Value);
}
```

The `logon` function creates a new communication object and uses the OSD host adapter to open a connection with the host application `AppA` on computer `HostA`. The first format is displayed and the login data is entered. In the event of a positive acknowledgment, the function returns the value `true`, otherwise it returns `false`.

The `join` function enters the command `"GOTO mA1"`, in order to navigate to the search format. The number assigned as a parameter is entered in the field for the personnel number, and the search is performed. If no communication error occurs, the value of the date field is returned.

The functions implement a particular technical logic. As well as being used by the WebTransactions integration application, they can also be used locally to process certain tasks.

WebTransactions integration application

The WebTransactions integration application creates the HTML interface for the browser. It determines the data by accessing the remote WebTransactions applications. Access is processed via the HTTP host adapter with the aid of the `WT_RPC` class.

Two `WT_RPC` objects `appA` and `appB` are first created for the connection to the remote host applications. References are stored in the system object for subsequent dialog steps. The methods `logon` and `join` are attached using the `addMethod` method. The remote function can now be executed using the method call `appA.logon`, for example.

The closing HTML format enables the search in one of the two remote applications:

```
<wtInclude name="wtRPC">
<wtOnCreateScript>
  if(! WT_SYSTEM._appA)
  {
    WT_SYSTEM._appA = appA =
      new WT_RPC('WebTaA/cgi-bin/WTPublish.exe', '/home/WTSrvA');
    appA.addMethod('logon', 'accHost');
    appA.addMethod('join', 'accHost');
    WT_SYSTEM._appB = appB =
      new WT_RPC('WebTaB/scripts/WTPublish.exe', 'D:/WTSrvB');
    appB.addMethod('logon', 'accHost');
    appB.addMethod('join', 'accHost');
    appA.logon('testuser', '123');
    appB.logon('testuser', '123');
  }
  else
  {
    appA = WT_SYSTEM._appA;
    appB = WT_SYSTEM._appB;
  }
</wtOnCreateScript>
The date of joining is
##WT_POSTED.SEARCH == "A" ? appA.join(WT_POSTED.PNUM)
                         : appB.join(WT_POSTED.PNUM) #
<wtDataForm>
  Search for date of joining for personnel number
  <INPUT TYPE="TEXT" NAME="PNUM"> in the application
  <INPUT TYPE="SUBMIT" NAME="SEARCH" VALUE="A"> or
  <INPUT TYPE="SUBMIT" NAME="SEARCH" VALUE="B">
</wtDataForm>
```

6 Appendix: The WT_REMOTE interface

This appendix describes the `WT_REMOTE` interface. `WT_REMOTE` is an interface between WebTransactions applications and any clients, enabling controlled access to WebTransactions applications.

The information presented here is particularly relevant to accesses from remote clients. Predefined class libraries, such as the `WT` classes for Java clients and `WT_RPC` for WebTransactions clients, are available for the most common client applications, Java applets, and other WebTransactions applications. These class libraries are discussed in more detail in previous sections of this manual.

6.1 Introduction

Up to now, there were only two ways to access WebTransactions via a Web browser, namely using synchronized and non synchronized dialog. Both methods create HTML output that can be displayed directly by the browser.

The `WT_REMOTE` interface provides a new method of requesting the services of a WebTransactions session via the Web. In contrast to the former methods, the purpose of the new method is to:

- call services within a WebTransactions session
- transfer data to or from a WebTransactions session

The transmitted data is not limited to HTML data that can be presented by the browser, rather can also include any structured data, which is exchanged in the form of XML documents.

The following sections describe the methods of the `WT_REMOTE` interface, the format of the HTTP messages in which these coded methods must be transmitted, and finally the XML document types used to encode the methods within the HTTP messages. This interface can therefore be used not only by the supplied classes for Java applets and WebTransactions clients, but also by any other applications.

6.2 WT_REMOTE methods

This section provides an overview of the various methods offered by `WT_REMOTE`. The table below lists these methods and their purpose:

Method	Meaning
`START_SESSION`	Starts a new session and returns the session parameters for subsequent requests.
`EXIT_SESSION`	Closes the session.
`PROCESS_COMMANDS`	Transfers data to or from a WebTransactions session, creates objects, or calls methods.

Table 7: WT_REMOTE methods

Please note that there is no `WT_REMOTE` method for executing a WTML document. Such documents can be executed by means of a non synchronized dialog step, whereby the desired document is specified as `WT_ASYNC_PAGE`. These asynchronous calls can be mixed as desired with `WT_REMOTE` calls. A detailed description of how the HTTP messages must be structured for the individual methods can be found in the section “Request messages without data part” on page 67” and section “Request messages with control part and data part” on page 69”.

6.2.1 START_SESSION method

This method is used exclusively to start a WebTransactions session.

As the response message, this method supplies an XML document containing the session parameters for subsequent access to this session (see also section “Response message for START_SESSION” on page 87).

6.2.2 EXIT_SESSION method

This method closes the specified session and as a response message returns an XML document with an empty `response` element as confirmation (see also section “Response message for EXIT_SESSION” on page 87). `EXIT_SESSION` can only be used when a session is active, since the control part of the request message must contain name/value pairs for the session ID and signature. A data part for this method is unnecessary and may therefore be ignored.

6.2.3 PROCESS_COMMANDS method

This method transmits data to or from a WebTransactions session, creates an object in the WebTransactions session using a constructor call, or calls a WTScript method of the WebTransactions session. Various actions, which can be executed by means of a request, are available for performing these tasks:

- `data`, `uploadData`
  Upload data to the WebTransactions session.
- `downloadData`
  Download data from the WebTransactions session.
- `createObject`
  Create objects in the WebTransactions session.
- `callMethod`
  Call WTScript methods of the WebTransactions session.

A number of these actions can be performed in a request message using the `PROCESS_COMMANDS` method. The request message can be addressed to an active session or can be executed in a separate session; see also the following section.

One `data` element is returned in the XML document of the response message for each `downloadData`, `createObject` and `callMethod` element in the request message. `data` and `uploadData` elements in the request message are ignored in the response message. In addition, the response message may contain an `error` element for each error that occurred (see also section “Response message for PROCESS_COMMANDS” on page 88).

6.3 Single-step and multi-step transactions

A client can access an active WebTransactions session via `WT_REMOTE` by specifying his or her session ID during the access procedure. A WebTransactions session can also be started independently by the client access. The first case always involves multi-step transactions, while the second case allows both single-step and multi-step transactions.

6.3.1 Single-step transactions

In a single-step transaction, a WebTransactions session is started to execute a request message. The actions specified in the message are executed and the session is then closed again. All of this occurs with a single client access to `WT_REMOTE` using the `PROCESS_COMMANDS` method (see section “PROCESS_COMMANDS method” on page 63).

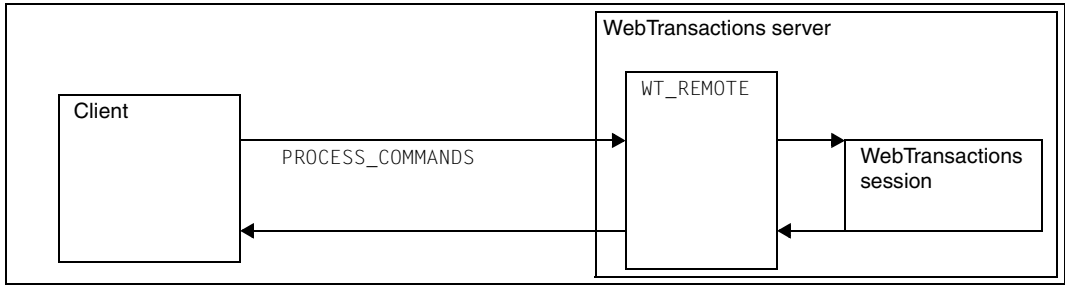

Figure 9: Single-step transaction

6.3.2 Multi-step transactions

There are two possible scenarios for a multi-step transaction. In the first case, a WebTransactions session is started explicitly by the `WT_REMOTE` access `START_SESSION`, several client accesses are then performed with `PROCESS_COMMANDS`, and the session is finally closed with `EXIT_SESSION`.

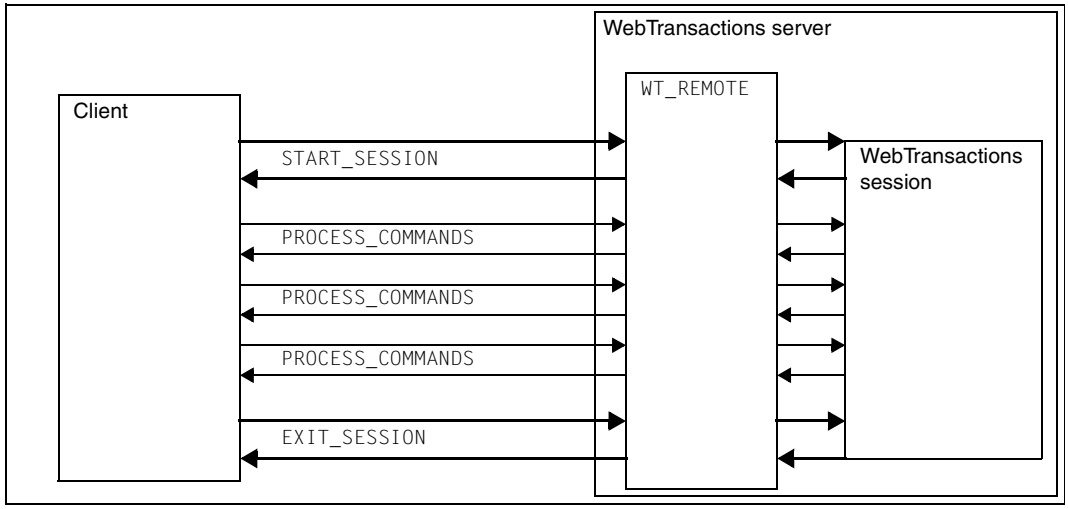

Figure 10: Multi-step transaction (started by WebTransactions)

In the second case, a client can address an active WebTransactions session (e.g. an applet is started with a dynamic page and activates itself in the same session). This is achieved by specifying `PROCESS_COMMANDS` with the appropriate session parameters.

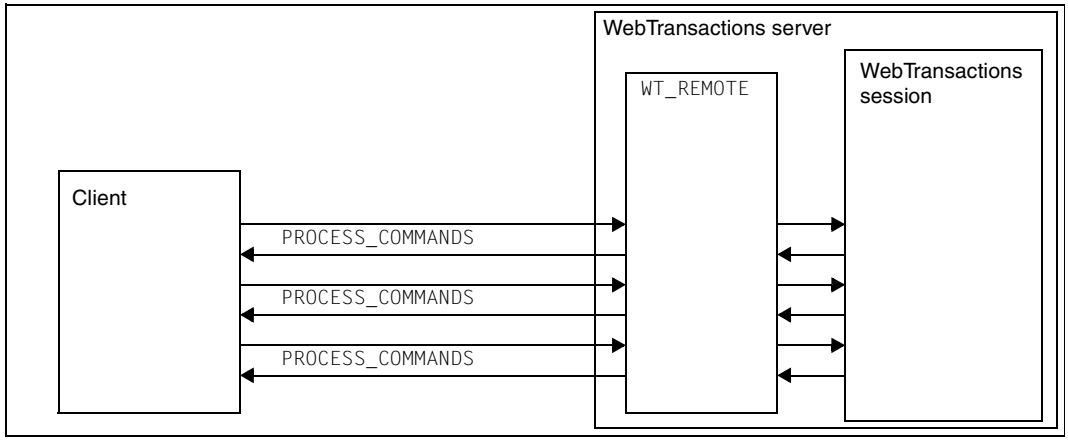

Figure 11: Multi-step transaction (started by another client)

6.4 Structure of request messages for WT_REMOTE

Communication between the client and the `WT_REMOTE` interface comprises a sequence of request messages from the client and response messages from the `WT_REMOTE` interface. The following sections describe the structure of these messages.

The requests described here are executed in the context of `WT_REMOTE`. This means that there is a set of global variables available exclusively for `WT_REMOTE` requests. These variables are not deleted automatically and are therefore available in a number of requests. The `WT_SYSTEM` and `WT_HOST` objects are used in conjunction with the synchronous and asynchronous accesses to the corresponding WebTransactions session.

The HTTP requests to WebTransactions primarily comprise a control part and an optional data part. If both parts are present, you use a HTTP-POST message of the mime type `multipart/mixed`. The messages are always structured in this way for the `PROCESS_COMMANDS` method.
If the data part is missing (with the methods `START_SESSION` and `EXIT_SESSION`), a single `POST` message with mime type `application/x-www-form-urlencoded` or a `GET` message can be used.

The following text provides some HTTP messages. To demonstrate the logical structure of the messages consisting of HTTP header and body more clearly, the precision of a hexadecimal representation has been dropped. For the HTTP protocol the line feeds are important:

– every line of the header ends with the characters carriage return and line feed (two characters).

– the HTTP header ends with an additional blank line (carriage return and line feed).

Therefore the line feed characters explicitly are displayed as [CR] [LF].

6.4.1 Request messages without data part

If the data part is missing, a POST message is basically structured as following:

```
POST /cgi-bin/WTPublish.exe HTTP/1.0 [CR] [LF]
Content-type: application/x-www-form-urlencoded [CR] [LF]
Content-length: Length [CR] [LF]
[CR] [LF]
. . . control part of message . . .
```

In contrast, a GET message has the following structure:

```
http://server/cgi-bin/WTPublish.exe?<control part of message>
```

This structure is illustrated below using some examples of the `START_SESSION` and `EXIT_SESSION` methods.

Request messages for START_SESSION

The path part of the URL for WebTransactions after the `WTPublish` program must contain the word `startup`. This method could then be called as follows, for example:

- `WT_REMOTE` method `START_SESSION` and HTTP method `POST`:

```
POST /cgi-bin/WTPublish.exe/startup HTTP/1.0 [CR] [LF]
Content-type:application/x-www-form-urlencoded [CR] [LF]
Content-length:60 [CR] [LF]
[CR] [LF]
WT_REMOTE=START_SESSION&WT_SYSTEM_BASEDIR=base-directory
```

  The length 60 in `Content-length` is an example. The length depends on the precise length of the message and the values for the base directory, sessionid etc. That applies to all other examples.

- `WT_REMOTE` method `START_SESSION` and HTTP method `GET`:

```
http://server/cgi-bin/WTPublish.exe/startup?\
WT_REMOTE=START_SESSION&WT_SYSTEM_BASEDIR=base-directory
```

Request messages for EXIT_SESSION

- `WT_REMOTE` method `EXIT_SESSION` and HTTP method `POST`:

  ```
  POST /cgi-bin/WTPublish.exe HTTP/1.0 [CR] [LF]
  Content-type:application/x-www-form-urlencoded [CR] [LF]
  Content-length:130 [CR] [LF]
  [CR] [LF]
  WT_REMOTE=EXIT_SESSION&WT_SYSTEM_BASEDIR=base-directory&\
  WT_SYSTEM_SESSION=sitzungs-id&WT_SYSTEM_SIGNATURE=signature
  ```

- `WT_REMOTE` method `EXIT_SESSION` and HTTP method `GET`:

  ```
  http://server/cgi-bin/WTPublish.exe?\
  WT_REMOTE=EXIT_SESSION&WT_SYSTEM_BASEDIR=base-directory&\
  WT_SYSTEM_SESSION=session-id&WT_SYSTEM_SIGNATURE=signature
  ```

A detailed description of the control part of the message can be found in section “Control part of the HTTP message” on page 70.

6.4.2 Request messages with control part and data part

This type of request message is used for the `WT_REMOTE` method `PROCESS_COMMANDS`, because this method always requires parameters which must be specified in the data part.

The control part has the mime type `application/x-www-form-urlencoded` and specifies the selected WebTransactions session. The data part has the mime type `text/xml` and contains the data to be processed in the current request.

Since both the control part and the data part cannot accept any arbitrary values, the message boundary is strictly defined by the character string `<<'<<"<<42>>">>'>>`. Such a HTTP message is therefore basically structured as follows:

```
POST /cgi-bin/WTPublish exe HTTP/1.0 [CR] [LF]
Content-type: multipart/mixed; boundary=<<'<<"<<42>>">>'>> [CR] [LF]
Content-length: 270 [CR] [LF]
[CR] [LF]
--<<'<<"<<42>>">>'>> [CR] [LF]
Content-type: application/x-www-form-urlencoded [CR] [LF]
[CR] [LF]
. . .control part of message . . . [CR] [LF]

--<<'<<"<<42>>">>'>> [CR] [LF]
Content-type: text/xml [CR] [LF]
[CR] [LF]
. . .data part of message . . .
```

This message structure describes a multi-step transaction. In a single-step transaction, whereby the `PROCESS_COMMANDS` method creates a separate WebTransactions session solely for the execution of the `WT_REMOTE` method, the keyword `startup` must also be specified after the `WTPublish` program:

```
POST /cgi-bin/WTPublish exe/startup HTTP/1.0 [CR] [LF]
Content-type: multipart/mixed; boundary=<<'<<"<<42>>">>'>> [CR] [LF]
Content-length: 270 [CR] [LF]
[CR] [LF]
--<<'<<"<<42>>">>'>> [CR] [LF]
Content-type: application/x-www-form-urlencoded [CR] [LF]
[CR] [LF]
. . .control part of message . . . [CR] [LF]

--<<'<<"<<42>>">>'>> [CR] [LF]
Content-type: text/xml [CR] [LF]
[CR] [LF]
. . .data part of message . . .
```

6.4.3 Control part of the HTTP message

The control part of the HTTP message (mime type `application/x-www-form-urlencoded`) contains information on which WebTransactions application is to be addressed (base directory) and which `WT_REMOTE` method is to be executed (`START_SESSION`, `EXIT_SESSION` or `PROCESS_COMMANDS`; see also section "WT_REMOTE methods" on page 62). In addition, the control part can contain further information corresponding to that of a conventional, local WebTransactions session:

- application timeout
- user timeout
- language
- style

If an active session is addressed, i.e. in a multi-step transaction with `PROCESS_COMMANDS` or `EXIT_SESSION`, the control part must contain additionally:

- session ID
- signature

This information is encoded as a name/value pair in the form *Name*=*Value* and is linked with the character `&`. The syntax is shown below:

```
  WT_SYSTEM_BASEDIR=base-directory
 &WT_REMOTE={START_SESSION|EXIT_SESSION|PROCESS_COMMANDS}
[&WT_SYSTEM_TIMEOUT_APPLICATION=timeoutApplication]
[&WT_SYSTEM_TIMEOUT_USER=timeoutUser]
[&WT_SYSTEM_LANGUAGE=language]
[&WT_SYSTEM_STYLE=style]
[&WT_SYSTEM_SESSION=session-id
 &WT_SYSTEM_SIGNATURE=signature]
```

The sequence of name/value pairs is arbitrary within the message body.

Example

```
POST /cgi-bin/WTPublish exe HTTP/1.0 [CR] [LF]
Content-type: multipart/mixed; boundary=<<'<<"<<42>>">>'>> [CR] [LF]
Content-length: 270 [CR] [LF]
[CR] [LF]
--<<'<<"<<42>>">>'>> [CR] [LF]
Content-type: application/x-www-form-urlencoded [CR] [LF]
[CR] [LF]
WT_SYSTEM_BASEDIR=base-directory&WT_REMOTE=PROCESS_COMMANDS\
&WT_SYSTEM_SESSION=sessionID&WT_SYSTEM_SIGNATURE=signature [CR] [LF]
--<<'<<"<<42>>">>'>> [CR] [LF]
Content-type: text/xml [CR] [LF]
[CR] [LF]
. . . data part of message . . .
```

6.4.4 Data part of the HTTP message

The data part of the HTTP message, if it is present, is of the mime type `text/xml` and contains further information on how the specified `WT_REMOTE` method is to be executed. It contains the parameters of the `WT_REMOTE` method, so to speak. These are encoded as an XML document (see section “XML documents for request messages” on page 73 for the syntax of this XML document).

Example 1

In this example, a complete HTTP message is specified which calls the `eval()` function in a WebTransactions session. In this case, the message is directed to an active session, i.e. the `SESSION` and `SIGNATURE` parameters must also be specified:

```
POST /cgi-bin/WTPublish exe HTTP/1.0 [CR] [LF]
Content-type: multipart/mixed; boundary=<<'<<"<<42>>">>'>> [CR] [LF]
Content-length: 270 [CR] [LF]
[CR] [LF]
--<<'<<"<<42>>">>'>> [CR] [LF]
Content-type: application/x-www-form-urlencoded [CR] [LF]
[CR] [LF]
WT_SYSTEM_BASEDIR=base-directory&WT_REMOTE=PROCESS_COMMANDS\
&WT_SYSTEM_SESSION=sessionID&WT_SYSTEM_SIGNATURE=signature [CR] [LF]
--<<'<<"<<42>>">>'>> [CR] [LF]
Content-type: text/xml [CR] [LF]
[CR] [LF]
<request>
<callMethod name="eval">
<string name="0">2*21</string>
</callMethod>
</request>
```

Example 2

In this example, the request from Example 1 is formulated as a single-step transaction. This means that a separate WebTransactions session is started to execute the function:

```
POST /cgi-bin/WTPublish exe/startup HTTP/1.0 [CR] [LF]
Content-type: multipart/mixed; boundary=<<'<<"<<42>>">>'>> [CR] [LF]
Content-length: 270 [CR] [LF]
[CR] [LF]
--<<'<<"<<42>>">>'>> [CR] [LF]
Content-type: application/x-www-form-urlencoded [CR] [LF]
[CR] [LF]
WT_SYSTEM_BASEDIR=base-directory&WT_REMOTE=PROCESS_COMMANDS [CR] [LF]
--<<'<<"<<42>>">>'>> [CR] [LF]
Content-type: text/xml [CR] [LF]
[CR] [LF]
<request>
<callMethod name="eval">
<string name="0">2*21</string>
</callMethod>
</request>
```

6.5 XML documents for request messages

This section describes the structure of the XML documents in request messages, i.e. the data part of multi-part request messages.

In the following sections, DTDs (Document Type Definitions) are used to describe the permitted syntax of these XML documents. The structure of such DTDs is therefore described here first using a brief example:

```
<!ELEMENT callMethod                        ((undefined | number | boolean |
                                            string | object | function)*)>
<!ELEMENT number                            (#PCDATA)>
<!ATTLIST callMethod    name        CDATA   #REQUIRED
                        codeBase    CDATA   #IMPLIED>
<!ATTLIST number        name        CDATA   #REQUIRED>
```

These four entries of a DTD describe the structure of two elements, namely `callMethod` and `number`. The line `<!ELEMENT callMethod ...>` defines that the `callMethod` elements can contain any number (`*`) of subelements, i.e. the subelements `undefined`, `number`, `boolean`, `string`, `object` or `function`. On the other hand, the line `<!ATTLIST callMethod ...>` defines which attributes must (`#REQUIRED`) or can (`#IMPLIED`) have a `callMethod` element. In the case of `callMethod`, the `name` attribute must be specified, whereas the `codeBase` attribute is optional. The `number` element must contain a `name` attribute and free text (`#PCDATA`) which specifies the value of the element.

Example

```
<callMethod   name="eval">
   <string   name="0">
      2*21
   </string>
</callMethod>
```

This XML document satisfies the syntax described with the above DTD. It describes the call of the `eval` method with one parameter. The name of the parameter is `"0"`, i.e. it is the first parameter. The parameter has the value `"2*21"`.

6.5.1 The structure of the XML document (DTDrequest)

The data part of a request message comprises an XML document which is structured in accordance with the following DTD, hereafter referred to as `DTDrequest`:

```
<!ELEMENT request                        ((data | uploadData | downloadData
                                          |createObject | callMethod)*)>
<!ELEMENT data                           ((undefined | number | boolean |
                                          string | object | function)*)>
<!ELEMENT uploadData                     ((undefined | number | boolean |
                                          string | object | function)*)>
<!ELEMENT downloadData                   EMPTY>
<!ELEMENT createObject                   ((undefined | number | boolean |
                                          string | object | function)*)>
<!ELEMENT callMethod                     ((undefined | number | boolean |
                                          string | object | function)*)>
<!ELEMENT undefined                      EMPTY>
<!ELEMENT number                         (#PCDATA)>
<!ELEMENT boolean                        (#PCDATA)>
<!ELEMENT string                         (#PCDATA)>
<!ELEMENT object                         (#PCDATA? (undefined | number |
                                          boolean | string | object |
                                          function)*)>
<!ELEMENT function                       EMPTY>
<!ATTLIST downloadData  name        CDATA  #REQUIRED>
<!ATTLIST createObject  name        CDATA  #REQUIRED
                        constructor CDATA  #REQUIRED
                        codeBase    CDATA  #IMPLIED>
<!ATTLIST callMethod    name        CDATA  #REQUIRED
                        codeBase    CDATA  #IMPLIED>
<!ATTLIST undefined     name        CDATA  #REQUIRED>
<!ATTLIST number        name        CDATA  #REQUIRED>
<!ATTLIST boolean       name        CDATA  #REQUIRED>
<!ATTLIST string        name        CDATA  #REQUIRED>
<!ATTLIST object        name        CDATA  #REQUIRED
                        class       CDATA  #IMPLIED
                        reference   CDATA  #IMPLIED>
```

The root element `request` of the XML document includes the elements that define the actions:

`data`	Transfer data to a WebTransactions application
`uploadData`	Upload data to a WebTransactions application
`downloadData`	Download data from a WebTransactions application
`createObject`	Call a constructor in a WebTransactions application
`callMethod`	Call a method in a WebTransactions application

If only one of these actions is to be executed in a HTTP message, the respective element can also be used as the root element of the XML document without specifying the `request` element.

Example

In this example, the `request` element contains three subelements `data`, `downloadData` and `callMethod`.

```
<request>
   <data>
      <number name="answer">42</number>
   </data>
   <downloadData    name="WT_HOST"/>
   <callMethod   name="myMethod" codeBase="myTemplate.htm">
      <number   name="0">
         42
      </number>
      <number   name="1">
         24
      </number>
   </callMethod>
</request>
```

The structure of the possible subelements of `request` is described in more detail in the following sections.

6.5.2 Structure of the data and uploadData elements (DTDdata)

The elements `data` and `uploadData` and their substructures are equivalent elements for transferring data to the data area of the remote WebTransactions session. The methods operate additively, i.e. if an object does not yet exist, it is created; if it already exists, additional attributes are created or existing attributes are modified. The syntax of both elements is identical and corresponds to the following DTD (hereafter referred to as `DTDdata`):

```
<!ELEMENT data                          ((undefined | number | boolean |
                                        string | object | function)*)>
<!ELEMENT undefined                     EMPTY>
<!ELEMENT number                        (#PCDATA)>
<!ELEMENT boolean                       (#PCDATA)>
<!ELEMENT string                        (#PCDATA)>
<!ELEMENT object                        (#PCDATA? (undefined | number |
                                        boolean | string | object |
                                        function)*)>
<!ELEMENT function                      EMPTY>
<!ATTLIST undefined   name       CDATA  #REQUIRED>
<!ATTLIST number      name       CDATA  #REQUIRED>
<!ATTLIST boolean     name       CDATA  #REQUIRED>
<!ATTLIST string      name       CDATA  #REQUIRED>
<!ATTLIST object      name       CDATA  #REQUIRED
                      class      CDATA  #IMPLIED
                      reference  CDATA  #IMPLIED>
<!ATTLIST function    name       CDATA  #REQUIRED>
```

The root element `data` (or `uploadData`) of this DTD contains an arbitrary sequence of the elements `undefined`, `number`, `boolean`, `string`, `object` and `function`. These element types correspond to the WTScript data types of the same names. Each element has a mandatory `name` attribute which defines the name of the variable or of the variable part.

The `undefined` and `function` elements are always empty. The elements `number`, `boolean` and `string` contain `PCDATA` (parsed character data) character strings. The value of the elements is specified as text in these `PCDATA` parts, i.e. as numeric literals (e.g. `-0.717273E-42`) for `number` values, as the literals `true` and `false` for `boolean` values, and as any character strings for `string` values.

The `object` element can have a `class` attribute which specifies the class of the object presented. In this case, the element describes the complete structure of the object. In addition, it can have a `PCDATA` part as the first subelement which specifies the value of a `Number`, `Boolean` or `String` object. As with simple data types, this text must then be an appropriate literal.

After the `PCDATA` part, the `object` element can have any sequence of nested elements which represent the attributes or methods of the object.

If an object has already been described within the document and if a further reference to this object occurs in the same document, this is indicated by the `reference` attribute. This attribute contains the absolute name of the referenced object. The contained attributes are identical to those of the referenced object and are not listed a second time in the XML document.

The following example shows an extract from `WT_SYSTEM` and its representation in `DTDdata`:

```
WT_SYSTEM   (type object, class Object)
   BASEDIR    (type string, value "/home/WebTA")
   CGI      (type object, class Object)
      HTTP_USER_AGENT   (type string, value "Mozilla/4.0")
      REMOTE_HOST   (type string, value "pcfritz")
   FORMAT   (type string, value "wtstart")
   _myArray   (type object, class Array)
      0      (type string, value "the answer is ")
      1      (type number, value 42)
      2      (type boolean, value true)
      att   (type string, value "problem is the question")
      top   (type object, class Object, reference to WT_SYSTEM.CGI)
   _myStringObject   (type object, class String, value "another string")
      att1   (type object, class Boolean, value false)
      att2   (type boolean, value true)
   _myMethod   (type function)
```

If this data structure is presented as an XML document in accordance with `DTDdata`, it looks as follows (the line breaks and indents do not correspond to the actual conversion, rather are inserted for legibility purposes):

```
<data>
   <object   name="WT_SYSTEM"   class="Object" >
      <string   name="BASEDIR">
         /home/WebTA
      </string>
      <object   name="CGI"   class="Object">
         <string   name="HTTP_USER_AGENT">
           Mozilla/4.0
         </string>
         <string   name="REMOTE_HOST">
           pcfritz
```

```
            </string>
         </object>
         <string   name="FORMAT">
            wtstart
         </string>
         <object   name="_myArray" class="Array">
            <string   name="0">
              the answer is
            </string>
            <number   name="1">
              42
            </number>
            <boolean   name="2">
              true
            </boolean>
            <string   name="att">
              problem is the question
            </string>
            <object   name="top" reference="WT_SYSTEM.CGI"/>
         </object>
         <object   name="_myStringObject"    class="String">
            another string
            <object   name="att1"   class="Boolean">
              false
            </object>
            <boolean   name="att2">
              true
            </boolean>
         </object>
         <function   name="_myMethod"/>
      </object>
   </data>
```

`uploadData` (or `data`) transfers the specified data to the addressed WebTransactions session. In the example, the specified values and attributes are transferred from `WT_SYSTEM`.

6.5.3 Structure of the downloadData element (DTDdownload)

The `downloadData` element specifies one or more objects that are to be downloaded from the remote WebTransactions session. The desired objects are returned in the response. The element is structured in accordance with the following DTD, hereafter referred to as `DTDdownload`:

```
<!ELEMENT donwloadData                    EMPTY>
<!ATTLIST downloadData    name    CDATA   #REQUIRED>
```

The `downloadData` element has a `name` attribute which specifies the objects or attributes to be downloaded, in accordance with the following syntax:

object	Any object. The data object *object* and all its attributes. If attributes are themselves objects, the conversion for these objects is continued recursively.
object.	Any object without attributes. The data object *object*. However, no attributes of this object are converted due to the dot at the end.
object..	Any object without subobjects. The data object *object* and all the attributes of this data object (since no specification is made between the two dots). The dot at the end ensures that no subattributes or subobjects in *object* are converted.
object..value	All attributes of the same name one level below an object. All attributes with the name *value* which are contained in objects directly under the data object *object*.
object1|object2 *object..val1|val2*	Several objects or attributes under an object. The objects *object1* and *object2* or all attributes with the name *val1* or *val2* which are contained in objects directly under the data object *object*. The character `|` is a stronger link than the character `.`, which means that the following example applies: `WT_HOST|WT_SYSTEM.xyz` returns `WT_HOST.xyz` **and** `WT_SYSTEM.xyz`

Example

This example queries the three session attributes `BASEDIR`, `SESSION` and `SIGNATURE`.

```
<request>
   <downloadData name="WT_SYSTEM.BASEDIR|SESSION|SIGNATURE"/>
</request>
```

The following response message, for example, could then be returned as a response to this request, in accordance with the DTD `DTDresponse` (see also section “XML documents in response messages” on page 86):

```
<response>
<data>
   <object name="WT_SYSTEM" class="Object">
      <string name="BASEDIR">C:/basedirs/context
      </string>
      <string name="SESSION">E-935516492-8494
      </string>
      <string name="SIGNATURE">159177851380710585
      </string>
   </object>
</data>
</response>
```

6.5.4 Structure of the callMethod element (DTDmethod)

The `callMethod` element specifies the necessary information for calling a method of the remote WebTransactions session. The corresponding method is executed and the result of the function is returned. The element is structured in accordance with the following DTD, hereafter referred to as `DTDmethod`:

```
<!ELEMENT callMethod                    ((undefined | number | boolean |
                                        string | object | function)*)>
<!ELEMENT undefined                     EMPTY>
<!ELEMENT number                        (#PCDATA)>
<!ELEMENT boolean                       (#PCDATA)>
<!ELEMENT string                        (#PCDATA)>
<!ELEMENT object                        (#PCDATA? (undefined | number |
                                        boolean | string |object |
                                        function)*)>
<!ELEMENT function                      EMPTY>
<!ATTLIST callMethod  name        CDATA  #REQUIRED
                      codeBase    CDATA  #IMPLIED>
<!ATTLIST undefined   name        CDATA  #REQUIRED>
<!ATTLIST number      name        CDATA  #REQUIRED>
<!ATTLIST boolean     name        CDATA  #REQUIRED>
<!ATTLIST string      name        CDATA  #REQUIRED>
<!ATTLIST object      name        CDATA  #REQUIRED
                      class       CDATA  #IMPLIED
                      reference   CDATA  #IMPLIED>
<!ATTLIST function    name        CDATA  #REQUIRED>
```

The root element `callMethod` has two attributes, namely `name` and `codeBase`. The `name` attribute must be specified and must define the absolute name of the method to be executed. A simple identifier is specified for a global function, and an object-specific method is represented by dot notation.

The `codeBase` attribute is optional and can be used to specify a template name. WebTransactions then searches for the template of this name using the standard search sequence (described in the WebTransactions manual “Concepts and Functions”). If the method is already known to the system (e.g. an integrated function or a method of an object), this attribute can be omitted.

The elements within the root element represent the method to be called. They can be of any type, i.e. `undefined`, `number`, `boolean`, `string`, `object` or `function`.

The respective `name` attribute of this element on the top level is set to an integer which specifies the index of the respective parameter in the parameter list (i.e. `'0'` for the first parameter, `'1'` for the second, etc.). This name is used for references to these objects within the parameters. These parameters are defined within the calling XML document, and the parameters are transferred to the method "by value", i.e. modifications to the parameters within the method have no effect in the calling XML document.

A `name` attribute must be specified for each lower level of the parameter definition so that well-formed structures of named attributes can be established.

Here, the definition of values and classes corresponds to that of the DTD `DTDdata`, described in section "Structure of the data and uploadData elements (DTDdata)" on page 76.

Examples:

- Calling a global function, e.g. `eval`:

  ```
  <callMethod name="eval">...</callMethod>
  ```

- Calling a user-defined function `myFunction`. In this case, the template (in the example, `MyFunctions.htm`) containing the function must also be specified:

  ```
  <callMethod name="myFunction" codeBase="MyFunctions.htm">
      ...
  </callMethod>
  ```

- Calling an object-specific method for an object of the integrated classes; in this case, the name of the method called must also contain the name of the calling object.

  `myComm` is a communication object, i.e. of the class `WT_Communication`

  ```
  <callMethod name="WT_HOST.myComm.send"/>
  ```

- Calling a user-defined method `myMethod` in a user-defined class. In this case, the template (in the example, `MyMethods.htm`) containing the method must also be specified:

  `myObject` is an object of the user-defined class

  ```
  <callMethod name="myObject.myMethod" codeBase="MyMethods.htm">
      ...
  </callMethod>
  ```

- The following example shows the simple call of a global function with one parameter:

```
<request>
   <callMethod name="eval">
      <string name="0">2*21</string>
   </callMethod>
</request>
```

And below is the response message:

```
<response>
   <data>
      <number name="">42
      </number>
   </data>
</response>
```

6.5.5 Structure of the createObject element (DTDcreate)

The `createObject` element creates a constructor call in the addressed remote WebTransactions session. The element is structured in accordance with the following DTD, hereafter referred to as `DTDcreate`:

```
<!ELEMENT createObject                    ((undefined | number | boolean |
                                          string | object | function)*)>
<!ELEMENT undefined                       EMPTY>
<!ELEMENT number                          (#PCDATA)>
<!ELEMENT boolean                         (#PCDATA)>
<!ELEMENT string                          (#PCDATA)>
<!ELEMENT object                          (#PCDATA? (undefined | number |
                                          boolean | string |object |
                                          function)*)>
<!ELEMENT function                        EMPTY>
<!ATTLIST createObject  name        CDATA  #REQUIRED
                        constructor CDATA  #REQUIRED
                        codeBase    CDATA  #IMPLIED>
<!ATTLIST undefined     name        CDATA  #REQUIRED>
<!ATTLIST number        name        CDATA  #REQUIRED>
<!ATTLIST boolean       name        CDATA  #REQUIRED>
<!ATTLIST string        name        CDATA  #REQUIRED>
<!ATTLIST object        name        CDATA  #REQUIRED
                        class       CDATA  #IMPLIED
                        reference   CDATA  #IMPLIED>
<!ATTLIST function      name        CDATA  #REQUIRED>
```

The `name` attribute of this element identifies the name of the new object. The element attribute `constructor` also available specifies the desired constructor function. Finally, the optional attribute `codeBase` can be used to specify a WTML document of the remote WebTransactions session, whose functions are provided before execution. The constructor and the methods of the class are then defined in this document.

```
<createObject name="myObj" constructor="myCons" codeBase="remoteTest.htm">
    constructor arguments
    ...
</createObject>
```

The elements within `createObject` specify the arguments of the constructor. The created object is returned in the form of an XML document as the result (see section “XML documents in response messages” on page 86).

Example

The following example creates an object of type `Date` with the value 31.12.1999:

```
<createObject name="myDate" constructor="Date">
<number   name="0">
   1999
</number>
<number   name="1">
   11
</number>
<number   name="2">
   31
</number>
</createObject>
```

The response message (see following section) could then look as follows, for example:

```
<response>
<data>
   <object name="myDate" class="Date">
   </object>
</data>
</response>
```

6.6 XML documents in response messages

All response messages to `WT_REMOTE` requests always have the mime format `text/xml`, except in the event of connection errors. The XML documents supplied correspond to the following DTD (`DTDresponse`):

```
<!ELEMENT response                    (data* error*)>
<!ELEMENT data                        ((undefined | number | boolean | string
                                      | object | function)*)>
<!ELEMENT undefined                   EMPTY>
<!ELEMENT number                      (#PCDATA)>
<!ELEMENT boolean                     (#PCDATA)>
<!ELEMENT string                      (#PCDATA)>
<!ELEMENT object                      (#PCDATA? (undefined | number | boolean
                                      |string | object | function)*)>
<!ELEMENT function                    EMPTY>
<!ELEMENT error                       (#PCDATA?)>
<!ATTLIST undefined  name       CDATA  #REQUIRED>
<!ATTLIST number     name       CDATA  #REQUIRED>
<!ATTLIST boolean    name       CDATA  #REQUIRED>
<!ATTLIST string     name       CDATA  #REQUIRED>
<!ATTLIST object     name       CDATA  #REQUIRED
                     class      CDATA  #IMPLIED
                     reference  CDATA  #IMPLIED>
<!ATTLIST function   name       CDATA  #REQUIRED>
<!ATTLIST error      document   CDATA  #IMPLIED
                     line       CDATA  #IMPLIED
                     column     CDATA  #IMPLIED
                     message    CDATA  #REQUIRED>
```

The root element can contain a sequence of `data` elements and a sequence of `error` elements. The structure of the `data` element corresponds to that of `DTDdata`, described on section “Structure of the data and uploadData elements (DTDdata)” on page 76. With this element, the result of a download, of the creation of an object, or of a method call is returned.

If errors occur during processing in the remote WebTransactions session, information on every error is transferred to the client as `error` elements. These `error` elements can contain the following information, if it can be determined:

– document (`document` attribute)
– line (`line` attribute)
– column (`column` attribute)

The error number is always returned in the `message` attribute, and the error text is contained in the `error` element as `PCDATA` (see also the second example in section “Response message for PROCESS_COMMANDS” on page 88).

6.6.1 Response message for START_SESSION

The XML document returned as the response to `START_SESSION` has the following form:

```
<response>
   <data>
      <object name="WT_SYSTEM" class="Object">
         <string name="SESSION">session-id</string>
         <string name="SIGNATURE">signature</string>
         <string name="BASEDIR">base-directory</string>
      </object>
   </data>
</response>
```

6.6.2 Response message for EXIT_SESSION

The response message for `EXIT_SESSION` comprises an empty `response` element as confirmation.

```
<response/>
```

6.6.3 Response message for PROCESS_COMMANDS

A `data` element is returned in the response message for each `downloadData`, `createObject` and `callMethod` action. The `data` and `uploadData` actions in the request message are ignored in the response message (see the first example below). Furthermore, the response message can contain an `error` element for each error that occurred (second example).

Examples

The following examples further illustrate the interaction of request and response messages, along with the examples already given for the request messages.

- This is an example of several requests in one message. One of the methods is `uploadData`, which does not generate a response:

```
<request>
   <createObject name="NewString" constructor="String">
      <string name="0">Hello world</string>
   </createObject>
   <uploadData>
      <string name="NewString">Hello world</string>
   </uploadData>
   <downloadData name="NewString"/>
</request>
```

  The associated response message could then look as follows, for example:

```
<response>
//response to createObject
   <data>
      <object name="NewString" class="String">Hello world
      </object>
   </data>
//response to downloadData
   <data>
      <string name="NewString">Hello world
      </string>
   </data>
</response>
```

- This is an example of an error message in the `error` element, for example a typing error in the function name:

```
<request>
   <callMethod name="evla">
      <string name="0">2*21</string>
   </callMethod>
</request>
```

The associated response message:

```
<response>
   <data>
      <undefined name=""/>
   </data>
   <error message="216">Error: unknown function - function "evla" not
   defined or not of type function; correct WTML document.
   </error>
</response>
```

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
Connection to openUTM Applications via UPIC
User Guide

WebTransactions
Connection to OSD Applications
User Guide

WebTransactions
Connection to MVS Applications
User Guide

WebTransactions
Access to Dynamic Web Contents
User Guide

WebTransactions
Web Frontend for Web Services
User Guide

Index

X