Volume 4

Hyperwave Programmer's Guide

Hyperwave Information Server

Version 4.1

HYPERWAVE
Organize your future

Hyperwave Programmer's Guide

Hyperwave
Information Management Inc.
2350 Mission College Blvd.
Santa Clara, CA 95054
USA
Tel: 1-408-982-8228
Fax: 1-408-982-8229
Email: info@hyperwave.com
http://www.hyperwave.com

Hyperwave
Information Management GmbH
Stefan-George-Ring 19
D-81929 Munich
Germany
Tel.: ++ 49 89 993074-0
Fax: ++ 49 89 993074-99
Email: info@hyperwave.de
http://www.hyperwave.de

Document Version 1.0

TABLE OF CONTENTS

1 CONFIGURING THE WAVEMASTER INTERFACE

This chapter describes PLACE, the language used to configure the appearance of the WaveMaster interface. Here you can find out how the functions in the default interface are programmed and in which file you can find which functions. Based on this information, you should be able to alter the interface to suit your needs.

1.1 PLACE

PLACE is a meta-HTML language, with which the appearance and function of WaveMaster's user interface can be configured. The name PLACE comes from the so-called *placeholders* used in the language. Placeholders transfer various types of information between the server and WaveMaster, including information about the currently accessed collection or cluster, about the Hyperwave Information Server WaveMaster is connected to, the users currently logged in to the server, etc. For a complete list of placeholders including a brief explanation of each, see the *Hyperwave Reference Guide*. It is on the Hyperwave CD and at `http://www.hyperwave.com`.

The PLACE language consists of standard language constructs such as if-statements, macro-statements and while-loops as well as a number of placeholders. This documentation is not fully sufficient in order to enable you to use all the possibilities of the PLACE language. To really know what you are doing, a basic knowledge of the server and its features are necessary, in particular a knowledge of what attributes are and how they are used. It is also very helpful if you have a good background in HTML.

1.1.1 PLACE TEMPLATES

A PLACE template is a file which contains HTML and PLACE statements and which is applied to objects which users access. Such a template could, for example, check if an error occurred and display the appropriate message, then if no error occurred proceed to display the appropriate buttons at the top of the page (for example identify and search buttons, and, if the user is identified, an "edit" button). Then it could check what type of object is being accessed at which point it would display a list of links if the object is a collection or display the text if it is a text object, etc. Lastly, it might display the user name of the current user and show links to the parent objects of the current object.

There are two ways of applying PLACE templates to documents on your server. In the first case, the template resides in the `wavemaster` directory of the user as which Hyperwave is running (usually `~hwsystem`), in which case it is applied by default to every document on the entire server. In the second case they are used to override the default template for specific documents. To find out how to apply a PLACE template only to specific objects see page 22.

When you install Hyperwave the `wavemaster` directory contains a file called `master.html` which is the default template for the server. This template already implements all the main features of Hyperwave, such as editing, user management, etc. Thus it is not necessary to learn how to use PLACE unless you would like to change WaveMaster's default configuration.

1.1.2 BASIC PLACE SYNTAX

PLACE statements are put into normal HTML files and are surrounded by HTML commands or normal text. All PLACE constructs are enclosed by two percentage symbols (`%%`) to distinguish them. A typical PLACE statement is for example `%%object.title%%.` When a document is retrieved, this PLACE statement is evaluated to the title of the current object. WaveMaster interprets the PLACE statements every time a document is retrieved.

Comments are made by starting the PLACE statement with a "#", e.g.

`%%# object is a collection %%`.

The comment ends at the next "`%%`".

1.1.2.1 FEATURES OF THE PLACE LANGUAGE

There are two constructs which can be used to alter the program flow in PLACE: "if" and "while" statements.

If-statements are used to branch off in the PLACE code according to whether particular conditions are fulfilled or not.

The syntax of if-statements is as follows:

`%%if` *expression*`%%`
[*text block or more PLACE*]
{`%%else%%`}
{[*text block or more PLACE*]}
`%%endif%%`

An *expression* is a comparison using one of the following operators:

`==`	is equal to
`!=`	is not equal to
`>`	is greater than
`<`	is less than
`>=`	is greater than or equal to
`<=`	is less than or equal to

such as `object.title != Help Collection`. These expressions can further be combined using AND (&&), OR (`||`) and NOT (`!`) operators, e.g. `!(coll.entry.get_attrib(PresentationHints) == "Hidden") && !(coll.entry.get_attrib(PresentationHints) == "CollectionHead")`. This expression is fulfilled if the current item in the current collection has as value of the **PresentationHints** attribute neither "Hidden" nor "CollectionHead".

If-statement examples:

```
%%if action == action.identify%%
Hyperwave Identification
%%endif%%
```

This displays the text "Hyperwave Identification" if the user has activated the identify function.

```
%%if collhead.get_attrib(TimeModified) != "" %%
%%collhead.get_attrib(TimeModified)%%
%%else%%
%%collhead.get_attrib(TimeCreated)%%
%%endif%%
```

This checks if the **TimeModified** attribute of the collection head of the current collection exists, and if so the value of this attribute is displayed. If not, the attribute **TimeCreated**, which always exists, is displayed.

The while-statement is used in PLACE to perform an action repeatedly until a certain condition is no longer fulfilled.

The syntax of while-statements is as follows:

```
%%while expression%%
[text block or more PLACE]
%%endwhile%%
```

In this case *expression* can also be as described above for the expression in the if-statement, but in most cases it isn't. For a while loop to work, the expression to be tested at the beginning of each loop must be one that changes so that it eventually becomes false or else the while loop will never end. Thus the only variables which are normally useful in while loops are those of the type Boolean. In PLACE, unlike in many programming languages, it is not possible to declare your own variables and use them as counters.

An example of a placeholder that could be tested at the beginning of a while loop is `cluster.next_entry`. This placeholder is used for example when displaying the contents of a cluster. You can use a while loop to go through the items contained in a cluster one by one and display each one, and when there are no more items, the system sets this placeholder to false, and the while loop is terminated.

1.1.2.2 STRUCTURING AND MODULARITY

Though it is possible to use one big file as a PLACE template, it is also possible to break it down into smaller files using the `include` statement. If you include a file in this way, it is exactly as if the contents of that file appeared on that spot in the file. The entry to make in the master file looks like the following:

`%%include "`*file name*`"%%`

where *file name* is the full file name with path relative to the `wavemaster` directory, e.g. `%%include "include/header.html"%%`.

If you upload the template to the server in order to apply it to specific documents (see page 22), any files you want to include in the template must also be uploaded to the server and can be included in the master file with the `hyperinclude` statement, i.e.

`%%hyperinclude "`*name*`"%%`

where *name* is the name or GOid of the text to be included, e.g.

`%%hyperinclude "mytemplate/body"%%`.

The PLACE code can be organized into modules which are called *macros*. Macros are like functions but somewhat simpler. They are bits of PLACE code which you assign a unique name to; and when that name appears anywhere in the remainder of the template, the macro is executed. Macros have to start with

`%%macro` *name*`%%`

and end with

`%%endmacro%%`

with HTML or PLACE in between these delimiters. Note that the macro itself must appear in the template at a point before it is called. Also note that macros can call other macros which in turn can call other macros for up to as many levels as you want.

Here is an example of a macro named `attributes`:

```
%%macro attributes%%

<HR>
<H1>%%ge:Attribute von, :Attributes of%% "%%object.title%%"</H1>
<HR>
<P>
<CENTER><TABLE BORDER=1>
%%while object.attributes.next_entry%%
<TR><TD>%%object.attributes.entry.key%%<TD>%%object.attributes.entry.value%%
%%endwhile%%
</TABLE></CENTER><P>

%%endmacro%%
```

This macro uses the currently set language preference to decide whether to display the text "Attributes of" in English or German, followed by the title of the current object. It then uses a while-loop to produce a table showing the attributes of the current object.

This macro is called by inserting `%%attributes%%` anywhere in the place code.

1.1.2.3 PROVIDING MULTILINGUALITY

With Hyperwave you can provide multilingual documents. If there are documents on your server in different languages (title attribute with language tags like en:, ge:, jp:), the WaveMaster interface should also be changeable according to the user's language preference (English and German texts are provided in the default distribution). There are two ways to program PLACE to display text in the language corresponding to the currently set language preference.

USING IF-STATEMENTS

The placeholder `%%session.language%%` holds the user's current language setting as a Hyperwave language string ("en" for English, "ge" for German, "sp" for Spanish, etc.). To show the text "user" in different languages using if-statements, you could use:

```
%%if session.language == "sp" %%
   Usario
%%else%%
   %%if session.language == "ge" %%
      Benutzer
   %%else%%
      User
   %%endif%%
%%endif%%
```

USING THE LANGUAGE CONSTRUCT

For short texts the method above might be awkward. For this purpose you can also use the special construct:

```
%%sp:Usario, ge:Benutzer, :User%%
```

This has the same effect as the statement above. As you can see, you can use a single colon to define a default text. The different language texts can be in arbitrary order. Note that the language texts may not include commas and percentage symbols.

The following language abbreviations are used by Hyperwave:

Language	Abbreviation
English	en
German	ge
French	fr
Italian	it
Spanish	sp
Japanese	jp

1.1.2.4 ACTIONS

Placeholders come in many types: there are placeholders that have to do with annotations, clusters, collections, the search function, the server itself, etc., just to name a few. A further type of placeholder are *actions*. Actions are the only type of placeholder which is used to actually perform an action instead of to communicate a value to WaveMaster. An action is a process with which certain information is retrieved from the server or altered in some way. To achieve this WaveMaster has to know about the actions and their parameters to communicate with the Hyperwave Information Server.

Actions are in the form **action.call.***, e.g. **action.call.identify**. They are always used in the form of anchors which the user can activate, e.g.

```
<A HREF="%%action.call.home%%">[text or image for the link]</A>
```

This would provide a link which the user could click on to go directly to his home collection.

When an action is called a special variable called **action** is set according to the action that was called. For example, if the help action was called **action** is set to **action.help** and if the identification action was called then it is set to **action.identify**. The **action** variable can then be used in expressions in if-statements as in the example below.

```
%%if action == action.identify%%
Hyperwave %%ge:Identifizierung, :Identification%%
%%endif%%
```

Here we can see that after the user calls the identify action, the word "Identification" is displayed according to the user's language preference.

Until now we have been discussing **predefined** actions. It is also possible for the programmer to create arbitrary, **self-defined** actions. This type of action is used to determine the appearance of the WaveMaster interface in specific situations. The difference here is that predefined actions perform an action that has directly to do with the server and self-defined actions perform actions which are important to the user but not important to the server. For example, the predefined action **action.call.search** starts an actual search on the server using information that the user has already entered. However, to cause a search form to appear where the user can fill in a search term and set other search preferences (an action which does not directly concern the server), you have to define an action yourself.

Self-defined actions are in the form **action.call(***ActionName***)**, where *ActionName* is arbitrary but is in the form `*.action` throughout the default PLACE templates. They are similar to predefined actions in that they are accessed the same way (using links to the action name) and when the action is accessed, the variable **action** is set to the action name. The difference is that for a self-defined action you have to use PLACE and HTML to get WaveMaster to do what you want it to do when the action is called.

A typical example of a self-defined action is to implement an "options" icon which the user can use to set his preferred language or get the current server status and user list. If the icon is clicked on, the options action (i.e. calling a page on the screen with information for the user) should be triggered. In the template file this is expressed as:

`<A HREF="%%action.call(options.action)%%">[`*Text or Image for the Link*`]</A>`

If the options icon is clicked on, the action called `options.action` is triggered and the PLACE template file is requested with the variable `action` set to `options.action`. Thus, the options page can be defined with an expression like:

```
%%if action == "options.action"%%
[HTML text for the options page]
%%endif%%
```

If, for example, you want to create an action called `search.action` which causes a form to appear where users can fill in their search preferences, you must

1. Choose a name for it (e.g. `search.action`).
2. Write an if-statement that carries out certain actions when **action** is set to this name, e.g.

   ```
   %%if action == "search.action"%%
   %%searchform%%
   %%endif%%
   ```

3. Create links to the action where you want the user to have the option of calling this action, e.g.

   ```
   <A HREF="%%action.call(search.action)%%">Start a search</A>
   ```

Note*: In the example in item 2 the body of the if-statement doesn't actually contain the PLACE/HTML statements used to produce the form but rather a macro* `searchform` *which contains the statements for the form is called. If you use a macro, you must write the macro before calling the action for the action to work.*

A number of self-defined actions are already implemented in the default distribution. It is however possible to define any other action you wish with PLACE.

1.1.3 TYPICAL APPLICATIONS OF PLACE CONSTRUCTS

The use of several placeholders and self-defined as well as predefined actions is explained here by showing several functions that can be implemented with them. The explanations below refer to the default PLACE templates (located in `$HOME/wavemaster/include/` and referenced by `$HOME/wavemaster/master.html`), which are distributed with the server. It gives you an idea what files, placeholders and actions are necessary to implement the functions in your own layout.

1.1.3.1 DISPLAYING AND EDITING ATTRIBUTES

DISPLAYING ATTRIBUTES

In the default PLACE templates, attributes of a given object are displayed by calling the macro `attributes` (contained in the file `attributes.html`) using a self-defined action. The attributes display can be defined using the following placeholders:

· `object.attributes.next_entry`

  This is a Boolean placeholder which moves the attributes pointer to the next attribute entry of the current object. It returns true if there is a next entry, otherwise false.

· `object.attributes.entry.key`

  This placeholder's value is the name of the attribute the attribute pointer is pointing to, e.g. `Author`.

· `object.attributes.entry.value`

  This placeholder's value is the value of the attribute the attribute pointer is pointing to.

· `object.attributes.entry`

  This placeholder can be used to display the key-value pair for an attribute (e.g. `Author=hwsystem`).

These placeholders are typically used in a while loop like the following:

```
%%while object.attributes.next_entry%%
  %%object.attributes.entry.key%%
  %%object.attributes.entry.value%%
%%endwhile%%
```

which simply goes through the list of attributes for an object one by one and displays each name with its corresponding value.

A further bit of useful information about a document can be retrieved using the placeholder `object.size`:

```
%%if object.size !=0%%
      Document size (in Bytes): %%object.size%%
%%endif%%
```

EDITING ATTRIBUTES

The file `editattributes.html` contains by default a macro called `editattributes` which displays a form where users can edit the attributes of the object they are browsing. The form contains a `<FORM ACTION>` field which calls the predefined action `action.call.edit.attributes`, an action which modifies these attributes on the server using the entries the user made in the form. You may provide access to the edit attributes form to users who have write access using the `object.is_editable` placeholder (a Boolean placeholder which is set to true if the current object can be edited by the current user) with a link on the attributes page or from the header or footer which calls `action.call(editattributes.action)`.

If this link is followed, a table similar to the one presented when viewing attributes is presented, but editable attribute entries can be modified by the user with an `<INPUT>` field. The values entered are handled in a special way `(object.attributes.entry.value.escaped)` to allow umlauts and other special characters to be expressed as HTML entities (e.g. "ä" as `&auml;`).

To let users have a form where they can modify the value of existing attributes, a loop such as the following is necessary:

```
%%while object.attributes.next_entry%%

%%if object.attributes.entry.is_editable%%

<TR><TH ALIGN=LEFT COLSPAN=2>%%object.attributes.entry.key%%<TD
ALIGN=LEFT><INPUT
NAME="%%object.attributes.entry.key%%=%%object.attributes.entry.value.escap
ed%%" VALUE="%%object.attributes.entry.value.escaped%%" SIZE=30>

%%endif%%

%%endwhile%%
```

All this does is go through all the attributes one by one and if the attribute is of an editable type, an entry is made for it in a table. The type of attribute is displayed in the first column of the table using the placeholder `object.attributes.entry.key` and on the right side, an input field containing the current value of the attribute (`object.attributes.entry.value.escaped`) appears. In this case, the symbolic name (`NAME`) of the field is not a placeholder of the type `*.form.*` (see below). It is instead the attribute key and the attribute value placeholder separated by an equal sign ("="). This is because when the user sends the modified information back to the server by clicking on the "Change attributes" button, the server needs to know which value goes with which key.

Besides the possibility to edit existing attributes, the attribute editing form should offer the option of assigning further attributes to the object using a listbox as follows:

```
<SELECT NAME=“%%edit.attributes.form.key(input1)%%“>
<OPTION  VALUE="Keyword">Keyword
<OPTION  VALUE="TimeOpen">TimeOpen
<OPTION  VALUE="TimeExpire">TimeExpire
<OPTION  VALUE="DocAuthor">DocAuthor
<OPTION  VALUE="DocDate">DocDate
<OPTION  VALUE="Rights">Rights
</SELECT>
<INPUT NAME=“%%edit.attributes.form.value(input1)%%“ SIZE=30>
```

***.FORM.* PLACEHOLDERS**

In HTML, input fields are given a symbolic name which is used when putting together the query string that gets sent to the remote server when the filled-out form is submitted. PLACE usually uses special predefined constants for the names of inputs fields. These constants are in the form `*.form.*` and there is a special constant for most inputs.

In the example above, the first part of the text displays a menu and lets the user select a value to be submitted to the server. When submitting an attribute name, the constant `edit.attributes.form.key` is used. The last line in the example simply presents the user with an empty text field where he can enter the value of the new attribute. This input field must have the symbolic name `edit.attributes.form.value`. You have probably noticed that both constants have "`(input1)`" at the end. Attributes always come in name-value pairs, and thus the server must have a mechanism for matching up two entries. This is done by putting a word in parenthesis at the end of the constant. This can be any word you like as long as both constants in each corresponding pair of entries have the same word and it is different from the word used for all other pairs.

To display a pair of fields where you can enter an arbitrary new attribute type in the first and a value in the second, the following PLACE code can be used:

```
New Attribute:
<INPUT NAME=“%%edit.attributes.form.key(input2)%%“ SIZE=16>
<INPUT NAME=“%%edit.attributes.form.value(input2)%%“ SIZE=30>
```

See above for an explanation of how placeholders are used when editing attributes.

***Note**: Users and groups are treated as normal objects. They however have special attributes (home collection, password, hosts for auto-login). This has to be considered when showing/editing attributes of normal objects and users/groups with the same template.*

1.1.3.2 INSERTING COLLECTIONS AND OTHER OBJECTS

INSERTING COLLECTIONS

A form for inserting collections is provided by the macro `insertcollection` in the file `insertcollection.html` in the default PLACE templates. Because collections are really only a set of attributes, they can be inserted by calling the predefined action `action.call.insert.attributes` as can be seen in the `action` attribute of the `FORM` field. When creating a new collection you have to specify various attributes such as the collection it should be inserted into and a collection type (selected from a list in the default template). A title and associated language prefix are also required. There are also certain attributes which a collection always has and whose value cannot be changed, e.g. the attribute **Type** is always "Document" (to distinguish it from anchor objects) and **DocumentType** is "collection". Lastly, new attribute types can be selected from a list and their values entered in a field or an arbitrary new attribute type and value can be entered.

As can be seen above there are in total four different ways of entering attributes and other information when inserting a collection. How to realize them in PLACE is listed below:

1. Arbitrary new types and values: this is done for new collections (and other objects) in a way similar to the way it is done when editing attributes (page 6). The difference is that the placeholders `insert.attributes.form.key` and `insert.attributes.form.value` are used instead of `edit.attributes.form.key` and `edit.attributes.form.value`.
2. Type selected from a menu, value entered in a field: this is done for new collections (and other objects) in a way similar to the way it is done when editing attributes (page 6). The difference is that the placeholders `insert.attributes.form.key` and `insert.attributes.form.value` instead of `edit.attributes.form.key` and `edit.attributes.form.value`.
3. Fixed type and fixed value: this is done with input type `HIDDEN` and with `NAME` and `VALUE` set equal to the respective values in quotes, e.g. `<INPUT TYPE=HIDDEN NAME="Type" VALUE="Document">`. In this case the attributes appear in the input form and cannot be changed by the user.
4. Fixed type, variable value: in the case of collections these are e.g. parent collection, language, title, name, rights, etc. These are split into two categories:
   a. Information entered which later appears as an attribute of the object exactly as the user entered it. These are **Title**, **Name**, etc. Here the `NAME` attribute in the input field is the attribute type in quotes, e.g.:

```
<TR><TH ALIGN=LEFT>Title<TD ALIGN=RIGHT> <FONT
  %%mandatory%%>(%%ge:notwendig,:required%%)</FONT><TD><INPUT TYPE=INPUT
  NAME="Title" VALUE="" SIZE=30>
```

   The code in this example causes a row to appear in the form with the words "Title (required)" on the left and an empty input field on the right.

   This form of input allows server administrators to design input forms where an arbitrary fixed attribute type appears on the left of the form and a blank field where users can enter a value appears on the right. A case where this could be useful is when the administrator has configured the server to index an arbitrary attribute (see Custom Attributes in the *Hyperwave Administrator's Guide*).

   b. Information entered which is required by the gateway but which does not appear as actual attributes, for example, parent collection and language. The parent collection is required by the gateway but is not stored as an attribute of a collection, and language is stored not as a separate attribute but as part of the title. This information is entered using `*.form.*` placeholders such as `insert.form.collection` and `insert.form.language`, e.g.:

```
<TR><TH ALIGN=LEFT>%%ge:Einf&uuml;gen in Kollektion,:Insert into
   Collection%% <TD ALIGN=RIGHT><FONT
   %%mandatory%%>(%%ge:notwendig,:required%%)</FONT><TD><INPUT TYPE=INPUT
   NAME="%%insert.form.collection%%" VALUE="%%if object.get_attrib(Name)
   !=""%%%%object.get_attrib(Name)%%%%else%%%%object.get_attrib(GOid)%%%%e
   ndif%%" SIZE=30>
```

Here the text "Insert into Collection (required)" is displayed in the currently selected language and the user is given an entry field with the name or GOid (if the parent collection is a cluster without a name) of the current collection as default entry. As can be seen, `insert.form.collection` is used.

***Note**: The collection should normally inherit all rights from the parent collection. This is done by using the placeholder* `object.inherit.rights` *as default entry in the Rights field.*

INSERTING REMOTE OBJECTS

Like collections (see above), remote objects are actually only a set of attributes stored on the server, the only difference being that some of the information that has to be entered is different. The macro `insertremotedoc` in the file `insertremotedoc.html` displays a form where users enter information about the remote documents they want to insert including title, name, URL, etc. The various types of input fields are coded by the same rules as for collections.

A special placeholder which is only used with remote documents is `insert.attributes.form.url`. This is used to get the URL a user entered in a form. The URL the user enters is divided into Protocol, Host, Path, and Port attributes.

INSERTING CGI OBJECTS

The default templates contain a macro `insertcgi` in file `insertcgi.html` which is used to display the form for inserting CGI objects. Like collections, CGI objects are only a set of attributes on the server and the template used to create the CGI insertion form is similar to that for collections (see Inserting collections). The only attribute which is necessary for CGI objects and for no other objects is **Path**. The actual CGI scripts are stored in the special directory `~hwsystem/dcserver/cgi` and **Path** contains their location relative to this directory.

INSERTING DOCUMENTS

The macro `insertdocument` in file `insertdocument.html` enables users to upload a file along with additional information such as **Title** and **Name**. The placeholder used to upload the form data is `action.call.insert.upload`. Like for the other object types, a parent collection (retrieved by the placeholder `insert.form.collection)` and a title with an associated language (retrieved by `insert.form.language)`are required. With the default templates, fields for name, rights, etc. also appear. For information on how to make further attribute fields available to users see Inserting collections on page 8.

If the file being uploaded is text, a `<BASE>` tag is required to correctly resolve links. The following construct accomplishes this task:

```
<INPUT NAME="%%insert.upload.form.base%%"
VALUE="%%gate.protocol%%://%%gate.hostname_with_port%%">
```

As can be seen, `insert.upload.form.base` is used to retrieve the base, and it is given the name of the server the document is being uploaded to as default.

The name of the file to be uploaded can be typed in (or selected by browsing if the browser supports the `ENCTYPE="multipart/form-data"` feature) with

```
<INPUT Type=FILE Name="%%insert.upload.form%%" VALUE="">
```

***Note:** The multipart/form-data feature is supported in Netscape starting with 3.0 and in Internet Explorer starting with version 4.0.*

INSERTING TEXT

Like for the other objects mentioned above (collections, remote objects, etc.), texts are inserted using a form (see macro `inserttext` in file `inserttext.html`). In the case of a text, however, the object consists of a group of attributes as well as the text itself. Much of the information that needs to be entered when inserting a text is the same as for the other objects (title, name, parent collection, etc.), but of course the text itself must be entered as well. For information on how to use PLACE to create input fields for inserting attributes, see Inserting collections on page 8.

The "Insert text" function in the default templates is such that the text is entered by the user in a text area just before it is uploaded (to upload prepared text files, or any other type of file, the "Insert document" function is used). The text is retrieved using the placeholder `insert.text.form.content.` In the default templates, this looks like this:

```
<TEXTAREA NAME="%%insert.text.form.content%%" rows=25 cols=70 wrap=virtual>
&lt;HTML&gt;
&lt;HEAD&gt;
&lt;TITLE&gt; %%ge: Zu beachten,:Note%%: %%ge: Ein Titel ist zwingend
notwendig,:A title is required%% &lt;/TITLE&gt;
&lt;/HEAD&gt;
&lt;BODY&gt;
&lt;H1&gt;%%ge:&amp;Uuml;berschrift,:Heading%%&lt;/H1&gt;
%%ge: Ihr HTML Text hier...,:Your HTML Body here...%%

&lt;/BODY&gt;
&lt;/HTML&gt;
</TEXTAREA>
```

In this case "rows=25" and "cols=70" are used to specify the size of the text area. As can be seen here, the skeleton of an HTML text is generated in this text area to help the user entering the text.

For texts, the MIME types `text/html` and `text/plain` are supported. This attribute is **required** when submitting a text to tell the gateway which of these to expect. In the default templates, **MimeType** is set to `text/html` automatically with the line

```
<INPUT TYPE=HIDDEN NAME="MimeType" VALUE="text/html">
```

and cannot be changed or seen by the user (`TYPE=HIDDEN`).

The action used to actually upload the text and attributes to the server is `action.call.insert.text`.

1.1.3.3 INSERTING QUERY OBJECTS

The insertion of query objects is integrated into the search function in the Hyperwave user interface. How this function looks to the user differs depending on which browser he or she is using. For users of Netscape or MSIE 3.0, the query object form appears below the results list after a search. For users of Netscape or MSIE 4.0 and above, a button which can be used to access the Hyperwave Query Object Wizard appears in the search results list. Both of these alternatives are handled in the template file `querycreator.html`, which contains the two macros `querycreatorform` and `querycreator`.

The macro `querycreatorform` produces the actual form, while `querycreator` calls `querycreatorform` and uses it to display the Query Object Wizard.

1.1.3.4 INSERTING ANNOTATIONS

Inserting annotations is, as far as the server is concerned, the same as inserting text. The same action (`action.call.insert.text`) is used to upload both. The only differences are in what users see when they access these functions.

In the default templates, the insert annotation form is designed to be simpler than the insert text form, the idea being that people who want to comment on information on a server may not necessarily know how to use HTML. The text is in fact inserted as HTML, but the header and footer are hidden from the user and are inserted automatically. Because annotations allow HTML markup, it is possible for users to format their annotations as they wish, however, the use of HTML is not necessary. It is also simpler to set access permissions when inserting annotations.

As with text insertion, the placeholder used to retrieve the text entered in the text area is `insert.text.form.content`. However, some mechanism for allowing different parts of the text to be attached in the right order before they are uploaded to the server is necessary (see last paragraph). This is done by giving the placeholder a parameter in parenthesis at the end which the server uses to sort the pieces by. This makes it possible to mix automatically generated text with user entries in the order desired.

As can be seen in the macro `insertannotation` (in file `insertannotation.html`), when the title is entered, the entry is not given the symbolic name `"Title"` as in `inserttext.html`, but rather the placeholder `insert.text.form.content(b)` is used. This causes the title the user enters to be inserted in the proper place among the other parts of the text. Below is the part of the macro where the rest of the text is generated.

```
<INPUT TYPE=HIDDEN NAME="%%insert.text.form.content(a)%%"
VALUE="&lt;HTML&gt;&lt;HEAD&gt;&lt;TITLE&gt;">

<INPUT TYPE=HIDDEN NAME="%%insert.text.form.content(c)%%"
VALUE="&lt;/TITLE&gt;&lt;/HEAD&gt;&lt;BODY&gt;&lt;H2&gt;%%ge:Ein Kommentar
zu,:A comment on%% &lt;A HREF=&quot;%%object.relpath%%&quot;
REL=&quot;annotation&quot;&gt;%%object.title%%&lt;/A&gt;&lt;/H2&gt;">

<TR BORDER=0><TH COLSPAN=3 ALIGN=CENTER><TEXTAREA
NAME="%%insert.text.form.content(d)%%" rows=13 cols=70 wrap=virtual>

</TEXTAREA>

<INPUT TYPE=HIDDEN NAME="%%insert.text.form.content(e)%%"
VALUE="&lt;/BODY&gt;&lt;/HTML&gt;">
```

As can be seen, the "d" part is where the user enters the text in a text area. The title, which is "A comment on `%%object.title%%`" by default and the rest of the elements are alphabetized according to parameter.

By default, the annotation is inserted into the current collection if it is a collection and if the user has write permission for it, otherwise into the user's home collection:

```
<TR><TH ALIGN=LEFT>%%ge:Einf&uuml;gen in die Collection,:Insert into
Collection%% <TD ALIGN=RIGHT> <FONT
%%mandatory%%>(%%ge:notwendig,:required%%)</FONT><TD><INPUT
NAME="%%insert.form.collection%%"
VALUE="%%if object.get_attrib(DocumentType) == "collection"%%
%%if object.is_editable%%
%%object.get_attrib(Name)%%
%%else%%%%session.user.get_attrib(Home)%%%%endif%%
%%else%%%%if lastcoll.is_editable%%
%%lastcoll.get_attrib(Name)%%%%else%%%%session.user.get_attrib(Home)%%%%endif%%
%%endif%%" SIZE=30>
```

If the current object is not a collection, then the annotation is inserted into the last accessed collection if it is editable by the user and if not into the user's home collection.

The MIME type of the annotation is automatically set to `text/html` in the same way as it is done when inserting texts (see Inserting text).

As mentioned above, setting access rights is simple when inserting annotations. Users select from a menu with the choices public (everyone can read it), private (only the author can read it) and inherited (the annotation inherits the access rights of the collection it is contained in). This is realized as follows:

```
</SELECT>

<TR><TH ALIGN=LEFT>%%ge:Rechte,:Rights%%<TD ALIGN=RIGHT><FONT
%%mandatory%%>(%%ge:notwendig,:required%%)</FONT><TD><SELECT NAME="Rights" >
<OPTION  SELECTED VALUE="">%%ge:&ouml;ffentliche Annotation,:public
annotation%%
<OPTION  VALUE="R: a">%%ge:private Annotation,:private annotation%%
<OPTION  VALUE="%%object.inherit.rights%%">%%ge:vererbt,:inherited%%

</SELECT>
```

As can be seen, the "public annotation" option sets the value of the rights attribute to "", thus giving the annotation the default access rights, i.e. everyone can read the annotation. "Private annotation" restricts read access to the author (`R:a`) and "inherited" sets the rights attribute to the value of the placeholder `object.inherit.rights` that is, the value of the **Rights** attribute of the current object.

AFTER UPLOADING IS FINISHED

After a new object has been inserted into the server a page appears by default (generated by macro `insertresult` in file `insertresult.html`) which gives the user the choice of returning to the parent collection or visiting the newly inserted object. An icon which can be clicked on to view the attributes of the respective object appears next to the title of the object. This is possible by using the following placeholders:

- `insert.newobject.title, insert.newobject.uri`

  These two placeholders return the title and the URI of the most recently inserted object, respectively.

- `insert.newobject.get_attrib(`*AttributeName*`)`

  This placeholder allows you to get the value of any attribute of the object which was just inserted. In the default template this is used display the **PresentationHints** attribute next to the title of the new object.

- `insert.newobject.call(`*ActionName*`)`

  This placeholder is used to call a self-defined action. In the section Actions on page 4, it says that self-defined actions are called using the placeholder `action.call(`*ActionName*`)`, however, there are also other similar placeholders which are used to apply an action to a specific object. In the default templates, `insert.newobject.call` is used to call the action `attributes.action`, which is an action used to display the attributes of a given object. Because the special `newobject` placeholder is used, the action is applied not to the current object but to the object which was just inserted.

1.1.3.5 DELETING, MOVING AND COPYING OBJECTS

MOVING/COPYING OBJECTS

The macro `mvcpobject` in file `movecopyobjects.html` produces a form which allows users to use checkboxes to select the items they want to move or copy from a given collection. The form submits the information to the server by calling the action `action.call.movecopy.object`. Besides the GOids of the items, various other information has to be submitted to the server. There are several placeholders meant to retrieve this information.

- `movecopy.object.form.parent`

  This submits the GOid of the collection the objects are to be moved/copied from. It is set equal to "`object.get_attrib(GOid)`", a placeholder which returns the value of the global object id for the current object. This input is done automatically with

```
<INPUT TYPE=hidden NAME="%%movecopy.object.form.parent%%"
VALUE="%%object.get_attrib(GOid)%%">
```

- `movecopy.object.form`

  This is used to submit the GOids of the objects to be moved or copied. The objects are shown as a list of items with checkboxes next to them. The checkboxes are created with an input field like the following:

```
<INPUT TYPE=CHECKBOX NAME="%%movecopy.object.form%%"
VALUE="%%coll.entry.get_attrib(GOid)%%">
```

  As can be seen, the value submitted is the GOid of the entry.

- `movecopy.object.form.destination`

  This placeholder retrieves the destination collection or cluster that the user enters in a field, i.e.:

```
<INPUT NAME="%%movecopy.object.form.destination%%" VALUE="" SIZE=40>
```

- `movecopy.object.form.mode`

  This is used to submit the preference to move or copy the checked objects. In the default templates this is done with radio buttons:

```
<INPUT TYPE="radio" NAME="%%movecopy.object.form.mode%%"
VALUE="%%movecopy.object.form.mode.copy%%">%%ge:Kopieren,:Copy%%

<INPUT TYPE="radio" CHECKED NAME="%%movecopy.object.form.mode%%"
VALUE="%%movecopy.object.form.mode.move%%">%%ge:Verschieben,:Move%%
```

- `movecopy.object.form.mode.copy` and `movecopy.object.form.mode.move`

As can be seen above, these are the possible values for `movecopy.object.form.mode`.

DELETING OBJECTS

The macro `deleteobjects` in file `deleteobjects.html` produces a form where users can select the items they want to delete from a collection by using checkboxes. The way this works is similar to the way the move/copy objects function works except that (obviously) it is not necessary to submit the destination collection or to submit a preference to move or copy. There are two placeholders which are important for the delete templates:

· `delete.object.form.parent`

This submits the GOid of the collection or cluster the objects are to be deleted from. The collection the user was browsing at the time the delete function was called is submitted automatically with a "hidden" form field in the default template by setting its value equal to the GOid of the current object `(object.get_attrib(GOid))`:

```
<INPUT TYPE=hidden NAME="%%delete.object.form.parent%%"
VALUE="%%object.get_attrib(GOid)%%">
```

· `delete.object.form`

This is used to submit the GOids of the objects to be deleted. The objects are shown as a list of items with checkboxes next to them. The checkboxes are created with an input field like the following:

```
<INPUT TYPE=CHECKBOX NAME="%%delete.object.form%%"
VALUE="%%coll.entry.get_attrib(GOid)%%">
```

The predefined action used to submit the information entered in the form to the server is `action.call.delete.object`.

1.1.3.6 LOCKING OBJECTS

You may provide a button with a link to the action `action.call.lock.object` to enable users to lock objects. Locked objects are still visible to all users with read access to them, but although someone else may have write access no one will be able to edit this object unless it is unlocked by the same user who locked it (using `action.call.unlock.object)`. Information on who is locking the object can be displayed with `object.lock.get_attrib(`*AttributeName*), e.g. to find out who is locking the object use `object.lock.get_attrib(`*Author*`)`, or to find out when the object was locked use `object.lock.get_attrib(`*TimeCreated*`)`, preferably for users with write access.

1.1.3.7 EDIT AND REPLACE FUNCTIONS

EDIT FUNCTION

The macro `edittext` in file `edittext.html` allows users to edit HTML and plain text documents which already reside on the server. For this function there are two important placeholders. They are described below.

· `edit.text.form.content`

This placeholder is used to retrieve the edited text by making it the value of the TEXTAREA NAME attribute:

```
<TEXTAREA NAME="%%edit.text.form.content%%" rows=25 cols=70
wrap=virtual>%%object.edit.content%%</TEXTAREA>
```

· `object.edit.content`

This placeholder returns the content of the current object. It is used, as can be seen above, to display this content in the text area so the user can edit it.

The edited text is submitted to the server with the action `action.call.edit.text`.

Besides the actual editing of the text, one more thing that has to be considered is the avoidance of write conflicts. It is important that the data base does not become inconsistent when two users edit the same text simultaneously. This is done as follows:

1. At the instant the user activates the edit function, the value of the **TimeModified** attribute (the time the text was last edited) of the document to be edited is retrieved .

2. This value is automatically submitted back to the server in a HIDDEN input field with the edited text. The server compares this value with the current value of this attribute for the document. If the values are the same, no one else has edited the text since the user started editing it, and the text is submitted to the server. If the values are different, someone else has edited the text. In this case the user receives an error message and the edited text is not uploaded. The code used to send **TimeModified** back is as follows:

```
%%while object.attributes.next_entry%%
%%if object.attributes.entry.key == "TimeModified"%%
<INPUT TYPE=HIDDEN NAME="%%object.attributes.entry%%"
VALUE="%%object.attributes.entry.value%%">
%%endif%%
%%endwhile%%
```

REPLACE FUNCTION The default templates offer the user the option of replacing existing documents on the server (macro `replacedocument` in `replacedocument.html`). The process is similar to normal file upload, therefore the same action `(action.call.edit.upload)` is called to submit the new file to the server. The file to upload and replace the existing document (by keeping the same attributes) is handed over using the placeholder `edit.upload.form` as `NAME` attribute:

```
<INPUT TYPE=FILE NAME="%%edit.upload.form%%" VALUE="">
```

The base is submitted with the placeholder `insert.upload.form.base`.

To use the Multipart feature of some browsers the `<FORM>` tag must have the attribute `ENCTYPE="multipart/form-data"`.

1.1.3.8 DISPLAYING ANNOTATIONS

When an object has been annotated, it is usually desirable that users browsing the object are made aware that the annotations exist, and thus the default templates display links to any existing annotations in the footers of documents. The following lines are found in the macro `footer` in `footer.html`:

```
%%if annotations.count > 0%%
    %%if annotations.count == 1 %%
      %%if annotations.next_entry #dummy expression to get first element%%
, <A HREF="%%annotations.entry.uri%%"><B>%%ge:Bemerkung, :Annotation%%</B></A>
      %%endif%%
    %%else%%
, <A HREF="%%action.call(annotations.action)%%"><B>%%ge:Bemerkungen,
:Annotations%% (%%annotations.count%%)</B></A>
    %%endif%%
  %%endif%%
```

This code displays links to the annotations of the current document, if there are any. If there is exactly one annotation, a link directly to that annotation is generated, if there is more than one, a link to the self-defined action `annotations.action` is created. This calls a macro `annotations` in file `annotations.html`, which makes a list of links to the annotations with a link to the attributes of the annotation:

```
%%macro annotations%%

<HR>
<H1>%%ge:Bemerkungen zu, :Annotations to%% "%%object.title%%"</H1>
<HR><DL>
%%while annotations.next_entry%%
<DT><DD><A
HREF="%%annotations.entry.call(attributes.action)%%">%%show_infoicon%%</A>&nbsp
;<IMG src="%%annotations.entry.icon%%" ALIGN="middle"> <A
HREF="%%annotations.entry.uri%%">%%annotations.entry.title%%</A>
%%endwhile%%
</DL><P>

%%endmacro%%
```

1.1.3.9 IMPLEMENTING THE SEARCH FUNCTION

The search function in Hyperwave is very powerful (with fulltext and attribute search on arbitrary indexed and non-indexed attributes), resulting in a fairly complex handling of the search template.

The search form is provided with macro `exsearchform` in file `exsearchform.html` (see note below for an explanation of "extended") and uses the action

`action.call.extended.search` to start the search on the server. How the search results are displayed is defined in `exsearchresults.html`.

SEARCH FORM

The default search form (called by the self-defined action `extended.search.action)` contains a field where the user can type in a search term, as well as checkboxes and radio buttons which are used to specify other search preferences. The search term entered is retrieved by `extended.search.form.var(`*variable1*`)`, which is the placeholder used to retrieve all terms entered in fields in the search form. *variable* is any name you want, as long as it is different for each input field. In the default template, the input field for the search term is produced as follows:

```
<INPUT NAME="%%extended.search.form.var(s1)%%">
```

thus storing the search query in variable `s1`. The next step is to map the term to different indexed attributes. In Hyperwave, the attributes **Title**, **Keyword** and **Name** are always indexed (and it is also possible to configure the server to index further custom attributes, see the *Hyperwave Administrator's Guide*). Users can select which of these attributes they want to search for the term in using checkboxes in the default templates. When an attribute is selected to be searched in, the search term is mapped to that attribute using the placeholder `extended.search.form.map(`*variable2*`)` by setting the value to `"`*attribute*`=`*variable1*`"`. The example below shows how this can be done:

```
<INPUT TYPE="checkbox" NAME="%%extended.search.form.map(title)%%"
VALUE="Title=s1">

<INPUT TYPE="checkbox" NAME="%%extended.search.form.map(keyword)%%"
VALUE="Keyword=s1">

<INPUT TYPE="checkbox" NAME="%%extended.search.form.map(name)%%"
VALUE="Name=s1">
```

You may add any other indexed attribute here (the only other attributes which are automatically indexed are **Author**, **TimeCreated** and **TimeModified**). If the user wants to run a fulltext query, the following expression provides a checkbox for this:

```
<INPUT TYPE="checkbox" NAME="%%extended.search.form.map(fulltext)%%"
VALUE="s1">
```

Other placeholders which are important for submitting search preferences are:

· `extended.search.form.languages`

  This is used to submit the language or languages you want to restrict the search to. Only items whose title is declared as the specified language will be considered in the search.

· `extended.search.form.scope`

  This placeholder allows you to submit the part of the server you want to search in. Options are currently the entire server (by setting `VALUE` equal to `"%%extended.search.form.scope.server%%"`) or the last collection visited (`"%%lastcoll.get_attrib(GOid)%%"`). Any other collection can be added to customize the search.

· `extended.search.form.maxobjects`

  This placeholder, submitted as "hidden" in the default template, controls the number of items shown in the search results list.

· `extended.search.form.sortorder`

  This placeholder lets you submit the order you want the result list to be sorted in.

See the file `exsearchform.html` for examples of the use of these placeholders.

EXTENDED SEARCH OPTIONS

The extended search form allows users to specify further restrictions on the search by letting them specify that the objects found should be filtered according to further (not necessarily indexed) attributes such as author, creation time, and modification time.

Like for the search query, the user enters this information in different input fields, and the template uses the placeholder `extended.search.form.var(`*variable*`)` to store the input in arbitrary variables:

```
<TD COLSPAN=2>%%ge:Autor,:Author%%
<TD COLSPAN=2><INPUT NAME="%%extended.search.form.var(s2)%%" SIZE="20"
MAXLENGTH="512">

<TR><TD COLSPAN=2>%%ge:Erstellt vor,:Created before%%
<TD COLSPAN=2><INPUT NAME="%%extended.search.form.var(s3)%%" SIZE="20"
MAXLENGTH="20"> %%ge:(jjjj/mm/tt) oder relatives Datum,:(yyyy/mm/dd or
relative date)%%

<TR><TD COLSPAN=2>%%ge:Ver&auml;ndert nach,:Modified after%%
<TD COLSPAN=2><INPUT NAME="%%extended.search.form.var(s4)%%" SIZE="20"
MAXLENGTH="20"> %%ge:(jjjj/mm/tt) oder relatives Datum,:(yyyy/mm/dd or
relative date)%%
```

As can be seen, the author name is stored in `s2`, the creation date in `s3` and the modification date in `s4`.

Once the user has entered all preferences for the search, this information has to be put together in queries to be submitted to the server. This is done in the default templates using "hidden" input fields, e.g.:

```
<INPUT TYPE="hidden" NAME="%%extended.search.form.map(restrict)%%"
VALUE="Author=s2 && TimeCreated<s3 && TimeModified>s4">
```

Here we can see that the placeholder `extended.search.form.map(`*variable*`)` is used once again, this time to map the inputs in the extended search options fields to the respective attribute types, put them together in a query (in this case using the `AND` operator (`&&`),) and set the variable `restrict` equal to this query.

```
<INPUT TYPE="hidden" NAME="%%extended.search.form.objectquery%%"
VALUE="restrict">
```

The value of `restrict` is submitted to the server as an object query (query on non-indexed attributes) with placeholder `extended.search.form.objectquery`. Here any other (indexed or non-indexed) attribute can be added to the search restrictions using any relation.

```
<INPUT TYPE="hidden" NAME="%%extended.search.form.fulltextquery%%"
VALUE="fulltext">
```

The fulltext query is submitted with `extended.search.form.fulltextquery` with value `fulltext` (which was set equal to the search term above).

```
<INPUT TYPE="hidden" NAME="%%extended.search.form.keyquery%%"
VALUE="(keyword || title || name ) && restrict" >
```

The keyquery (`extended.search.form.keyquery`) is the query on indexed attributes and its value is a combination of the variables whose values were set to a mapping of the search term (`s1`) to attributes. Here the OR operator (`||`) is used to combine `title`, `keyword` and `name` and `restrict` is added using AND (`&&`). These operators can be changed as desired to AND, OR or ANDNOT (`&!`).

The search is started with the action `action.call.extended.search`.

WHY VARIABLE "RESTRICT" IS SUBMITTED WITH BOTH THE KEYQUERY AND OBJECT QUERY SEARCH

You may be wondering why `restrict` is submitted both with the keyquery and the object query. This has to do with the fixed internal structure of the Hyperwave search mechanism. The general scheme is that first two separate searches are made, a keyquery search for indexed attributes and a fulltext search in the content of objects. The results of these searches are added together and then filtered for non-indexed attributes with the object query. However, in our particular case, the attributes searched for with the extended search options (author, creation time and modification time) are indexed, and thus can be quickly searched for during the keyquery search. This makes the object query quicker and more efficient by reducing the number of objects that have to be searched through. The objects from the fulltext search however still need to be filtered, and thus `restrict` must also be submitted as the object query.

It should also be noted that because of the search mechanism, search in fulltext cannot be "ANDed" together with keyquery search. For example, you can’t specify that a term should be found in *both* the title (in the keyquery search) and the content of an object. This must be done by running the fulltext search first and then filtering the results with the object query.

***Note**: Hyperwave versions 2.0 and later use an "extended" search form compared to all previous releases (the set of placeholders used in the simple search form are still included in the distribution). Therefore all search placeholders added after version 2.0 consist of names with the prefix "extended".*

SEARCH RESULTS The first few lines of the search results template (`exsearchresults.html`), write the values of the search parameters the user entered if their respective values (in our case `s1` through `s4`) are not empty, e.g.:

```
%%if extended.search.vars.entry.name =="s1" &&
extended.search.vars.entry.value != ""%%%%ge:Suchbegriff,:Search
expression%%: %%extended.search.vars.entry.value%%<BR>%%endif%%

%%if extended.search.vars.entry.name =="s2" &&
extended.search.vars.entry.value != ""%%%%ge:Autor,:Author%%:
%%extended.search.vars.entry.value%%<BR>%%endif%%
```

etc.

The placeholder `extended.search.count` gives the total number of matches and if it is not zero, the results are displayed using a while loop like the following:

```
%%while extended.search.next_entry%%

    %%if extended.search.entry.title!="" ||
extended.search.entry.get_attrib(Keyword)!=""%%

<DT><DD><IMG src="%%if extended.search.entry.get_attrib(FullTextQuery) !=
"" && extended.search.entry.get_attrib(KeyQuery) !=
""%%%%gate.object(ball_blueyellow.gif)%%%%else%%%%if
extended.search.entry.get_attrib(FullTextQuery) !=
""%%%%gate.object(ball_blue.gif)%%%%else%%%%gate.object(ball_yellow.gif)%%%
%endif%%%%endif%%"> <A
HREF="%%extended.search.entry.call(attributes.action)%%">
%%show_infoicon%%</A> <IMG src="%%extended.search.entry.icon%%">
(%%extended.search.entry.get_attrib(Score)%%%) <A HREF="%%if
extended.search.entry.get_attrib(DocumentType)=="Image"%%%%extended.search.
entry.call(image.action)%%%%else%%%%extended.search.entry.uri%%%%endif%%">%
%if
extended.search.entry.title!=""%%%%extended.search.entry.title%%%%else%%%%e
xtended.search.entry.get_attrib(Keyword)%%%%endif%%</A>

      %%if extended.search.entry.parents.count == 1 %%
%%ge:in Kollektion, :in collection%%
        %%while extended.search.entry.parents.next_entry%%
 <A
HREF="%%extended.search.entry.parents.entry.uri%%">[%%extended.search.entry
.parents.entry.title%%]</A>
        %%endwhile%%
      %%endif%%
     %%endif%%
    %%endwhile%%</DL>
```

The results themselves are listed with their titles (or with their keywords if they don't have any titles, which usually should not happen), each with a link to the object. The list entry gets a different colored marker depending on if it was found in a fulltext search (`extended.search.entry.get_attrib(FullTextQuery)`), a keyword search (`extended.search.entry.get_attrib(KeyQuery`)), or both. A score in percent which ranks the results (`extended.search.entry.get_attrib(Score)`) is also displayed.

Note that the placeholder `extended.search.entry.get_attrib(`*attribute*`)` can be used to retrieve any regular attribute of an object in the result list, as well as to retrieve the special attributes `Score`, `FullTextQuery` and `KeyQuery`, which are set by WaveMaster.

1.1.3.10 IMPLEMENTING USER MANAGEMENT

When accessing WaveMaster's user management facilities, the first thing you see is a form which can be used to search for users and/or groups. This form is created in the default template by the macro `userform` in `userform.html`. The expression to be searched for is retrieved by the placeholder `search.form.searchexp`:

```
<INPUT NAME="%%search.form.searchexp%%" SIZE="40" MAXLENGTH="512">
```

The only other preference that has to be entered is whether the search should be done in groups or users or both. This is done with checkboxes like the following:

```
<INPUT TYPE="checkbox" NAME="%%search.form.searchin.uname%%"
VALUE="%%search.form.yes%%" CHECKED> %%ge:Benutzern,:Users%%

<INPUT TYPE="checkbox" NAME="%%search.form.searchin.ugroup%%"
VALUE="%%search.form.yes%%"> %%ge:Gruppen,:Groups%%
```

The search is started on the server by calling the action `action.call.search`.

At the bottom of the search form there is button with "Show All Users/Groups" on it. This is implemented like the search above but automatically enters fixed information, telling the server to search in user names and groups and to enter "*" as the search term, and calling the action `action.call.search`:

```
<FORM METHOD="GET" ACTION="%%action.call.search%%">

<INPUT TYPE="hidden" NAME="%%search.form.searchin.uname%%"
VALUE="%%search.form.yes%%">

<INPUT TYPE="hidden" NAME="%%search.form.searchin.ugroup%%"
VALUE="%%search.form.yes%%">

<INPUT TYPE="hidden" NAME="%%search.form.searchexp%%" VALUE="*">

<INPUT TYPE="submit" VALUE="%%ge:Alle User/Gruppen zeigen,:Show All
Users/Groups%%">

</FORM>
```

1.1.3.11 CREATING USERS AND GROUPS

The last thing that the macro `userform` does is display buttons which can be used to create a new user or group if the current user is a system user. The following code creates the "New User" button which calls the action `newuser.action` when clicked on:

```
%%if session.user.is_system%%
<TD ALIGN=left>
<FORM method=GET action="%%action.call(newuser.action)%%">
<INPUT TYPE=submit VALUE="%%ge:Neuer Benutzer,:New User%%">
</FORM>
```

`newuser.action` (which calls the macro in `newuser.html`) is a self-defined action which displays a form where you can enter the various information necessary for a user record. Like with collections, a user record consists only of attributes and thus the action `action.call.insert.attributes` is used to insert it on the server. Two special placeholders used in the new user form are `insert.attributes.form.newpassword` (which is used to retrieve the password) and `insert.attributes.form.newpasswordverify` (which is used to retrieve the verification of the new password):

```
<TR><TH ALIGN=LEFT>Password<TD ALIGN=RIGHT> <FONT
%%recommended%%>(%%ge:empfohlen,:recommended%%)</FONT><TD><INPUT
TYPE="password" NAME="%%insert.attributes.form.newpassword%%" VALUE=""
SIZE=12><TD><INPUT TYPE="password"
NAME="%%insert.attributes.form.newpasswordverify%%" VALUE="" SIZE=12>
(%%ge:&Uuml;berpr&uuml;fung, :Verification%%)
```

Because inserting new users and groups is similar to inserting collections (both are nothing more that a set of attributes stored on the server), refer to Inserting collections and other objects on page 8 for more information on how you can edit the `newuser` macro.

The mechanism for letting users add new groups is analogous except that different information has to be entered in the form.

1.1.3.12 EDITING USERS/GROUPS

After you have made a search, the search results are displayed as a list of links by WaveMaster using the macro `searchresults`. If one of the links is clicked on, the attributes of that particular user or group record are displayed by macro `attributes` (see Displaying attributes, page 6). An "Edit" button which calls the action `editattributes.action` is also displayed if the current user has permission to edit the attributes. Editing user and group records is handled the same way editing of any other attributes is handled (see Editing attributes on page 6).

1.1.3.13 IDENTIFY AS USER

A user can identify by calling the predefined action `action.call.identify` (typically a link from one of the Hyperwave buttons at the top of the screen). This causes a dialog box to be displayed where users can enter their name and password. If identification is successful the macro `identification` (in file `identification.html`) is called to display a page saying this, including links to various objects such as the user's home collections. This page uses the following placeholders in the default template:

· `session.username`

Shows the name of the currently logged in user.

· `action.call.home`

This action accesses the current user's home collection.

· `object.uri`

This is the URI of the current object.

1.1.3.14 CHANGING YOUR PASSWORD

Providing the user with a form to change his or her password is quite straightforward (see macro `passwordform` in file `passwordform.html`). Each user entry is assigned to the appropriate placeholder `(password.form.oldpassword`, `password.form.newpassword`, `password.form.newpasswordverify).` After this information is submitted the macro `passwordresults` in file `passwordresult.html` either displays a message that the user has identified successfully or displays the appropriate error message.

1.1.3.15 SHOWING INFORMATION ABOUT THE GATEWAY AND THE SERVER

Information about the gateway and server is by default accessible on the options page, which is formatted using the macro in the file `options.html`.

The placeholder `gate.version` is used to display the version number of the WaveMaster you are using, which is a useful piece of information you or your users may have to provide to our support team in case of problems. The version information that appears has a link calling the self-defined action `about.action`, which calls a macro named `about`, which displays a page containing further information including who to contact if you have bug reports, comments or suggestions.

The action call of the self-defined action `serverstatus.action` results by default in a table of detailed information about start-up time and several other characteristics of the server modules (including version numbers). This information will help you to get an overview of what is happening on your server.

By default this information can be viewed by anyone who accesses the server. You may choose to show this information to the server administrator only (users belonging to group "system") using an if-statement with the Boolean placeholder `session.user.is_system`:

```
%%if session.user.is_system%%
<FORM method=GET action="%%action.call(serverstatus.action)%%">
<STRONG>Get information about %%hyper.servername%%:</STRONG>
<INPUT TYPE=submit VALUE="Server Status">
%%endif%%
```

A list which contains data about the users currently logged in to the server is available by using a button to call the action `users.action` in the default distribution.

1.1.3.16 NAVIGATIONAL FEATURES

SHOWING PARENTS

The server administrator may provide several features to help the user navigate on the server. One possibility is to offer information in the form of sequences (see page 21), another is to always show a link to one or more parent collections whenever an object is displayed so that users can move up one step in the collection hierarchy. In the default template the automatic display of these links is implemented in the macro footer as follows:

```
%%if parents.count <= 3 %%
%%if parents.count == 0 %%
%%ge:keine sichtbaren Eltern, :no visible parents%%
    %%else%%
      %%while parents.next_entry%%
        %%if parents.entry.get_attrib(RemoteID)==""%%
[<A HREF="%%parents.entry.uri%%"><B>%%parents.entry.title%%</B></A>]
        %%endif%%
      %%endwhile%%
    %%endif%%
  %%else%%
<A HREF="%%action.call(parents.action)%%"><B>%%ge:Eltern,
:Parents%%</B></A>
%%endif%%
```

This construct generates a list of links to the parent collections using a while-loop if the total number of parents (`parents.count`) is 3 or less. If the Boolean placeholder `parents.next_entry` is `true` (meaning there is at least one more entry in the list of parents), the title of the collection (`parents.entry.title`) with a link to that collection (`parents.entry.uri`) is displayed. If there are four or more parents, a link with the text "Parents" is offered to the user. This link calls the self-defined action `parents.action`, which generates a page with a list of links to the parents (see macro `parents` in `parents.html)`.

1.1.3.17 HANDLING ERRORS

The file `error.html` shows a list of error messages which can be customized for your users. This is called by the special action `error` (at the beginning of `master.html`)

1.1.3.18 USING SESSION VARIABLES

If you want the user to be able to change the WaveMaster interface during the session (e.g. switch to a preferred language, switch to frame mode, etc.), you have to use *session variables*. As with actions, there are two types of session variables: predefined and self-defined. Currently the only predefined session variable that can be directly set with the use of a predefined action is **session.language**, which has as value the user’s currently set language preference in the form of a Hyperwave language prefix (see page 4).

There is a special action for setting the session language called **action.call.setlanguage**. When this action is called, **action** is set to `action.setlanguage`. In the default template, a menu is used to let users select a language. The action **action.setsession.form.language** retrieves the preferred language value from the form the user enters it in. After the user clicks on the submit button, `session.language` is set to the language selected in the form. This is done with the following PLACE code in the default template:

```
<FORM method=POST action="%%action.call.setlanguage%%">
<STRONG>Set Language to </STRONG>
<SELECT NAME=%%action.setsession.form.language%%><OPTION VALUE=en SELECTED>
English <OPTION VALUE=ge> German </SELECT>
<INPUT TYPE=submit VALUE="Submit Language Setting">
</FORM>
```

The server uses this language setting to retrieve documents of the proper language from clusters. The language setting is also used when you want to display a text in the WaveMaster interface according to language preference (see page 4).

DEFINING NEW SESSION VARIABLES

You may also define any session variable you like. The value of the variable is set using the action **action.call.setvar**. Self-defined variables can be used for example to switch to edit mode or make the WaveMaster interface offer the possibility to switch to a frame mode. A form in which the user can choose frame or non-frame mode and that calls the action **action.call.setvar** must be coded. In PLACE this would look like:

```
<FORM method=GET action="%%action.call.setvar%%">
Frame Mode:
<INPUT TYPE="radio" NAME="FRAME_MODE" VALUE="YES" CHECKED> Frames
<INPUT TYPE="radio" NAME="FRAME_MODE" VALUE="NO">  No Frames
<INPUT TYPE=submit VALUE="Change Frame Mode">
</FORM>
```

The self-defined variable above is called `FRAME_MODE` and is declared using `NAME="FRAME_MODE"`. Note that self-defined variables always have a value of type STRING.

Once the variable is set, you can use the placeholder `session.getvar(`*variable*`)` to get the value of the variable. Distinguishing between frame mode and non-frame mode would look like:

```
%%if session.getvar(FRAME_MODE) == "YES"%%
  [HTML text for showing object in frame mode]
%%else%%
  [HTML text for showing object without frames]
%%endif%%
```

1.1.3.19 BUILDING SEQUENCES

Hyperwave provides various collection types including *sequences*. These are collections whose members you can page through using "forward" and "back" buttons. Sequences are created by setting the attribute **CollectionType** of a collection to **Sequence** (this can be done while inserting the collection with WaveMaster or afterwards by editing its attributes). When a sequence is retrieved from the server, the WaveMaster interface should provide five additional text or icon links that are used to do the following when activated:

1. go to the first document in the sequence
2. go to the last document in the sequence
3. go to the previous document in the sequence
4. go to the next document in the sequence
5. leave sequence (show list of the sequence’s contents)

The file `sequence.html` in the default template contains a macro `sequence` which provides this additional button set with links and displays the sequence title, the current sequence position, and the number of documents in the sequence. The macro looks as follows:

```
%%macro sequence%%
%%if object.is_in_sequence%%
<HR>
  %%if sequence.prev%%
    <A HREF="%%sequence.first.uri%%"> <IMG ALT="FIRST" WIDTH="35" HEIGHT="35"
SRC="%%gate.object(seq_first.gif)%%" BORDER=2></A>
<A HREF="%%sequence.prev.uri%%"> <IMG ALT="PREVIOUS" WIDTH="35" HEIGHT="35"
SRC="%%gate.object(seq_prev.gif)%%" BORDER=2></A>
  %%else%%
<IMG ALT="FIRST" WIDTH="35" HEIGHT="35" SRC="%%gate.object(seq_first.gif)%%"
BORDER=2>
<IMG ALT="PREVIOUS" WIDTH="35" HEIGHT="35" SRC="%%gate.object(seq_prev.gif)%%"
BORDER=2>
    %%endif%%
    %%if sequence.next%%
<A HREF="%%sequence.next.uri%%"> <IMG ALT="NEXT" WIDTH="35" HEIGHT="35"
SRC="%%gate.object(seq_next.gif)%%" BORDER=2></A>
<A HREF="%%sequence.last.uri%%"> <IMG ALT="LAST" WIDTH="35" HEIGHT="35"
SRC="%%gate.object(seq_last.gif)%%" BORDER=2></A>
    %%else%%
<IMG ALT="NEXT" WIDTH="35" HEIGHT="35" SRC="%%gate.object(seq_next.gif)%%"
BORDER=2>
<IMG ALT="LAST" WIDTH="35" HEIGHT="35" SRC="%%gate.object(seq_last.gif)%%"
BORDER=2>
    %%endif%%
<A HREF="%%sequence.uri%%"> <IMG ALT="LAST" WIDTH="35" HEIGHT="35"
SRC="%%gate.object(seq_toc.gif)%%" BORDER=2></A>
<B>%%ge:Sequenz, :Sequence%%: %%sequence.title%%
(%%sequence.number%%/%%sequence.count%%)</B>
  %%endif%%
%%endmacro%%
```

This macro uses several placeholders to display information related to the sequence and to access certain members of the sequence. For instance, the value of the Boolean placeholder `object.is_in_sequence` is used to decide whether the object should be treated as a member of a sequence. If the Boolean placeholder `sequence.prev` is true, links to the previous and the first objects in the sequence are made, otherwise the icons are displayed but not linked. `sequence.number` and `sequence.count` display the total number of objects in the sequence and the number of the current object, respectively. The icons in this case are in the same directory as the template files. To display such an icon, the **gate.object(***objectname***)** placeholder must be used; where *objectname* is the file name of the object (e.g.

`gate.object(seq_prev.gif)`). The icons can also be in a different directory, in which case the variable `HTML_DIR` can be used if it has been properly set in the server's configuration file `.db.contr.rc`.

Note: *If a sequence has a collection head, the collection head will be shown just like the collection head of an ordinary collection and will not be included in the sequence.*

Note: *If a sequence has a full collection head, you will immediately jump into the sequence, starting at the position of the full collection head*

1.1.3.20 ACTIVATING YOUR PLACE TEMPLATES

If you edit your PLACE templates, the changes you made will not appear in the user interface until you have reactivated the templates. This is because WaveMaster reads the PLACE templates only once when it is started. To activate the PLACE templates, you can either kill the WaveMaster process and restart it, or use the server configuration tool WaveSetup. In the latter case, access the "Appearance" page and click on the "Reread Templates" button. See the *Hyperwave Administrator's Guide* for instructions on how to access WaveSetup.

1.1.3.21 USER TEMPLATES WITH PLACE

In Hyperwave, the master template file resides in the WaveMaster home directory and its formatting commands are usually applied to every object accessed on the server. However, you can override the templates to give specific objects a different appearance.

ACTIVATING USER TEMPLATES FOR YOUR SERVER

By default, user templates are disabled, which means that they can be uploaded to the server but they have no effect. User templates can be activated either for all users of the server or for certain user groups using WaveSetup. This is done as follows:

1. Access WaveSetup with a browser with the URL `http://`<your_server_name>`:9999`
2. Select WaveSetup's **Appearance** tab.
3. Click on the **Enable** checkbox to allow all users to use user templates, or enter group names in the **Groups** field to allow only members of the specified groups to use user templates.
4. Click on the **Update general settings** button to make the changes in Hyperwave's configuration file.
5. **Stop and restart your server** by clicking on the **Restart server now** button. This must be done for the changes to take effect.

UPLOADING USER TEMPLATES

To use your own templates, do the following:

1. Insert all the templates (including the `include` directory) into the Hyperwave Information Server with the `hwinsdoc` tool, giving them the **MimeType** attribute `text/plain` (using option `-mime text/plain`). Then give the master template a **Name** attribute, e.g. `mylayout/master.html`.
2. Give each template a **Name** attribute. Use `%%hyperinclude "`*name*`"%%` instead of `%%include%%` to include templates from the server in the master template. If you use the normal `%%include%%` statement instead of `%%hyperinclude%%` in the serverside template, the original template from the `wavemaster` directory will be included. This is useful if you want to change only one aspect of the interface but use the default templates otherwise. You may, for example, store a new master template on the server where only one `%%include%%` is changed to a `%%hyperinclude%%` to change, for instance, only the footer.
3. Add the attribute **PLACETemplate** with the name of the master template as value to the objects that should have the new look (if you want a whole collection with the same layout, every object in this collection must have this attribute).

INCLUDING TEMPLATES FROM THE FILE SYSTEM IN USER TEMPLATES

Starting with version 2.6 of Hyperwave, new templates are installed in a directory corresponding to the version number rather than being installed directly in the `wavemaster` directory. Thus the old templates will no longer be moved to a time-stamped directory after version 2.6 (see also *Hyperwave Installation Guide*). For example, for version 2.6, templates are installed in the `wavemaster/v2.6` directory. This means that when including templates from the file system in user templates, it is necessary to use paths such as `wavemaster/v2.6/include`, which insures that server-side templates always refer to the correct templates in the file system.

If you are upgrading from 2.5 to 2.6, note that the templates for 2.5 are put into a time-stamped directory. If you want to have your server-side templates refer to them, move the `include` directory from the `wavemaster.`*date*`.`*time* directory to the `wavemaster` directory.

1.1.3.22 IMPLEMENTING FRAMES IN HYPERWAVE

There are two possible situations where frames and PLACE interact:

1. An HTML document which happens to be a frame document (i.e. contains no `<BODY>` but rather a `<FRAMESET>`) is to be displayed by the gateway.
2. The desired layout of the navigational elements created in PLACE uses frames.

In the first case we need to know if the currently displayed HTML document is a frame document or not.

If an HTML document contains a `<FRAMESET>`, it will get an attribute **BodyType**, which will be set to "FrameSet" ("BodyType=FrameSet"). Additionally the **FrameTarget** attribute will be set to "_self" ("FrameTarget=_self").

With these attributes we can distinguish between a plain HTML and a frame document with

```
%%if object.get_attrib(BodyType)=="FrameSet"%%
   This is a Frame Document
%%else%%
   Normal case
%%endif%%
```

In the second case we can ignore this condition, because we are always displaying frames created in PLACE anyway.

In both cases we have to make sure that documents retrieved as part of a frame document are recognized. This is necessary to prevent that e.g. header and footer are displayed within a sub-frame.

If a document is being retrieved from within a frame (either by the frame itself or by following a link within frames) two placeholders will be set:

1. The current action (`%%action%%`) will be set to `%%action.framecontent%%`

```
%%if action == action.framecontent%%
        We are within a frame document
    %%else%%
        Normal case
    %%endif%%
```

2. The `%%object.frame.target%%` will be set to the target frame. When you follow a link the target frame may be specified by the `TARGET` attribute in the link (e.g. `<A HREF="something.html" TARGET=_parent>`). Using this information you can control when to refresh the complete frames displayed by the gateway and when not to.

Typically we will want to refresh the frames if `%%object.frame.target%%` is either "_top" or "_blank":

```
%%if action == action.framecontent && (object.frame.target == "_top" ||
object.frame.target == "_blank")%%
   We will refresh the complete frames as created in PLACE.
%%else%%
   Normal case
%%endif%%
```

FRAMES AND SELF-DEFINED ACTIONS

In order to pass the query-, and fragment-part of the request-URI ("#", "?"-parts) on to the user action in frames, one needs to use `%%action.recall(action)%%` rather than `%%action.call(action)%%`. Otherwise these parts of the URI will not be submitted to the self-defined action. This is necessary to make destination anchors and queries work in frames as the body is displayed in a separate self-defined action as the original request.

```
<FRAMESET ROWS="120,*,60">
<FRAME SRC="%%action.recall(header.action)%%" >
<FRAME SRC="%%action.recall(body.action)%%">
<FRAME SRC="%%action.recall(footer.action)%%" >
</FRAMESET>
```

1.1.3.23 PROVIDING ONLINE HELP

There are two possibilities to supply online help for your WaveMaster interface. One is to put the online help texts into your template file, the other to provide a help collection on your Hyperwave Information Server.

PROVIDING ONLINE HELP IN THE TEMPLATE FILE

When providing online help directly in the template file, the predefined action **action.call.help(***topic***)** must be used to access the help topics. When a help topic is accessed, the variable **action** is set to `action.help(`*topic*`)`. If you want, for example, to make a link calling the help text for the topic "options", this is expressed in PLACE as:

```
<A HREF="%%action.call.help(options)%%">[Text or image for the link]</A>
```

To then display the corresponding help text, lines such as the following must appear in your template file:

```
%%if action == action.help(options)%%
   [HTML text for help on options]
%%endif%%
```

PROVIDING A HELP COLLECTION

If you use a collection to provide help in Hyperwave, you have to set the variable `HELP_COLLECTION` in the main configuration file `.db.contr.rc` to the name of the collection on your server which contains the online help texts. Then if you use

```
<A HREF="%%action.call.help(options)%%">[Text or image for the link]</A>
```

the object with name "HELP_COLLECTION/**options**" is retrieved from the server. If this object does not exist on the server, an error is reported.

Note*: It is recommended to use the second method to keep the template file small.*

1.2 DESCRIPTION OF THE DEFAULT PLACE TEMPLATES

This section contains a brief description of the default PLACE template files that are supplied with Hyperwave. It is intended to help you find which macros in which files control which aspect of the WaveMaster interface if you want to make your own changes.

File: `about.html`

Macro: `about`

Purpose: Shows information about who contributed what to the development of the current WaveMaster version when the Hyperwave icon in the toolbar is clicked on.

File: `admin.html`

Macro: `admin`

Purpose: This macro controls the display of server status infomation and user and group administration facilities, in short, of all the functions that can be accessed by clicking on the "Admin" icon in the default release. Anonymous and identified users see only information about the version numbers of the server modules, system users see much more detailed information about the number of operations on the server, and have access to facilities for creating, editing and deleting users and groups. They can also see who is currently logged in to the server.

File: `annotations.html`

Macro: `annotations`

Purpose: This macro displays a list of links to the annotations to an object.

File: `attributes.html`

Macro: `attributes`

Purpose: This macro lists object attributes, and gives identified and system users a button which can be used to let them edit the attributes and a button which lets them display the incoming and outgoing links of the current object. When the "Edit" button is clicked on, the `editattributes` action in the file `editattributes.html` is called.

File: `body.html`

Purpose: This file contains various macros which display documents, collections and clusters.

File: `buttons.html`

Purpose: Contains several macros whose purpose is to display a particular icon.

File: `buttons_java.html`

Macro: `button_permanent_java`

Purpose: This macro is called in the `header` macro.

File: `colors.html`

Macros: `backcol_d, backcol_l, mandatory, recommended`

Purpose: Defines colors for various aspects of the interface.

File: `colllistingselectable.html`

Macro: `coll_listing_selectable`

Purpose: Displays the collection list with checkboxes next to each item. This is used by functions such as copy, move, etc.

File: `copyobjects.html`

Macro: `copyobjects`

Purpose: Displays the form that allows you to submit a request to copy one or more objects to a different collection.

File: `copyobjectswizard.html`

Macro: `copyobjects`, `copyobjects2`

Purpose: The "copy" page of the Modify Objects Wizard.

File: `customindexedattributes.html`

Macros: `customindexedattributes_rows, customindexedattributes_select, customattribute, customattribute_edit`

Purpose: `customindexedattributes_rows` handles the insertion of custom indexed attributes when inserting a new object. `customindexedattributes_select` handles the editing of custom indexed attributes. `customattribute` handles the input of custom attributes during upload. `customattribute_edit` handles editing of custom attributes.

File: `deletelinks.html`

Macro: `deletelinks`, `deletelinks2`

Purpose: Displays the page for deleting links from text.

File: `deleteobjectswizard.html`

Macro: `deleteobjects`, `deleteobjects2`

Purpose: The "delete" page of the Modify Objects Wizard.

File: `delgroup.html`

Macro: `delgroup`

Purpose: Displays the form that allows you to submit a request to delete one or more user groups from the server.

File: `deluser.html`

Macro: `deluser`

Purpose: Displays the form that allows you to submit a request to delete one or more users from the server.

File: `docreferences.html`

Macro: `docreferences`

Purpose: Displays the parents and children and the incoming and outgoing links of a given object. This function is accessed through the attributes window.

File: `duplicatedocumentwizard.html`

Macros: `duplicatedocument`, `duplicatedocument2`

Purpose: The "duplicate" page of the Wizard for modifying text.

File: `duplicateobjectwizard.html`

Macros: `duplicateobjects`, `duplicateobjects2`

Purpose: The "duplicate" page of the Modify Object Wizard.

File: `editattributes.html`

Macro: `editattributes`

Purpose: Displays the form which lets identified users submit edited attribute values to the server.

File: `editgroupattributes.html`

Macro: `editgroupattributes`

Purpose: Displays the form which lets the server administrator submit edited group attribute values to the server.

File: `edittext.html`

Macro: `edittext`

Purpose: Displays the form which allows identified users to edit HTML or plain text objects on the server.

File: `edituserattributes.html`

Macro: `edituserattributes`

Purpose: Displays the form which lets the server administrator submit edited user attribute values to the server.

File: `error.html`

Macro: `errorhandling`

Purpose: Handles errors received from the server by printing out an appropriate text.

File: `exsearchform.html`

Macro: `exsearchform`

Purpose: Template which submits the search query used for simple and extended search when using the native search engine.

File: `exsearchresults.html`

Macro: `exsearchresults`

Purpose: Displays the results of a search and allows users to search with these results as domain.

File: `exsearchverity.html`

Macros: `search_attribute_search`, `search_filter_search`, `search_sort`, `search_filter_mimeselect`, `verity_ext_search`

Purpose: Displays the search page used with the Verity search engine.

File: `footer.html`

Macro: `footer`

Purpose: Displays various attribute information about the current object at the bottom of every page.

File: `groupuser.html`

Macro: `groupuser`

Purpose: Displays a form which allows administrators to add users to groups or remove them from them.

File: `header.html`

Macro: `header`

Purpose: Calls the macros in `header_en.html` and `header_ge.html`.

File: `header_en.html`

Macro: `header_en`

Purpose: Displays the toolbar which gives access to Hyperwave's functions at the top of every page in English.

File: `header_ge.html`

Macro: `header_ge`

Purpose: Displays the toolbar which gives access to Hyperwave's functions at the top of every page in German.

File: `help.html`

Macro: `help`

Purpose: Displays the main help screen with links to help topics.

File: `identification.html`

Macro: `identification`

Purpose: Displays a message and links to the current object or the home collection after successful identification.

File: `insertannotation.html`

Macro: `insertannotation`

Purpose: Displays a form which lets users insert annotations to an object.

File: `insertcgi.html`

Macro: `insertcgi`

Purpose: Displays a form which lets users insert CGI objects.

File: `insertcollection.html`

Macro: `insertcollection`

Purpose: Displays a form which lets users insert collections.

File: `insertdocument.html`

Macro: `insertdocument`

Purpose: Displays a form which lets users upload files to the server.

File: `inserthtml.html`

Macro: `inserthtml`

Purpose: Displays a form which lets users upload HTML texts (which are entered just before uploading, not as files) to the server.

File: `insertremotedoc.html`

Macro: `insertremotedoc`

Purpose: Displays a form which lets users insert remote documents into the server.

File: `insertresult.html`

Macro: `insertresult`

Purpose: If the server issued a warning during upload of an object, this macro displays the warning.

File: `inserttext.html`

Macro: `inserttext`

Purpose: Displays a form which lets users upload ASCII texts (which are entered just before uploading, not as files) to the server.

File: `mime_java.html`

Macro: `mime_java`

Purpose: Java code for displaying the menu for selecting a MIME type when inserting a document.

File: `mklink.html`

Macros: `createlinks`, `refinelinks`

Purpose: Displays page for making links in text and for selecting anchor starts if there is more than one option.

File: `moveobjects.html`

Macro: `moveobjects`

Purpose: Displays the form that allows you to submit a request to move one or more objects to a different collection.

File: `moveobjectswizard.html`

Macros: `moveobjects`, `moveobjects2`

Purpose: The "move" page of the Modify Object Wizard.

File: `namesuggest.html`

Macro: `name_suggest_code`

Purpose: Used when inserting a document to suggest a name attribute if the document has not been named.

File: `newannotations.html`

Macro: `newinsertannotation`, `refineannotation`

Purpose: `newinsertannotation` displays the Annotation Wizard, `refineannotation` displays a window where you can select the instance of the phrase you want to annotate in the document.

File: `newgroup.html`

Macro: `newgroup`

Purpose: Displays a form which lets system users create new groups.

File: `newuser.html`

Macro: `newuser`

Purpose: Displays a form which lets system users create new users.

File: `parents.html`

Macro: parents

Purpose: Displays a list of the parents of an object.

File: `passwordform.html`

Macro: `passwordform`

Purpose: Displays a form for changing your password.

File: `passwordresult.html`

Macro: `passwordresult`

Purpose: If a user changes his password successfully, this macro displays message and links to the global and personal home collections.

File: `preferences.html`

Macro: `preferences`

Purpose: Displays forms that let you submit your preferences for the session or change your password.

File: `querycreator.html`

Macro: `querycreatorform`, `querycreator`

Purpose: Displays the Query Object Wizard.

File: `replacedocument.html`

Macro: `replacedocument`

Purpose: Displays a form which lets you replace a document on the server with a new file.

File: `rightswizard.html`

Macro: `call_rightswizard_code`

Purpose: Java code which is used to call the rights wizard. The rights wizard can be called when inserting a new object into the server.

File: `searchuserresults.html`

Macro: `searchuserresults`

Purpose: Displays a list of the members of a given group (this option is accessed by clicking on a button when editing group attributes).

File: `seqparents.html`

Macro: `seqparents`

Purpose: Displays the list of parents of an object in a sequence.

File: `showlinks.html`

Macros: `showlinks`, `showlinks2`

Purpose: Displays the page for editing links in text.

File: `statusdefault_java.html`

Macro: `statusdefault_java`

Purpose: Java script which writes the name of the current user in the status bar of the browser being used.

File: `tabsmodifydoc.html`

Macro: `tabs_modifydoc`

Purpose: Display the tabs used in the Modify Objects Wizard.

File: `user.html`

Macro: `user`

Purpose: Displays a list of all user accounts on the server with edit buttons.

File: `userlist.html`

Macro: `userlist`

Purpose: Displays a list of all users currently logged in to the server.

File: `versioncheckin.html`

Macro: `checkinobject`

Purpose: Displays the window for checking in a single object. This window includes a field for changing the version number and giving the version a comment.

File: `versioncheckincollection.html`

Macro: `checkincollection`

Purpose: Displays the window for checking in one or more objects in a collection.

File: `versioncheckout.html`

Macro: `checkoutobject`

Purpose: Displays the window for checking out a single object.

File: `versioncheckoutcollection.html`

Macro: `checkoutcollection`

Purpose: Displays the window for checking out one or more objects in a collection.

File: `versionhistory.html`

Macro: `versionhistory`

Purpose: Displays the version history of a given version controlled document including version numbers, comments, etc.

File: `versionrevert.html`

Macro: `revertobject`

Purpose: Displays a window that lets you select a version of a version controlled document to revert to.

2 JAVASCRIPT

2.1 THE HWJS COMMAND

The hwjs command line tool is a program that interprets files that consists of pure JavaScript. With this utility it is possible to access a Hyperwave server with a script language.

First we start by writing a little JavaScript program to see how `hwjs` works and how it is invoked. We will call the file `hello.js`.

```
// this is the file hello.js
writeln(“hello world”);
```

To start `hwjs` with this file, simply enter:

```
hwjs hello.js
// prints hello world on the terminal
```

It is also possible to install `hwjs` at a common directory and invoke the script via the system loader. For this you must add a line in the script:

```
   #!/usr/local/bin/hwjs
   // this is our test program
writeln(“hello world”);
```

Now you can start the script without explicitly starting `hwjs`:

```
   hello.js
   // prints hello world on the terminal
```

2.2 JAVASCRIPT OBJECTS IN HWJS

This chapter shows which objects are present in the hwjs software and which global objects and functions are instantiated from the beginning or when a request comes in.

The following classes and objects are available in the hwjs software JavaScript interface:

Class/Object	Description
`HW_API_Attribute`	attribute
`HW_API_Object`	object
`HW_API_ObjectArray`	objectarray
`HW_API_Reason`	reason for error reporting
`HW_API_Error`	error
`HW_API_Content`	content representation

`HW_API_Server`	the communication class
`KeyValue`	key value pairs for http header
`KeyValueField`	field of keyvalue
`HTMLJavaScriptParser`	HTML parser
`SendMail`	mail facility
`File`	file access facility
`OptionParser`	Command line parsing tool
`argv`	the command line object
`environment`	Environment variables
`write()`	writes into the output document
`writeln()`	writes into the output document
`writeError()`	writes on stderr (debug only)
`read()`	reads from stdin
`readln()`	reads a line from stdin
`getpass()`	reads a hidden information (password) from the stty
`interactive()`	returns true when hwjs was executed in the command line
`exit()`	exits the program
`include()`	includes a file
`system()`	executes a command line in the shell
`sleep()`	pauses the program for a number of seconds
`crypt()`	encrypts a string
`escape()`	encodes a URL
`unescape()`	unencodes a URL

2.2.1 ARGV

DESCRIPTION The argv string array objects holds the command line options that are given when starting hwjs. Usually it is used with an `OptionParser` to process the command line.

EXAMPLES

```
//we type hwjs test.js a b c
// test.js
for (x in argv)
{
writeln(argv[x]) ;
}
// prints test.js a b c
```

2.2.2 ENVIRONMENT

DESCRIPTION The environment object is used to map the environment variables into the javascript context. It is often used when programming a CGI script with hwjs.

EXAMPLES

```
for (x in environment)
{
writeln(environment[x]) ;
}
// prints all environment variables
```

2.2.3 GLOBAL FUNCTIONS

2.2.3.1 WRITE

SYNTAX	`write(String)`
PARAMETERS	`String`
RETURNS	nothing
DESCRIPTION	Writes a string on stdout.

2.2.3.2 WRITELN

SYNTAX	`writeln(String)`
PARAMETERS	`String`
RETURNS	nothing
DESCRIPTION	Writes a string followed by a newline on stdout.

2.2.3.3 WRITEERROR

SYNTAX	`writeError(String)`
PARAMETERS	`String`
RETURNS	nothing
DESCRIPTION	Writes a string on stderr.

2.2.3.4 READ

SYNTAX	`read(Number)`	
PARAMETERS	`Number`	number of bytes to be read
RETURNS	`String`	
DESCRIPTION	Reads a number of bytes from stdin.	

2.2.3.5 READLN

SYNTAX	`readln()`
PARAMETERS	none
RETURNS	`String`
DESCRIPTION	Reads a line from stdin.

2.2.3.6 GETPASS

SYNTAX	`getpass()`
PARAMETERS	none
RETURNS	`String`
DESCRIPTION	Reads a password from the terminal without echoing the characters.

2.2.3.7 EXIT

SYNTAX	`exit(integer)`
PARAMETERS	integer (exit code)
RETURNS	`nothing`
DESCRIPTION	Exits the program.

2.2.3.8 INCLUDE

SYNTAX	`include(string)`
PARAMETERS	string
RETURNS	`nothing`
DESCRIPTION	Includes a file.

2.2.3.9 SYSTEM

SYNTAX `system(string)`

PARAMETERS string

RETURNS `integer`

DESCRIPTION Executes a command line in the shell and returns and exit code.

2.2.3.10 SLEEP

SYNTAX `sleep(int)`

PARAMETERS integer

RETURNS `nothing`

DESCRIPTION Pauses the program for a number of seconds.

2.2.3.11 CRYPT

SYNTAX `crypt(string)`

PARAMETERS string

RETURNS `string`

DESCRIPTION Encrypts a string.

2.2.3.12 ESCAPE

SYNTAX `escape(string)`

PARAMETERS string

RETURNS `string`

DESCRIPTION Encodes a URL.

2.2.3.13 UNESCAPE

SYNTAX `unescape(string)`

PARAMETERS string

RETURNS `string`

DESCRIPTION Unencodes a URL.

2.3 JAVASCRIPT IN THE WAVEMASTER TEMPLATES

JavaScript can be embedded in the PLACE templates. The PLACE templates are similar to HTML files and JavaScript is integrated via an HTML tag.

JavaScript is used in the templates to enhance the functionality of PLACE or to cover functionality which PLACE is not able to perform. The normal approach of developing a application with JavaScript in the templates is to process the request to see what should happen, then use some classes and objects to access the Hyperwave server and prepare the response to fulfill the request.

JavaScript can be embedded in HTML in 4 ways:

· The JavaScript is enclosed by <SERVER> and </SERVER> tags.

- The pure JavaScript is in its own file and is included via the include tag. <SERVER SRC="file://path/filename" >
- The JavaScript is embedded in a normal HTML tag as an attribute name with backquotes.
- The JavaScript is embedded in a normal HTML tag as an attribute value with backquotes and is automatically surrounded by doublequotes.

EXAMPLES

```
<!-- this is a HTML template -->
<H1>Welcome</H1>
<SERVER>
// this is pure JavaScript
function pref()
{
return "HREF";
}
function url()
{
return "http://hyperwave.com";
}
write("hello and welcome");
</SERVER>

<!—- invocation in a tag -->
<A 'pref()'='url()'>Go to Hyperwave</A>
<!—- a include -->
<SERVER SRC="file://test.js">
```

2.4 JAVASCRIPT OBJECTS IN WAVEMASTER VIA TEMPLATES

This chapter shows which objects are present in the WaveMaster software and which global objects and functions are instantiated from the beginning or when a request comes in.

The following classes and objects are available in the WaveMaster software JavaScript interface:

Class/Object	Description
`HW_API_Attribute`	attribute
`HW_API_Object`	object
`HW_API_ObjectArray`	objectarray
`HW_API_Reason`	reason for error reporting
`HW_API_Error`	error
`HW_API_Content`	content representation
`HW_API_Server`	the communication class
`KeyValue`	key value pairs for http header
`KeyValueField`	field of keyvalue
`HTMLJavaScriptParser`	HTML parser
`SendMail`	mail facility
`File`	file access facility
`server`	the current server object
`request`	the current request object

`response`	the current response object
`wavemaster`	wavemaster information object
`client`	the client object
`write()`	writes into the document
`writeln()`	writes into the document
`writeError()`	writes on stderr (debug only)
`crypt()`	encrypts a string
`escape()`	encodes a URL
`unescape()`	unencodes a URL

2.4.1 SERVER

DESCRIPTION The server object is an instantiation of the `HW_API_Server` class that is already identified and connected to the right server.

2.4.2 REQUEST

DESCRIPTION The request object is a global object that is created every time the user requests a specific page, and it reflects the current request, which is mostly the http request itself. It has only properties and no functions.

PROPERTY SUMMARY

Property	Type	Description
url	`String`	The requested URL
method	`String`	The method of the request (GET/POST/PUT)
version	`String`	The version of the request
ip	`String`	The IP address of the user that issued the request
host	`String`	The hostname of the user that issued the request
agent.version	`String`	Version of the user-agent
agent.versionminor	`Number`	Minor version number
agent.versionmajor	`Number`	Major version number
agent.name	`String`	Name of the user-agent
agent.architecture	`String`	Architecture of the client
field	`KeyValueField`	The HTTP header fields
content	`HW_API_Content`	The post/put data
object	`HW_API_Object`	The current requested object

2.4.3 RESPONSE

DESCRIPTION The response object holds the http header that is sent back as a response to requests. The header and headerfields are automatically set, but the programmer can override these defaults. Some of the reasons a programmer might want to change the response object are:

- To create an http error code
- To create a redirection
- To force an identification via the http mechanism

· To change the content-type of the response

These modifications are carried out by simply changing the appropriate values in the response object.

PROPERTY SUMMARY

Property	Type	Description
code	`String`	The response code (HTTP)
version	`String`	The version of the response
reason	`String`	The reason (human readable string)(HTTP)
field	`KeyValueField`	The HTTP header fields

2.4.4 WAVEMASTER

DESCRIPTION The wavemaster object is a global object which holds some information about the WaveMaster itself.

PROPERTY SUMMARY

Property	Type	Description
version	`String`	The version of the WaveMaster
protocol	`String`	The protocol used with colon (http:, https:)
port	`Number`	The port used
host	`String`	The hostname
hostname	`String`	The hostname with port if necessary

METHOD SUMMARY

Method	Description
icon	Gets the icon path for an `HW_API_Object`

2.4.4.1 METHODS

ICON

SYNTAX `icon(item)`

PARAMETERS item `HW_API_Object`

RETURNS `String`

DESCRIPTION Returns the absolute-relative URL to the icon of an HW_API_Object.

2.4.5 CLIENT

DESCRIPTION WaveMaster maintains a client object which is cookiefied and sent to the client. On the next request the client sends the cookie information back to the WaveMaster where it is converted to a client object. The client object is global and is initialized for every request. Any string property can be added and is then sent to the client. The client object has several predefined properties as shown below.

PROPERTY SUMMARY

Property	Type	Description
client.preferredmimetypes	`String`	List of MimeTypes/Quality (Syntax: see User Object: PrefMimeTypes
client.languages	`String`	One or more two-letter language tags separated by a space (e.g. "en ge fr"), representing the users preferred language choices.
client.language	`String`	The first language from the list Languages

2.4.6 GLOBAL FUNCTIONS

2.4.6.1 WRITE

SYNTAX	`write(String)`
PARAMETERS	`String`
RETURNS	nothing
DESCRIPTION	Writes a string into the current document being processed.

2.4.6.2 WRITELN

SYNTAX	`writeln(String)`
PARAMETERS	`String`
RETURNS	nothing
DESCRIPTION	Writes a string followed by a newline into the document.

2.4.6.3 WRITEERROR

SYNTAX	`writeError(String)`
PARAMETERS	`String`
RETURNS	nothing
DESCRIPTION	Writes a string on stderr which is recorded into the logfile. This may be useful for debugging purpose only.

2.4.6.4 CRYPT

SYNTAX	`crypt(string)`
PARAMETERS	string
RETURNS	`string`
DESCRIPTION	Encrypts a string.

2.4.6.5 ESCAPE

SYNTAX	`escape(string)`
PARAMETERS	string
RETURNS	`string`
DESCRIPTION	Encodes a URL.

2.4.6.6 UNESCAPE

SYNTAX	`unescape(string)`
PARAMETERS	string
RETURNS	`string`
DESCRIPTION	Unencodes a URL.

2.5 JAVASCRIPT OBJECT MODEL

This chapter covers the different native objects in the Hyperwave JavaScript world.

All `HW_API` objects are derived from the language independent Hyperwave API specification. The other objects are additional objects that are either used by WaveMaster or by hwjs (the command line tool). All objects are explained here, but some are not available in WaveMaster or hwjs.

HW_API_String and HW_API_StringArray are mapped to the native JavaScript String and Array classes.

All count() methods in the language independent API specification are mapped to length properties in JavaScript.

2.5.1 HW_API_ATTRIBUTE

CREATED BY `HW_API_Attribute` constructor:

`new HW_API_Attribute(key, value)`

`new HW_API_Attribute(key, values)`

PARAMETERS

key	`String`
value	`String`
values	`Array` of `String`s

DESCRIPTION `HW_API_Attribute` holds one key value(s) pair of an `HW_API_Object`. This is one pair of meta-information that is represented by `HW_API_Object`. It is used to access single meta-information of an object or to create a new object out of a collection of `HW_API_Attributes`.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `HW_API_Attribute`
key	The key of the attribute
value	A value of the attribute
values	An `Array` of values of the attribute

METHODS SUMMARY none

2.5.1.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

KEY

The key of the attribute

EXAMPLES

```
x = new HW_API_Attribute(“Name”,new Array(“foobar”,”foobar2”));
writeln(x.key);
// the result is Name
```

VALUE

This property reflects one value (`String`) of possible more values.

EXAMPLES

```
x = new HW_API_Attribute(“Name”,new Array(“foobar”,”foobar2”));
writeln(x.value);
// the result is foobar or foobar2
```

VALUES

This property reflects all values (`Array` of `String`s) of an attribute.

EXAMPLES

```
x = new HW_API_Attribute(“Name”,new Array(“foobar”,”foobar2”));
writeln(x.values);
// the result is foobar, foobar2
```

2.5.2 HW_API_OBJECT

CREATED BY `HW_API_Object` constructor:

`new HW_API_Object(attr1,attr2,attr3,...);`

PARAMETERS attr1...attrn `HW_API_Attributes` that construct the whole object.

The constructed object consists of this attributes (key value(s) pairs).

DESCRIPTION `HW_API_Object` is a basic type for the Hyperwave JavaScript interface. It holds the whole meta-information of a document. It is used to display, alter and insert objects on the server.

PROPERTY SUMMARY All inserted `HW_API_Attribute` objects are mirrored into properties of an `HW_API_Object` and result in their corresponding value property. This means that if an object has an `HW_API_Attribute` with a key of “Name” and a value of “foobar” then the `HW_API_Object` has a property of “Name” with its value (`String`) “foobar”. If an `HW_API_Attribute` has more than one value a random one is chosen. The properties can be assigned to have new values. If an array of strings is assigned to a property then all strings are values of the property.

Property	Description
prototype	Defines a property that is shared by all `HW_API_Object` objects
length	The number of HW_API_Attributes in the HW_API_Object
[“attribute name”]	The `HW_API_Attribute` in the `HW_API_Object`

METHOD SUMMARY

Method	Description
attribute	Returns the attribute at a specific position
insert	Inserts a new `HW_API_Attribute` into the `HW_API_Object`
title	Returns the title with a specific language(s)
attributeEditable	Returns true if attribute is editable if the object would be editable

2.5.2.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

LENGTH

This property represents the number of `HW_API_Attribute` objects that reside in an object.

[“ATTRIBUTE NAME”]

All `HW_API_Attribute` objects are mapped into properties of the `HW_API_Object`. If an object has an `HW_API_Attribute` with key “foo” and value “bar” then the objects property “foo” is “bar”

```
x=new HW_API_Object(new HW_API_Attribute(“foo”,”bar”)) ;
writeln(x.foo) ;
writeln(x[“foo”]) ;
```

2.5.2.2 METHODS

ATTRIBUTE

SYNTAX `attribute(Number)`

`attribute(String)`

PARAMETERS

`Number`	index of the `HW_API_Attribute` which should be returned
`String`	key of the `HW_API_Attribute` which should be retrieved

RETURNS `HW_API_Attribute`

DESCRIPTION Gets a specific `HW_API_Attribute` either by index or by key.

EXAMPLES none

INSERT

SYNTAX `insert(item)`

PARAMETERS item `HW_API_Attribute`

RETURNS nothing

DESCRIPTION Inserts a new `HW_API_Attribute` into the current `HW_API_Object`.

EXAMPLES none

TITLE

SYNTAX `title(String)`

PARAMETERS `String` A non character separated list of desired languages.

RETURNS `String`

DESCRIPTION An HW_API_Object can have several titles in different languages. The user normally prefers to have the titles in a specific language or when not possible in another one.

EXAMPLES writeln(obj.title("ge en es") ;

// prints either german title if available or english title if available or spanish title if available

ATTRIBUTEEDITABLE

SYNTAX `attributeEditable(attr, username, system)`

PARAMETERS

`attr`	`String/HW_API_Attribute`	Attribute(name) which should be checked.
`username`	`String`	The username of the current user.
`system`	`Boolean`	Is the user a system user.

RETURNS `Boolean`

DESCRIPTION Determines if an attribute in an object is editable when the username and the system flag is given. It does not check if the object itself is editable, only if the given attribute would be editable under the given context.

EXAMPLES

```
// assume that server is a HW_API_Server
out=server.object({objectidentifier:"rootcollection"});
if (out.object.attributeEditable("Title","tvollmer",true))
{
writeln("I can edit the title");
}
```

2.5.3 HW_API_OBJECTARRAY

CREATED BY `HW_API_ObjectArray` constructor:

`new HW_API_ObjectArray(obj1,obj2,obj3,...);`

PARAMETERS obj1...objn The different objects that make up the whole `HW_API_ObjectArray`.

DESCRIPTION An `HW_API_ObjectArray` holds a given (sorted) set of `HW_API_Objects` that could represent some special objects such as the children or parents of another object.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `HW_API_ObjectArray` objects
length	The number of `HW_API_Objects` in the array
[0]...[length-1]	The `HW_API_Objects` in the array

METHOD SUMMARY

Method	Description
sort	Sorts an `HW_API_ObjectArray`
append	Appens a `HW_API_Object` to the `HW_API_ObjectArray`

2.5.3.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

LENGTH

The number of `HW_API_Objects` in the array.

[0]...[LENGTH-1]

The arbitrary elements of an `HW_API_ObjectArray`.

2.5.3.2 METHODS

SORT

SYNTAX

```
sort(sortorder,languages[,typeinfo])
sort(sortorder,strong,languages[,typeinfo])
```

PARAMETERS

sortorder	`HW_API_String`	The specific sortorder
languages	`HW_API_String` preferred	A non character separated list of languages
strong	`Boolean`	Is the sortorder strong or weak
typeinfo	`HW_API_Object`	Additional Information (optional)

RETURNS nothing

DESCRIPTION The current `HW_API_ObjectArray` with its `HW_API_Object` objects is sorted according to the sortorder and the languages given.

When a `HW_API_ObjectArray` is retrieved via an `HW_API_Server` call then the sortorder is internally set to a "good guess". For example if you get the children of a collection the internal sortorder is set to the sortorder of the collection. When the sort() call without strongness is used the given sortorder is a "weak" one. This means that the interface uses e.g. the sortorder from the collection, but if there is none the weak one will be taken. If the sortorder is set to "strong" it will override any internal values set.

The typeinfo object is set internally, but if one needs to build his own `HW_API_ObjectArray` from the scratch it can be given optional.

EXAMPLES

```
// assume that server is a HW_API_Server object
```

```
out = server.children({objectidentifier:”rootcollection”}) ;
// now sort, if the rootcollection has a SortOrder it will
// be taken else the sortorder is set  to “#T”
out.objects.sort(“#T”,”en”) ;
printObjectArray(out.objects) ; // a user function
// now force a sort for creation Time
out.objects.sort(“C”,”en”) ;
printObjectArray(out.objects) ;
```

APPEND

SYNTAX `append(item)`

PARAMETERS item `HW_API_Object` to be appended

RETURNS nothing

DESCRIPTION Appends a `HW_API_Object` to a `HW_API_ObjectArray`

EXAMPLES none

2.5.4 HW_API_REASON

CREATED BY All `HW_API_Error` objects that are created can have `HW_API_Reason` objects in it.

DESCRIPTION A reason represents an incident. This can be an error, warning or message. During the execution of native functions a lot can happen. Everything that happens and is worth being recorded is collected as reasons in an error object. If an error occurred the last reason is an error otherwise no error occurred but there can be reasons in the `HW_API_Error` object returned. These reasons can be presented to the user.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `HW_API_Reason` objects
type	The type of reason (Error, Warning, Message)
source	The source that issued the reason
code	The code issued

METHOD SUMMARY

Method	Description
description	Returns a verbose description of the reason

2.5.4.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

TYPE

The type of reason:

0:	Error
1:	Warning
2:	Message

SOURCE

The source which issued the reason.

CODE

The code of the reason

2.5.4.2 METHODS

DESCRIPTION

SYNTAX `description(String)`

PARAMETERS `String` the language in which the description should come along

RETURNS `String`

DESCRIPTION Returns are human readable forms of the reason in a specific language.

EXAMPLES

```
for (x in out.error)
{
writeln(out.error[x].description("en")) ;
}
```

2.5.5 HW_API_ERROR

CREATED BY All `HW_API_Server` methods

DESCRIPTION All calls that access the Hyperwave Server return an object that has a property which holds an `HW_API_Error`. If the specific call failed the error is set and can be retrieved and printed in a verbose manner. An `HW_API_Error` object can hold several `HW_API_Reason` objects. If a warning or message occurred during a call they are simply appended to the error object as a new reason. If we get an error with a length this means that something has happened during the call. If error() returns true a real error occurred otherwise only some warnings or messages occurred but the operation was successful.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `HW_API_Error` objects
length	The number of `HW_API_Reaso`n objects in the `HW_API_Error`

METHOD SUMMARY

Method	Description
error	Returns whether an error occurred or not

2.5.5.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

LENGTH

The number of `HW_API_Reason` objects an `HW_API_Error` is holding. A length of non zero means that some reasons are present after a specific operation. It need not be an error, but could also be a warning.

2.5.5.2 METHODS

ERROR

SYNTAX `error()`

PARAMETERS none

RETURNS `Boolean`

DESCRIPTION If an error is in the `HW_API_Error`, error() returns true otherwise false. An `HW_API_Error` object can have a length but need not to have an error in it.

EXAMPLES

```
out=server.hwStat();
if (out.error.error())
{
writeln(“error”) ;
}
for (x in out.error)
{
writeln(out.error[x].description(“en”)) ;
}
```

2.5.6 HW_API_CONTENT

CREATED BY `HW_API_Content` constructor:

`new HW_API_Content(String)`

or by `HW_API_Server` calls that retrieve a content.

DESCRIPTION `HW_API_Content` is an encapsulation of some piece of data. This data can reside in memory but can also stay somewhere else (on file). `HW_API_Content` is used to transfer data from the server to somewhere else or vice versa.

A content can be read only once! The length is always the same for one given content, it does not adjust during the read operation. To see how many bytes you have already read use consumed() to determine how many bytes are already read.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `HW_API_Error` objects
length	The size of the content in bytes

METHOD SUMMARY

Method	Description
error	Returns whether an error occurred or not
read	Reads a number of bytes into a string
eof	Returns true if end of content
consumed	Returns the number of bytes already read
ok	Returns true if everything is ok with the content

2.5.6.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

LENGTH

Every `HW_API_Content` object has a length. If the length is known to the system it is set to the correct value, otherwise it is set to –1.

2.5.6.2 METHODS

ERROR

SYNTAX `error()`

PARAMETERS none

RETURNS HW_API_Error

EXAMPLES none

READ

SYNTAX `read(Number)`

PARAMETERS `Number` the number of bytes to read

RETURNS `String`

DESCRIPTION Reads the number of bytes or less if at end of content and returns them as string.

If the wished length is –1 the whole content is read in.

Attention: The language independent specification of read() is slightly different. In JavaScript read() returns exactly the wanted size or an empty string if eof is reached.

EXAMPLES None

EOF

SYNTAX `eof()`

PARAMETERS none

RETURNS `Boolean`

DESCRIPTION Returns true if the end of content is reached.

EXAMPLES None

CONSUMED

SYNTAX `consumed()`

PARAMETERS none

RETURNS `Number`

DESCRIPTION Returns the number of bytes already read from the content.

EXAMPLES none

2.5.7 HW_API_SERVER

CREATED BY `HW_API_Server` constructor:

```
new HW_API_Server(String,Number)
new HW_API_Server(String)
```

or in the WaveMaster, the global object `server`.

PARAMETERS

`String`	hostname of the server to be contacted
`Number`	port (default is 418)

DESCRIPTION `HW_API_Server` is the communication interface for the Hyperwave server. It has certain functions to communicate with the server, alter data etc.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `HW_API_Server` objects

METHOD SUMMARY

Method	Description
object	retrieves an object from the server
children	retrieves the children of an object
parents	retrieves the parents of an object
srcAnchors	retrieves the source anchors of an object
dstAnchors	retrieves the destination anchors of an object
find	searches objects

identify	identify on the Hyperwave server
remove	removes objects
content	get the content of a document
html	get the content of an HTML document
copy	copies an object
link	creates a link to an object
move	moves an object
lock	locks an object
unlock	unlocks an object
insert	inserts an object and content
replace	replaces an object and/or content
checkIn	checks in an object (versioncontrol)
checkOut	checks out an object (versioncontol)
revert	reverts to an previous version (versioncontrol)
history	gets the version history (versioncontrol)
objectByAnchor	retrieves an object by its anchor
dstOfSrcAnchor	retrieves the destination of a source anchor
srcsOfDst	retrieves all sources pointing to destination
user	retrieves its own user object
userlist	retrieves currently logged in users
hwStat	retrieves statistics of hgserver
dcstat	retrieves statistics of dcserver
dbStat	retrieves statistics of dbserver
ftStat	retrieves statistics of ftserver
info	retrieves information about the server configuration

2.5.7.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

2.5.7.2 METHODS

All methods of an `HW_API_Server` take one argument and return one object. These two objects have certain well known named properties. The parameters section lists the properties that the parameter object must have. Optional parameters are enclosed by brackets. The Returns section shows what properties the return object has.

There are certain special parameters which are explained here.

OBJECTIDENTIFIERS The object identifiers are either the name of the object or its GOid. It is used to address a specific object.

ATTRIBUTESELECTOR It is possible to request meta-information that is not normally part of an object returned in an operation. If the user wants this information he/she must specify the names of these attributes.

The following attributes can be requested for all methods.

HW_EffectiveAccess: Tells the user if he/she has write or read access to the object. Its value is either `READ_ACCESS` or `WRITE_ACCESS`.

HW_ExtendedAccess: Tells the user if he/she has lock access to the object, i.e. no one else is currently locking the object. Its value is `LOCK_ACCESS`.

HW_UserLock: This attribute exists when an object has been locked by a user. Its value is the second part of the GOid of the lock object (object which stores information about the lock on the object, e.g. who locked the object and when).

HW_SessionLock: This attribute exists while an object is session locked (this is only the case during the time a changed document is being uploaded). Its value is the user ID from the list of users who are online.

The following attributes can be requested when using the FIND method.

FullTextQuery: Value is `true` if the object was found using a fulltext query.

KeyQuery: Value is `true` if the object was found using a keyquery.

VERSION A version is a string that contains either a special keyword that represents a specific version or a version number. A version number consists of a major number and a minor number which are separated by a dot. For example “3.12” is a version string with “3”as major number and “12” as minor number.

The special keywords are:

· `HW_VERSION_EXPERIMENTAL`	Retrieves the experimental version
· `HW_VERSION_COMMITED`	Retrieves the last committed version
· `HW_VERSION_ACTUAL`	Retrieves the actual version

OBJECT

SYNTAX `object(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[attributeselector]	`Array` of `String`s	Names of additional attributes
[version]	`String`	Version of the object to retrieve

RETURNS

Name of Property	Type	Description
object	`HW_API_Object`	The requested object
error	`HW_API_Error`	Error information

DESCRIPTION object() retrieves an object which is requested via an objectidentifier. Only the meta information is fetched. To get the content (the document) use content().

EXAMPLES `out=server.object({objectidentifier:”rootcollection”});`

CHILDREN

SYNTAX `children(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[attributeselector]	`Array` of `String`s	Names of additional attributes
[objectquery]	`String`	Query that the children must match

RETURNS

Name of Property	Type	Description
objects	`HW_API_ObjectArray`	The requested objects
error	`HW_API_Error`	Error information

DESCRIPTION retrieves the children of a container object.

EXAMPLES `out=server.children({objectidentifier:”rootcollection”});`

PARENTS

SYNTAX `parents(parameterobject)`

PARAMETERS

Name of Property	Type	Description

objectidentifier	`String`	Name or GOid of object
[attributeselector]	`Array` of `String`s	Names of additional attributes
[objectquery]	`String`	Query that the children must match

RETURNS

Name of Property	Type	Description
objects	`HW_API_ObjectArra`	The requested objects
error	`HW_API_Error`	Error information

DESCRIPTION parents() retrieves the parents of an object.

EXAMPLES `out=server.parents({objectidentifier:”test/test4”});`

SRCANCHORS

SYNTAX `srcAnchors(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[attributeselector]	`Array` of `String`s	Names of additional attributes
[objectquery]	`String`	Query that the children must match

RETURNS

Name of Property	Type	Description
objects	`HW_API_ObjectArray`	The requested objects
error	`HW_API_Error`	Error information

DESCRIPTION `srcAnchors()` retrieves the source anchors of an object.

EXAMPLES `out=server.srcAnchors({objectidentifier:”test/test.html”});`

DSTANCHORS

SYNTAX dstAnchors(parameterobject)

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[attributeselector]	`Array` of `String`s	Names of additional attributes
[objectquery]	`String`	Query that the children must match

RETURNS

Name of Property	Type	Description
objects	`HW_API_ObjectArray`	The requested objects
error	`HW_API_Error`	Error information

DESCRIPTION `dstAnchors()` retrieves the destination anchors of an object.

EXAMPLES `out=server.dstAnchors({objectidentifier:”test/test.html”});`

FIND

SYNTAX `find(parameterobject)`

PARAMETERS

Name of Property	Type	Description
[keyquery]	`String`	The key query
[fulltextquery]	`String`	The fulltext query
[objectquery]	`String`	The object query
[scope]	`Array` of `String`s	the search scope
languages	`Array` of `String`s	the search languages

RETURNS

Name of Property	Type	Description
objects	`HW_API_ObjectArray`	The requested objects

error	`HW_API_Error`	Error information

DESCRIPTION find() performs a search on the Hyperwave server. Either a keyquery or a fulltextquery must be present. If both are present, then they will be executed in sequence. The found objects will be then filtered by the objectquery. If no scope is given the entire server is searched. The languages in which the search will be committed must be given in any case.

EXAMPLES

```
out=server.find({keyquery:"Title=Hyperwave*",languages:Array("en")});
```

IDENTIFY

SYNTAX `identify(parameterobject)`

PARAMETERS

Name of Property	Type	Description
Username	`String`	username
Password	`String`	password

RETURNS

Name of Property	Type	Description
Error	`HW_API_Error`	Error information

DESCRIPTION Identification for a Hyperwave server.

The identify() method cannot be used on the global server object in the WaveMaster!

EXAMPLES

```
serva= new HW_API_Server("hw.hyperwave.com");
out=serva.identify({username:"tvollmer",password:"nopass"});
```

REMOVE

SYNTAX `remove(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[parentidentifier]	`String`	Name or GOid of the parent
[mode]	`Number`	Mode of removal

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information

DESCRIPTION Removes object from the Hyperwave server. If an object is the last instance it will be removed physically. If object is a container all children will be removed. The parent must be set when the object is instantiated more than once and mode is not set to remove physical. No parent must be set when removing user or groups.

Mode can be set as follows

0	Normal remove mode (same as no mode)
1	remove physical, object is removed even when it is instantiated more than once
2	delete links, all anchors that are somehow linked to the object are removed when possible

The mode is a bitmask which means that modes can be combined. A mode of 3 means that objects are removed physically and all links are removed.

EXAMPLES

```
out=server.remove({objectidentifier:"test"});
```

CONTENT

SYNTAX `content(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
mode	`Number`	Mode

RETURNS

Name of Property	Type	Description
content	`HW_API_Content`	The requested content
error	`HW_API_Error`	Error information

DESCRIPTION content() retrieves the content of a specific document.

Modes

This only affects HTML documents and specifies which links are put into the document

0 All links are inserted

1 Only the reachable links (the ones that the current user can see) are inserted. This is the default behavior

2 No links are inserted

EXAMPLES

```
out=server.content({objectidentifier:"test/test.txt"});
write(out.content.read(out.content.length));
```

HTML

SYNTAX `html(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object

RETURNS

Name of Property	Type	Description
body	`String`	The body of the HTML document
bodyattributes	`String`	Body attributes of the HTML document
doctype	`String`	Document type of the HTML document
head	`String`	The head of the HTML document
error	`HW_API_Error`	Error information

DESCRIPTION html() retrieves the content of an HTML document and splits it up in the properties doctype, head, bodyattributes, and body. See the example on how to use these properties to exactly rebuild an HTML document as it was originally inserted on the server.

EXAMPLES

```
out=server.html({objectidentifier:"test/test.html"});
writeln(out.doctype);
writeln("<HTML>");
writeln("<HEAD>");
writeln(out.head);
writeln("</HEAD>");
writeln("<BODY " + out.bodyattributes + ">");
writeln(out.body);
writeln("</BODY>");
writeln("</HTML>");
```

COPY

SYNTAX `copy(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
destinationparentidentifier	`String`	Name or Goid of the destination
[attributeselector]	`Array` of `String`s	Names of additional attributes

RETURNS

Name of Property	Type	Description
object	`HW_API_Object`	The copied object
error	`HW_API_Error`	Error information

DESCRIPTION Copies an object and its content into a specific container and returns the new object.

EXAMPLES

```
out=server.copy({objectidentifier:”test/test.html”,destinationparentid
entifier:”rootcollection”});
```

LINK

SYNTAX `link(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
destinationparentidentifier	`String`	Name or GOid of the destination container

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information

DESCRIPTION Links the object into another container.

EXAMPLES

```
out=server.link({objectidentifier:”test/test.html”,destinationparentid
entifier:”rootcollection”});
```

MOVE

SYNTAX `move(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
sourceparentidentifier	`String`	Name or GOid of the source container
destinationparentidentifier	`String`	Name or GOid of the destination container

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information

DESCRIPTION Moves an object from one container in another one. An object can have multiple parents (see link) so it is necessary to specify from where to move.

EXAMPLES

```
out=server.move({objectidentifier:”test/test.html”,sourceparentidentif
ier:”test”,destinationparentidentifier:”rootcollection”});
```

LOCK

SYNTAX `lock(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[mode]	`Number`	Mode

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information

DESCRIPTION Locks an object. Only the person who locked it can alter the object. Only children of collections are locked when invoked recursively.

Modes

0	Normal (default if no mode is given)
1	recursive (descends into subcontainers)

EXAMPLES

```
out=server.lock({objectidentifier:”test/test.html”});
```

UNLOCK

SYNTAX `unlock(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
mode	`Number`	Mode

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information

DESCRIPTION Unlocks an object. Only the person who locked the object can unlock the object except system users.

Modes

0 normal (default)

1 recursive (descends into subcontainers)

EXAMPLES

```
out=server.unlock({objectidentifier:”test/test.html”});
```

INSERT

SYNTAX `insert(parameterobject)`

PARAMETERS

Name of Property	Type	Description
object	`HW_API_Object`	Object to insert
[parameters]	`HW_API_Object`	Parameters for insertion
[content]	`HW_API_Content`	content to insert
[attributeselector]	`Array` of `String`s	selector for inserted object
mode	`Number`	Mode

RETURNS

Name of Property	Type	Description
object	`HW_API_Object`	object inserted
error	`HW_API_Error`	Error information

DESCRIPTION This operation is used to insert objects and their optional content into the Hyperwave server.

Modes

0 Normal (default if no mode is given)

1 Force checkin (check in even if document not yet under version control)

2 Automatic checkout

EXAMPLES

```
insobj  = new HW_API_Object(
new HW_API_Attribute(“Type”,”Document”),
new HW_API_Attribute(“DocumentType”,”text”),
new HW_API_Attribute(“Title”,”en:Testing javascript”),
new HW_API_Attribute(“MimeType”,”text/plain”));
toinstext = “This is a plain text” ;
out = stub.insert({”object”:insobj,”parameters”:
new HW_API_Object(
new HW_API_Attribute(“Parent”,”tvollmer”)),
“content”:HW_API_Content(toinstext)}) ;
```

REPLACE

SYNTAX `replace(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object to be replaced
object	`HW_API_Object`	Object that replaces the old object
[parameters]	`HW_API_Object`	Parameters for replace
[content]	`HW_API_Content`	content for replace
[attributeselector]	`Array` of `String`s	selector for replaced object
mode	`Number`	Mode

RETURNS

Name of Property	Type	Description
object	`HW_API_Object`	object replaced
error	`HW_API_Error`	Error information

DESCRIPTION

This operation is used to replace objects and their optional content in a Hyperwave server.

Modes

0 Normal (default if no mode is given)

1 Force checkin (check in even if document not yet under version control)

2 Automatic checkout

4 Automatic checkin

EXAMPLES

```
// assume that server is a HW_API_Server object and we are
// identified as a valid user
out=server.object({objectidentifier:”tvollmer/test.html”}) ;
// override the current title
out.object[“Title”] = “en:Till testing” ;

server.replace({objectidentifier:”tvollmer/test.html”,
                object:out.object}) ;

// replace the content
mytext=”<HTML><HEAD><TITLE>Hello</TITLE></HEAD><BODY>Test</BODY></HTML
>” ;
server.replace({objectidentifier:”tvollmer/test.html”,
                content:HW_API_Content(mytext)}) ;

// you can replace content and object in one call
server.replace({objectidentifier:”tvollmer/test.html”,
                object:out.object,
                content:HW_API_Content(mytext)}) ;
```

CHECKIN

SYNTAX `checkIn(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[mode]	`Number`	Mode
[version]	`String`	Version
[comment]	`String`	comment for this version
[objectquery]	`String`	only matching objects will be checked in

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information

DESCRIPTION

Either one object or a hierarchy of objects will be checked in with a specific version and comment. Additionally, an object query can be given to select certain objects that will be checked in. If no version is given the minor version number is incremented automatically.

Modes

0 Normal (default if no mode is given)

1 recursive (descends into subcontainers)

2 Force checkin (check in even if document not yet under version control)

4 Revert (if no changes made, reverts to last committed version and deletes experimental version)

EXAMPLES

```
out=checkIn({objectidentifier:”test/test.html”});
out=checkIn({objectidentifier:”myhome/business”,mode:3});
```

Checks in or makes an initial version of all documents contained recursively in the collection “/myhome/business”.

CHECKOUT

SYNTAX `checkOut(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[mode]	`Number`	Mode
[objectquery]	`String`	only matching objects will be checked out

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information

DESCRIPTION Either one object or a hierarchy of objects will be checked out. Additionally, an object query can be given to select certain objects that will be checked out.

Modes

0 Normal (default if no mode is given)

1 recursive (descends into subcontainers)

EXAMPLES `out=checkOut({objectidentifier:”test/test.html”});`

REVERT

SYNTAX `revert(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[mode]	`Number`	Mode
version	`String`	Version
[objectquery]	`String`	only matching objects will be reverted

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information

DESCRIPTION Either one object or a hierarchy of objects will be reverted to a specific version. Additionally, an object query can be given to select certain objects that will be reverted.

Modes

0 Normal (default if no mode is given)

1 recursive (descends into subcontainers)

EXAMPLES `out=revert({objectidentifier:”test/test.html”,”version”:”1.1”});`

HISTORY

SYNTAX `history(parameterobject)`

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[attributeselector]	`Array of Strings`	Names of additional attributes
[objectquery]	`String`	only matching objects will be showed

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information

objects	`HW_API_ObjectArra`	The history objects

DESCRIPTION The version history of a specific object will be retrieved. Additionally an object query can be given to select certain objects. Also additional attributes can be requested.

EXAMPLES
```
out=history({objectidentifier:”test/test.html”});
```

OBJECTBYANCHOR

SYNTAX
```
objectByAnchor(parameterobject)
```

PARAMETERS

Name of Property	Type	Description
Objectidentifier	`String`	Name or GOid of object
[attributeselector]	`Array of Strings`	Names of additional attributes

RETURNS

Name of Property	Type	Description
Error	`HW_API_Error`	Error information
Object	`HW_API_Object`	The object

DESCRIPTION Gets the corresponding object for an anchor.

EXAMPLES
```
out=objectByAnchor({objectidentifier:”0x00000000 0x00000030”});
```

DSTOFSRCANCHOR

SYNTAX
```
dstOfSrcAnchor(parameterobject)
```

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[attributeselector]	`Array of Strings`	Names of additional attributes

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information
object	`HW_API_Object`	The object

DESCRIPTION Gets either a document object or an anchor object that is referenced by the source anchor.

EXAMPLES
```
out=dstOfSrcAnchor({objectidentifier:”0x00000000 0x00000030”});
```

SRCSOFDST

SYNTAX
```
srcsOfDst(parameterobject)
```

PARAMETERS

Name of Property	Type	Description
objectidentifier	`String`	Name or GOid of object
[attributeselector]	`Array of Strings`	Names of additional attributes
[objectquery]	`String`	only matching objects will be retrieved

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information
objects	`HW_API_ObjectArray`	The objects

DESCRIPTION Gets all objects that point to a specific destination object.

EXAMPLES
```
out=srcsOfDst({objectidentifier:”0x00000000 0x00000031”});
```

USER

SYNTAX
```
user(parameterobject)
user()
```

PARAMETERS

Name of Property	Type	Description
[attributeselector]	`Array of Strings`	Names of additional attributes

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information
object	`HW_API_Object`	the user object

DESCRIPTION Gets the own user object.

EXAMPLES

```
out=user();
writeln(out.object[“UName”]) ;
```

USERLIST

SYNTAX

```
userlist(parameterobject)
userlist()
```

PARAMETERS

Name of Property	Type	Description
[attributeselector]	`Array of`	Names of additional attributes

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information
objects	`HW_API_ObjectArra`	The objects

DESCRIPTION Gets the currently logged in users as objects.

EXAMPLES

```
out=userlist();
```

HWSTAT, DBSTAT, DCSTAT, FTSTAT

SYNTAX

```
hwStat(parameterobject)
dbStat(parameterobject)
dcStat(parameterobject)
ftStat(parameterobject)
hwStat()
dbStat()
dcStat()
ftStat()
```

PARAMETERS

Name of Property	Type	Description
[attributeselector]	`Array of`	Names of additional attributes

RETURNS

Name of Property	Type	Description
error	`HW_API_Error`	Error information
object	`HW_API_Object`	statistic object

DESCRIPTION Gets specific statistic information from the different Hyperwave server modules.

EXAMPLES

```
out=hwStat();
```

INFO

```
info(parameterobject)
info()
```

PARAMETERS

Name of Property	Type	Description

[attributeselector]	Array of	Names of additional attributes

RETURNS

Name of Property	Type	Description
error	HW_API_Error	Error information
typeinfo	HW_API_Object	the typeinfo object
languages	Array of String	the languages that the server understands
customidx	Array of String	the attributes that are custom-indexed
systemidx	Array of String	the attributes that are indexed by the server

DESCRIPTION Gets specific information about the configuration of a Hyperwave server. This includes the indexed attributes, the languages that the server understands, and the type of the different attributes that are known to the server.

EXAMPLES

```
out=info();
writeln(out.languages) ;
// prints the languages that are known to the server via the license
```

2.5.8 KEYVALUE

CREATED BY `KeyValue` constructor:

`new KeyValue(key, value)`

`new KeyValue(key, values)`

PARAMETERS

key	String
value	String
values	Array of Strings

DESCRIPTION `KeyValue` holds one key value(s) pair of a `KeyValueField`. A `KeyValue` consists of a key and one or more values associated with this key.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `KeyValue` objects
key	The key of the `KeyValue`
value	A value of the `KeyValue`
values	An `Array` of values of the `KeyValue`

2.5.8.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

KEY

The key of the KeyValue

EXAMPLES

```
x = new KeyValue(“Name”,new Array(“foobar”,”foobar2”));
writeln(x.key);
// the result is Name
```

VALUE

This property reflects the first value (`String`) of possible more values.

EXAMPLES

```
x = new KeyValue(“Name”,new Array(“foobar”,”foobar2”));
writeln(x.value);
// the result is foobar
```

VALUES

This property reflects all values (`Array` of `String`s) of an KeyValue.

EXAMPLES

```
x = new KeyValue(“Name”,new Array(“foobar”,”foobar2”));
writeln(x.values);
// the result is foobar, foobar2
```

2.5.9 KEYVALUEFIELD

CREATED BY `KeyValueField` constructor:

```
new KeyValueField(attr1,attr2,attr3,...);
new KeyValueFiled(String)
```

PARAMETERS

attr1...attrn	KeyValues that construct the whole object.
`String`	`A string consisting of url-encoded list of keys and values (key1=value1&key2=value2...)`

The constructed object consists of these attributes (key value(s) pairs).

DESCRIPTION A `KeyValueField` is a field of `KeyValues`.

PROPERTY SUMMARY All inserted `KeyValue` objects are mirrored into properties of a `KeyValueField` and result in their corresponding value properties. This means that if an object has a `KeyValue` with a key of “Name” and a value of “foobar” then `KeyValueField` has a property of “Name” with its value (`String`) “foobar”. If a `KeyValue` has more than one value the first one is chosen. The properties can be assigned to have new values. If an array of strings is assigned to a property then all strings are values of the property.

Property	Description
prototype	Defines a property that is shared by all `KeyValueField` objects
length	The number of `KeyValue` objects in the `KeyValueField`
[“attribute name”]	The `KeyValue` in the `KeyValueField`

METHOD SUMMARY

Method	Description
keyvalue	Returns the `KeyValue` at a specific position
insert	Inserts a new `KeyValue` into the `KeyValueField`

2.5.9.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

LENGTH

The number of `KeyValue` objects in the `KeyValueField.`

[“ATTRIBUTE NAME”]

All `KeyValue` objects are mapped into properties of the `KeyValueField`. If an object has a `KeyValue` with key “foo” and value “bar” then the objects property “foo” is “bar”

```
x=new KeyValueField(new KeyValue(“foo”,”bar”)) ;
writeln(x.foo) ;
writeln(x[“foo”]) ;
```

2.5.9.2 METHODS

KEYVALUE

SYNTAX

```
keyvalue(number)
keyvalue(String)
```

PARAMETERS number index of the `KeyValue` which should be returned

`String` key of the `KeyValue` which should be retrieved

RETURNS `KeyValue`

DESCRIPTION Gets a specific `KeyValue` either by index or by key.

EXAMPLES none

INSERT

SYNTAX `insert(item)`

PARAMETERS item `KeyValue`

RETURNS nothing

DESCRIPTION Inserts a new `KeyValues` into the current `KeyValueField`.

EXAMPLES

```
x = new KeyValueField(new KeyValue("Name","foobar"),
KeyValue("Title","en:this is a test")) ;

x.insert(new KeyValue("test","foobar");
```

2.5.10 OPTIONPARSER

CREATED BY `OptionParser` constructor:

`new OptionParser(stringarray,object);`

PARAMETERS stringarray an `Array` of `String`s with the options to parse, typically the global argv object

object An object having the parameters as keys and integers as their corresponding values

DESCRIPTION The OptionParser is used by hwjs to parse the command line options in a comfortable way.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `OptionParser` objects

METHOD SUMMARY

Method	Description
more	Returns true if more options are available
option	Returns the next option
parameter	Returns the options parameter
nextIsOption	Returns true if the next argument is an option
error	Returns errorcode
ambiguous	Returns the ambiguities of a command

2.5.10.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

2.5.10.2 METHODS

MORE

SYNTAX `more()`

PARAMETERS none

RETURNS `Boolean`

DESCRIPTION Returns true if more parameters are available in the `OptionParser`

OPTION

SYNTAX `option()`

PARAMETERS none

RETURNS `Number`

DESCRIPTION Returns the current option as positive integer or negative integers if error occurred:

- 1 option is unknown
- 2 option is ambiguous
- 3 internal error
- 4 no arguments
- 5 no more options
- 6 internal error

PARAMETER

SYNTAX `parameter()`

PARAMETERS none

RETURNS `String`

DESCRIPTION Returns the option’s parameter

NEXTISOPTION

SYNTAX `nextIsOption()`

PARAMETERS none

RETURNS `Boolean`

DESCRIPTION Whether next argument is an option or not

ERROR

SYNTAX `option()`

PARAMETERS none

RETURNS `Number`

DESCRIPTION Returns the current error, see option() for description of error codes

EXAMPLE

```
x = new OptionParser(argv,{“-help“:1,“-verbose“:2,“-object“:3,“-
version“:4}) ;
while (x.more())
{
switch (x.option())
   {
case 1:
write („you like help\n“) ;
break;
case 2:
write („you like verbose\n“) ;
break;
case 3:
if (x.more() && !x.nextIsOption())
         {
write(„you like object with“,x.parameter(),“\n“) ;
         }
else
         {
write(„-object requires parameter\n“) ;
         }
break;
case 4:
```

```
write („you like version\n“) ;
break;
case -1 :
write („* unknown option\n“) ;
break ;
case -2 :
write („* ambiguous commandline option:“, x.ambiguous(),“\n“) ;
break ;
default:
write („internal error\n“) ;
break;
   }
}
```

2.5.11 HTMLJAVASCRIPTPARSER

CREATED BY `HTMLJavaScriptParser` constructor:

`new HTMLJavaScriptParser();`

DESCRIPTION The HTMLJavaScriptParser parses a given HTML block and calls user defined functions if it encounters a valid JavaScript construct. This parser is used if one wants to execute JavaScript in a HTML file which resides in the server or in the file system.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `HTMLJavaScriptParser`
htmlCB	Callback for pure HTML
javascriptCB	Callback for pure JavaScript
javascriptToAttributeCB	Callback for JavaScript in an attribute
javascriptToValueCB	Callback for JavaScript in a value
includeCB	Callback for an include
errorCB	Callback for errors

METHOD SUMMARY

Method	Description
parse	Parses an text block and calls the user defined functions

2.5.11.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

HTMLCB

The html callback is called when a block of pure HTML is identified. The called function gets one argument which is the block of HTML.

JAVASCRIPTCB

The javascript callback is called when a block of pure JavaScript is identified. A block of pure JavaScript is enclosed by <server> and </server> tags. The called function gets one argument which is the block of JavaScript.

JAVASCRIPTTOATTRIBUTECB

The javascriptToAttribute callback is called when a HTML tag attribute name consists of JavaScript (in backquotes). The called function gets one argument which is the block of JavaScript.

JAVASCRIPTTOVALUECB

The javascriptToValue callback is called when a HTML tag attribute value consists of JavaScript (in backquotes). The called function gets one argument which is the block of JavaScript.

INCLUDECB

The include callback is called when a <server src="path"> is identified. The called function gets one argument which is the path.

ERRORCB

The error callback is called when an error occurred during parsing of the given text block. The called function gets one argument which is the errorcode:

1 invalid </server> tag

2 invalid <server> attribute

3 invalid <server src> tag

4 </server> tag missing

5 closing quote missing

6 closing backquote missing

2.5.11.2 METHODS

PARSE

SYNTAX `parse(String)`

PARAMETERS `String` the text block containing JavaScript constructs

RETURNS nothing

DESCRIPTION Parser the given text block and calls the user defined functions for the different kinds of JavaScript constructs.

EXAMPLE

```
function printHTML(data)
{
write(data) ;
}

function printJS(data)
{
this.eval(data) ;
}

function printJSAttrib(data)
{
write(this.eval(data));
}

function printJSValue(data)
{
write("\"" + this. eval(data) + "\"");
}

function printInclude(data)
{
// here we could include some file
// but we skip this here
}

function printError(error)
{
// error
// write("Error: ", error, "\n");
}

file = new File("test.html");
file.open("r");
text = "";
while (!file.eof()) {
line = file.readln();
text = text + line + "\n";
}
```

```
parser = new HTMLJavaScriptParser();
parser.htmlCB = printHTML;
parser.javascriptCB = printJS;
parser.javascriptToAttributeCB = printJSAttrib;
parser.javascriptToValueCB = printJSValue;
parser.includeCB = printInclude;
parser.errorCB = printError;

parser.parse(text);
```

2.5.12 SENDMAIL

CREATED BY `SendMail` constructor:

`new SendMail()`

PARAMETERS none

DESCRIPTION This class is used to send mails to users via the internet.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `SendMail` objects
Subject	Subject of the mail
From	The sender of the mail
To	The addressee of the mail
Cc	Carbon copy
Bcc	Blind carbon copy
Errorsto	Errors to
Body	The mail itself
Smtpserver	smtpserver used for sending
[“property”]	all other properties are header fields of the mail

METHOD SUMMARY

Method	Description
send	Sends the mail

2.5.12.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

SMTPSERVER

Is used to define the smtp server which is used to send the mail. When not specified, “localhost” is assumed.

BODY

The body of the mail itself.

OTHER PROPERTIES

All other properties are mapped into the header fields of the mail.

2.5.12.2 METHODS

SEND

SYNTAX `send()`

PARAMETERS none

RETURNS `Boolean`

DESCRIPTION Sends the mail and returns true if mail was sent correctly.

EXAMPLE

```
mail = new SendMail();
mail.To = „bmarsch@hyperwave.com“;
mail.From = „tvollmer@hyperwave.com“;
mail.Subject = „Test“;
mail.Body = „Hallo ....“;
mail[“Content-Type“] = „text/plain“;
mail.send();
```

2.5.13 FILE

CREATED BY `File` constructor:

`new File(String)`

PARAMETERS `String` filename of the file to work with

DESCRIPTION This class is used to open, create, read and alter files on the filesystem.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `File` objects

METHOD SUMMARY

Method	Description
open	Open file
close	Close file
read	Read bytes from file
readln	Read a line from file
write	Write bytes to file
writeln	Write a line to file
flush	Flush file
getLength	Get the length of a file
getPosition	Get the position of file pointer
setPosition	Set the Position of file pointer
eof	Returns true if end of file
Exists	Returns true if file exists
isDirectory	Checks if file is a directory
isReadable	Checks if file is readable
isWriteable	Checks if file is writeable
copy(destination)	Copies file to destination
move(destination)	Moves file to destination
remove	Deletes file

2.5.13.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

2.5.13.2 METHODS

OPEN

SYNTAX `open(String)`

PARAMETERS `String` mode for opening

RETURNS	`Boolean`
DESCRIPTION	Opens file with a specific mode. Returns true if opening was successful.

Modes:

- r open for reading
- r+ open for reading and writing
- w open for writing
- w+ open for reading and writing, truncate file
- a open for writing, append to file
- a+ open for reading and writing, append to file
- b open in binary mode (Windows)

CLOSE

SYNTAX	`close()`
PARAMETERS	none
RETURNS	`Boolean`
DESCRIPTION	Closes a file and returns true if the closing was successful.

READ

SYNTAX	`read(Number)`	
PARAMETERS	`Number`	number of bytes to read in
RETURNS	`String`	
DESCRIPTION	Reads a number of bytes and returns them as string. If end of file is reached then the number of bytes of the string will be less than requested.	

READLN

SYNTAX	`readln()`
PARAMETERS	none
RETURNS	`String`
DESCRIPTION	Reads a line from file.

WRITE

SYNTAX	`write(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Writes a string to file. Returns true if successful.

WRITELN

SYNTAX	`writeln(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Writes a string with an appended newline to file. Returns true if successful.

FLUSH

SYNTAX	`flush()`
PARAMETERS	none

RETURNS	`Boolean`
DESCRIPTION	Flushes a file and returns true if successful.

GETLENGTH

SYNTAX	`getLength()`
PARAMETERS	none
RETURNS	`Number`
DESCRIPTION	Returns the length of a file.

GETPOSITION

SYNTAX	`getPosition()`
PARAMETERS	none
RETURNS	`Number`
DESCRIPTION	Returns the current file position.

SETPOSITION

SYNTAX	`setPosition(Number offset, Number where)`
	`setPosition(Number offset);`

PARAMETERS	offset offset for positioning where: (0 is default) 0 seek from begin position 1 seek from current position 2 seek from end position
RETURNS	`Boolean`
DESCRIPTION	Seeks to a certain position in the file and returns true if successful.

EOF

SYNTAX	`eof()`
PARAMETERS	none
RETURNS	`Boolean`
DESCRIPTION	Returns true is at end of the file.

EXISTS

SYNTAX	`exists()`
PARAMETERS	none
RETURNS	`Boolean`
DESCRIPTION	Returns true if the file exists.

ISDIRECTORY

SYNTAX	`isDirectory(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Returns true if the file is a directory.

ISREADABLE

SYNTAX	`isReadable(String)`

PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Returns true if the file is readable.

ISWRITEABLE

SYNTAX	`isWriteable(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Returns true if the file is writeable.

COPY

SYNTAX	`copy(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Copies the file to the destination. Returns true if successful.

MOVE

SYNTAX	`move(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Moves the file to the destination. Returns true if successful.

REMOVE

SYNTAX	`remove(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Deletes the file. Returns true if successful.

EXAMPLE

```
f = new File(„test“);
f.open(„w“);
f.writeln(„line1“);
f.close();
```

2.5.14 DIRECTORY

CREATED BY `Directory` constructor:

`Directory (String)`

PARAMETERS `String` string path of directory

DESCRIPTION This class is used to create and move directories, as well as to read directory entries.

PROPERTY SUMMARY

Property	Description
prototype	Defines a property that is shared by all `Directory` objects

METHOD SUMMARY

Method	Description
create	creates a directory with the name `path`
move(destination)	moves the directory `path` to the destination
remove	deletes the directory `path`

open	opens the directory to read individual entries
read	reads a directory
close	closes the directroy
error	delivers an error message

2.5.14.1 PROPERTIES

PROTOTYPE

Represents the prototype for this class.

2.5.14.2 METHODS

CREATE

SYNTAX	`create(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Creates a directory `path`.

MOVE

SYNTAX	`move(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Moves the directory `path` to the destination. Returns true if successful.

REMOVE

SYNTAX	`remove(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Deletes the directory `path`. Returns true if successful.

OPEN

SYNTAX	`open(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Opens the directory to read individual entries. Returns true if successful.

READ

SYNTAX	`read(String)`
PARAMETERS	`String`
RETURNS	`String`
DESCRIPTION	Reads in a directory entry.

CLOSE

SYNTAX	`close(String)`
PARAMETERS	`String`
RETURNS	`Boolean`
DESCRIPTION	Closes the directory. Returns true if successful

ERROR

SYNTAX	`error(String)`
PARAMETERS	`String`
RETURNS	`Number`

DESCRIPTION Displays an error code. Non-zero return means error.

EXAMPLE

```
function readdir(path)
{
var dir = new Directory(path);

if (!dir.open()) {
 writeln("could not open directory: ", dir.error());
 exit(1);
}

writeln("listing of directory ", dir.name, ":");

for (;;) {
 var name = dir.read();
 if (!name)
  break;
 var f = new File(dir.name+"/"+name);
 writeln((f.isDirectory()?"d":"-")+(f.isReadable()?"r":"-
")+(f.isWriteable()?"w":"-"), f.name);
}

dir.close();
}

readdir ("testdirectory") ;
```

3 THE HYPERWAVE GATEWAY INTERFACE (HGI)

HGI provides a powerful way to integrate external databases in Hyperwave Information Server for UNIX. If you are using the Common Gateway Interface (CGI) you have the problem that each request to a CGI gateway initiates the execution of a program and after processing the request the program terminates. Thus with CGI it is not possible to hold persistent connections to external sources like databases, which has the disadvantage that a new connection must be made for every request, which can be time-consuming. With HGI you are able to write gateway programs which allow persistent connections even with exploitation of Hyperwave's multi-user concept (user sessions). Further, HGI provides the transparent integration of external data sources into the Hyperwave collection hierarchy.

Note: *HGI is not available for Hyperwave Information Server for Windows NT.*

3.1 HGI SPECIFICATION

3.1.1 THE GENERAL REMOTE OBJECT

Hyperwave uses different types of remote objects to describe common attributes of objects not stored on the server (FTP, Telnet, WWW, Gopher, WAIS, SQL). Some of the objects have additional attributes to satisfy special needs (Gopher, WAIS, SQL). The attribute **Protocol** is used to specify the type of remote object and therefore the type of gateway that will handle requests.

For the Hyperwave Gateway Interface there exists a further type, the *general remote object*, where the gateway designer has the possibility to define some attributes in addition to the attributes which are common to all remote objects. These additional attributes are needed to store important information used by the gateway to perform its task. Suppose you have programmed a gateway to a relational database, in which case you may need an attribute to store the SQL statements which are used to query the database.

Now let us consider some important attributes common to all remote objects.

· **Protocol**

  A protocol type must be specified to allow a clear distinction between the types of general remote objects. Each protocol type corresponds to one particular HGI gateway program except for the protocol types which belong to 'normal' remote objects like Gopher, Telnet, FTP, etc. For example, if you have designed a gateway to an Informix database, the entry in the attribute Protocol could be: `Protocol = Informix`.

· **MimeType**

  If a general remote object has been selected, it is necessary that the client knows what data to expect. The attribute **MimeType** enables this. There are two cases for the client:

  · **MimeType** has an entry: data which correspond to the MIME type specified are expected.

  · **MimeType** does not exist or is empty: children, created on the fly, are expected. These children are general remote objects of the same type. This way dynamically created collection hierarchies can be built.

- **Host** and **Port**

  The first important purpose is to tell the gateway the host name and the port number of the external source. These two attributes are also important for automatic identification when connecting to external sources.

3.1.1.1 HOW TO INSERT A GENERAL REMOTE OBJECT

Because a general remote object can have an arbitrary number of arbitrary attributes, the only way to insert such an object is to use the Hyperwave command `hwinsdoc` (see *Hyperwave Administrator's Guide*). It is a good idea to store the object attributes in a file and reference the file with the parameter `-attr`.

Suppose you have a file named `example` which is stored in the directory `~test/tmp/`.

```
Type=Document
DocumentType=Remote
Protocol=test
Host=fiicmss01
Port=7654
Name=HGI_test
Attr1=x
Attr2=y
```

(Attr1 and Attr2 are additional parameters.)

To insert this object in a collection named test-collection on a Hyperwave Information Server named hyperg (hyperg.tu-graz.ac.at) you have to invoke `hginsdoc` with the following parameters:

```
hginsdoc -hghost hyperg -pname test-collection -attr
~test/tmp/example
```

***Note**: The object must have a symbolic name for later referencing.*

3.1.1.2 IDENTIFICATION WHEN CONNECTING TO EXTERNAL SOURCES

If you want retrieve data from an external data base, a username and a password are often necessary. There are several ways to provide them:

- The user name/password combination is stored directly in the gateway program. This is simple, but not very flexible.
- The user name/password combination is included in the object's data by adding the attributes Uid and Pwd. Its also simple, but everyone is able to read these attributes. This is only useful for public access, where the username, password combination is not secret.
- Both attributes or only attribute Pwd can be left empty. In this case, the user has to supply the necessary authentication information during the session, in order to obtain data from the remote data source. Web clients display a dialog box for the user to type in username and password or the password. This is not implemented yet.
- You can provide them with help of the Hyperwave identification scheme. If a certain user selects a general remote object, the server searches for a previously stored user name/password combination, using the Hyperwave identification and the contents of the attributes Host and Port. This enables different authorizations depending on the user's rights (not implemented yet).
- If you are using HTML forms to operate with an HGI gateway you can create an appropriate form to insert the username and the password.

3.1.1.3 DATAFLOW WEB CLIENT - HGI GATEWAY

Suppose you have a Hyperwave Information Server with an HGI gateway and general remote objects which require this gateway. In principal the data flow is as shown in Figure 1.

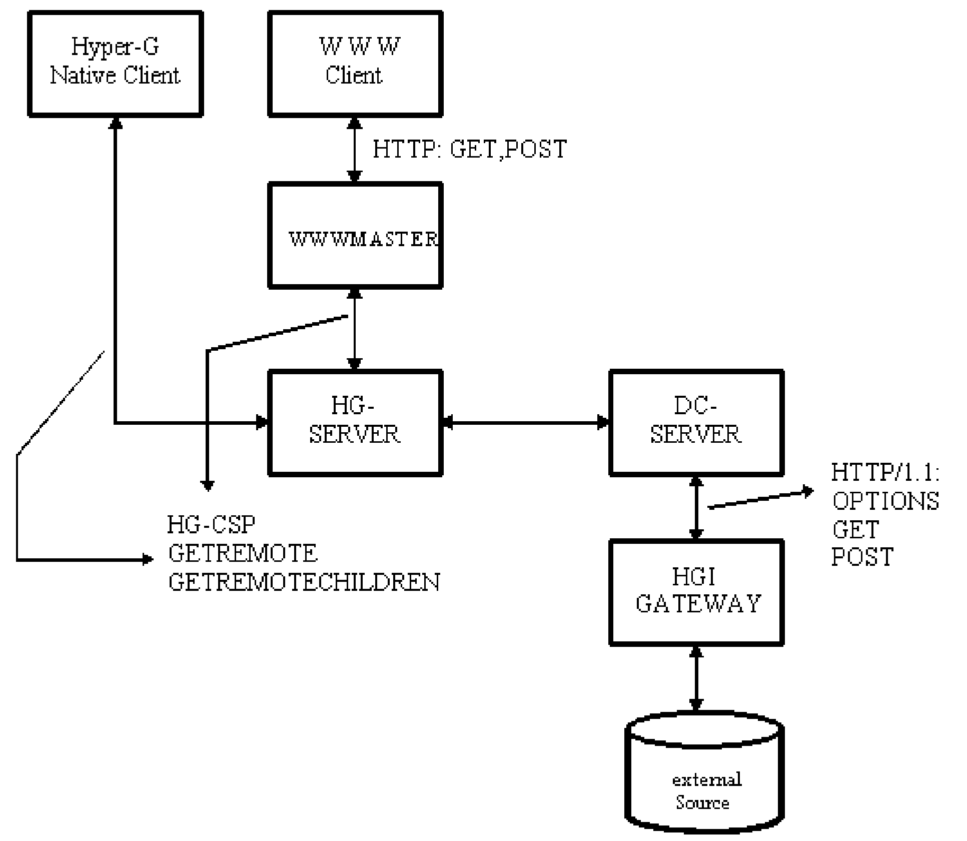

Figure 1: Data flow of the HGI Gateway

client

A user selects a general remote object. Depending on the MimeType attribute the client sends a request to hgserver. The client distinguishes between two cases:

- Attribute MimeType is empty or absent: The client sends the request GetRemoteChildren
- Attribute MimeType is not empty: The client sends the Request GetRemote

If it's a WWW client, the client sends a request (which refers to the appropriate object) to the WaveMaster of the server. The WaveMaster and not the client itself sends the request to the hgserver.

hgserver

The hgserver gets the request and sends it to the dcserver without interpreting the contents.

dcserver

The dcserver gets the request and, according to the Protocol attribute, sends it to the appropriate external gateway. For the data exchange between the dcserver and the gateway, the Hypertext Transfer Protocol Version 1.1 is used. (HTTP/1.1)

HGI gateway

The gateway parses the contents of the object and fetches the requested data from the external source.

Depending on the Request (GetRemote or GetRemoteChildren) appropriate data, or dynamically created children are sent back to the dcserver.

dcserver

Receives the data and send them to the hgserver

hgserver

Sends the data back to the client. In the case of a WWW client the data are sent back via the WaveMaster, which prepares the returned data for these clients.

client

Displays the result

3.1.1.4 DCSERVER AND GATEWAYS

The dcserver and the gateways operate in principle of the client/server model, where the dcserver works as client and the gateway as server. The data exchange between the client and the server takes place over a socket connection.

When the dcserver is started it performs an initialization process with the known HGI gateways, which it finds out about by reading a special configuration file. The dcserver is in addition able to start HGI gateways if required.

If the initialization task is successful, the dcserver registers this gateway in an internal list which contains all necessary information to operate with. The key for each gateway in this list is its protocol type. If the dcserver gets a request it looks at the entry of the **Protocol** attribute and compares the protocol type with the entries in its list to see if a gateway is available. If not, dcserver informs the client that the gateway is not available.

With this information the dcserver is ready to send requests to HGI gateways. The gateway parses the object's attributes, fetches the required data and sends the response back to dcserver.

3.1.1.5 GATEWAY TYPES

There are three types of gateways.

TYPE 1 - THE HGI GATEWAY DOES NOT FORK OFF CHILD PROCESSES

In this case the dcserver has a permanent connection to the HGI gateway. The gateway does not fork off child processes if it gets a request. The advantages are an easy implementation and if the connection between external gateway and external source is also permanent, no lost time to establish the connection. But it has the disadvantage that requests cannot be processed simultaneously. If the dcserver gets several requests within a short period it can come to a queue and this can lead to long response times. Thus this type is only useful if there is low demand.

TYPE 2 - THE HGI GATEWAY FORKS OFF A CHILD PROCESS FOR EACH REQUEST.

In this case, the gateway forks off a child process for each request, while the parent process listens for further connections. The child process is killed once the request has been processed. This makes it possible to handle several requests simultaneously. The disadvantage is that each request causes the establishing of a new connection to an external source. A typical example would be a WWW gateway.

TYPE 3 - THE HGI GATEWAY FORKS OFF CHILD PROCESSES WHICH ARE ABLE TO PROCESS SEVERAL REQUESTS.

The gateway is able to connect to more than one external source. The connections are established as required and remain open as long as the gateway designer has specified.

The gateway forks off children for each new required connection. The dcserver has a list of the open connections and so it is able to send the request to the right child directly.

To be able to uniquely associate an incoming request with a connection a unique identifier is needed. It is built with the contents of some attributes and the Session_key if required.

3.1.1.6 GATEWAY CONFIGURATION FILE

A configuration file called `hgi_config` is used to inform dcserver about the existing external gateways. It is found in directory `$DIRdcs/hgi`. It should include the host, the port, and the protocol type of each available HGI gateway. A further entry (Exec) which gives the name and path of the executable gateway program is necessary if you want dcserver to also execute the gateway. dcserver invokes a gateway program with /bin/sh -e. If the gateway is to be invoked by dcserver make sure that dcserver can execute the gateway program.

Note: *You must kill dcserver in order to make it read the* `hgi_config` *file after you have made changes.*

If you haven't made an `Exec` entry in `hgi_config`, dcserver assumes the gateway is running when it reads the entry in the `hgi_config` file. In this case you should start the gateway before you kill dcserver.

A further optional parameter `Mailto` can be specified if you want dcserver to send a mail when problems with the gateway arise.

An example configuration file

```
Host='Hostname'
Port='Portnumber'
Protocol='Protocol-Type'
Exec=/usr3/users/gateways/test.pl
Mailto=gate@iicm.tu-graz.ac.at

Host=localhost
Port='Portnumber'
Protocol='Protocol-Type'

Host=localhost
Port='Portnumber'
Protocol='Protocol-Type'
```

A line feed is used to separate the entries of the different gateways.

3.1.2 THE HTTP/1.1 REQUEST AND RESPONSE

```
CRLF = Carriage Return, Line Feed
SP = Space
```

An HTTP/1.1 Full Request

```
Full-Request =  Request-Line
   *(General-Header
   | Request-Header
   | Entity-Header)
   CRLF
   [Entity Body]
Request-Line =  Method SP Request-URI SP HTTP-Version CRLF
```

An HTTP/1.1 Full Response

```
Full-Request =  Status-Line
   *(General-Header
   | Request-Header
   | Entity-Header)
   CRLF
   [Entity Body]
Status-Line=    HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

The following sections describe only the required items.

3.1.3 THE INITIALIZATION TASK

The method used for the initialization task is OPTIONS. This method allows a client to determine the options and/or requirements associated with a resource, or the capabilities of a server without implying a resource action or initiating a resource retrieval. If the Request URI is an asterisk (*) the OPTIONS request is intended to apply to the server as a whole, this is exactly what is needed.

To perform the initialization task the dcserver has to establish a socket connection to the each gateway. The necessary information is found in the file `hgi_config`. If an `Exec` entry has been specified the dcserver has to execute the gateway first.

3.1.3.1 DCSERVER - HGI GATEWAY

The request line

```
OPTIONS * HTTP/1.1
```

This is the whole initialization request. It gets the gateway to inform the dcserver about its capabilities.

3.1.3.2 HGI GATEWAY - DCSERVER

If the request is successfully received, understood, and accepted the following response is returned from the gateway :

The Status-Line :

```
HTTP/1.1 200 OK
```

The response includes header fields which indicate optional features implemented by the gateway.

It has to inform the dcserver about its type in accordance with the types described in Section 6. This is done by an extension header of the Entity Header Fields. One of these three entries is understood by the dcserver.

Type: 1

Type: 2

Type: 3

For Type 1 and Type 2 the response is now complete.

For Type 3 additional information is necessary because the dcserver may have open connections to several child processes of an HGI gateway. (E.g. each child process manages a connection to an external data source). Thus the dcserver must know which child process is ready to process an appropriate request. This is performed by an additional list created by the dcserver, which registers the child processes of each HGI gateway of type 3. Thus a key for each child process must be available.

This key is built with the contents of some object attributes and the Session_Key (if required). To include the Session_Key is useful if there is heavy load and each user should have its own child processes to handle the requests. This additional information is sent in the Entity Body. The length and the type of the Entity Body is specified in the Entity Header with the fields Content-Length and Content-Type.

Content-Type: hgobject

The Content-Type used is not officially registered which is enabled by the HTTP protocol specification.

This type specifies that the data in the Entity Body have the structure of Hyperwave objects. A Hyperwave object consists of a number of name/value pairs, each pair in a separate line. No empty line is allowed inside a related object, because an empty line is used to mark a new object, if several objects are included in the Entity Body.

An example:

HTTP/1.1 200 OK	status line
Content-Length:21	entity header
Content-Type:hgobject	
Type:3	
CRLF	
Key=Host,Port,Session_Key	entity body

With this information the dcserver is now ready to perform real requests

3.1.4 REQUEST GETREMOTE AND GETREMOTECHILDREN

The content of the requests GetRemote and GetRemoteChildren is identical so they are treated together. If the dcserver gets a request it has to transmit this request to the right gateway. With its internal list, built during the initialization process, it will find the right one. The method used to transfer the request is POST, which is the suitable method to transfer a block of data to a data handling process (the gateway). The actual function performed by the POST method is determined by the server and is usually dependent on the Request-URI. In this case the Request-URI's GetRemote and GetRemoteChildren are used. A valid Request-Line:

POST GetRemote HTTP/1.1

POST GetRemoteChildren HTTP/1.1

The main content of a request are the object data of the selected object, the reference number RefNo of the request (a further extension-header of the Entity Header fields) and some additional information (in dependence of the gateway type).

The object data are transferred in the Entity-Body. The attributes Content-Length, Content-Type and RefNo are added by the dcserver. The media type used is hgobject.

Look at the following example :

POST SP GetRemote SP HTTP/1.1	request line
Content-Length:1243	entity header
Content-Type:hgobject	
RefNo:1	
CRLF	
ObjectID = 0x0000001b	entity body
Type = Document	
Document Type = Remote	
...	
Port = 1470	
Host = iicm	

This is the frame of each request, which is valid for all three gateway types. In dependence of the gateway type, there are some additional attributes, discussed in the following sections.

3.1.4.1 TYPE 1 AND TYPE 3

As described in Section 6, the dcserver has a persistent connection to the external gateway, or several persistent connections to external gateways. To continue this persistent connection, the dcserver and the gateway have to indicate this, on all requests and responses. This is done by Connection field of the General-Header. The request above gets an additional General-Header field, to indicate this.

Connection:Keep-Alive

3.1.4.2 TYPE 2

This gateway type has no persistent connections, so there is no additional field

3.1.4.3 ADDITIONAL DATA

After the object's data, separated by an empty line, there can follow data. Currently this option is used to transmit the data of fill out forms.

The form data are presented in the form of name/value pairs as described in the Common Gateway Interface - CGI description. The are also standard URL encoded and must be decoded before using.

3.1.5 RESPONSE GETREMOTE

After the gateway has successfully processed the request it transfers a response to the dcserver, which is similar to the response described in section 7.3. The differences are the content of the Entity-Body, which includes the from the external data source requested data, and some additional fields for special needs, like the reference number and caching.

The Content-Type field contains the MIME type of these data. An example :

```
HTTP/1.1 200 OK                                   status line
Content-Length:1035
Entity-Header
Content-Type:text/html
RefNo:1
CRLF
<TITLE>Query Response>/TITLE>                     entity body
<H1>Peter SMITH</H1>
.....
<HR>
```

To manage dcserver's caching mechanism, a further optional field can be used, the `Expires` field of the Entity-Header. Because the resource will be dynamic by nature, entities from that resources should be given an appropriate Expires value which reflects this dynamism.

Expires:"HTTP-date"

example: `Expires:Thu, 10.02.1996 16:00:00 GMT`

If the `Expires` field is absent, or its value is zero, or an invalid format is detected, it is treated as "expires immediately". This means no caching.

7.5.1 Type 1

The Keep-Alive keyword must be sent on all requests and responses of a persistent connection. So the General-Header field Connection must be added.

Connection:Keep-Alive

7.5.2 Type 3

A gateway of the third type is able to hold many connections to external data sources. It is very useful to have a mechanism to set some parameters of such connections (e.g. time-outs). This is done by the General-Header field, Keep-Alive. The gateway can set a time-out, a max, and for special reasons some user defined additional parameters. The time-out parameter allows the gateway to indicate the amount of time in seconds it is currently allowing between the response was generated and when the next request is received from the client. (i.e. a request time-out). The max parameter allows the server to indicate the maximum additional requests that it will allow on the current persistent connections. At the current state no further parameters are supported.

Now the gateway has some possibilities when adding the fields Connection and Keep-Alive (optional).

For example, no Keep-Alive field when there should be no time-out. An example with time-out and a maximum number of additional requests.

Connection:Keep-Alive

Keep-Alive: timeout=90, max=10

You may use the Keep-Alive field only if the "Keep-Alive" keyword is used as parameter of the Connection field. If the gateway wants to interrupt the connection after sending the response, it must only send the response without the Connection field.

3.1.6 RESPONSE GETREMOTECHILDREN

The response message of the request GetRemoteChildren is in principal equal to the response message of GetRemote. The only difference is the content of the Entity-Body which includes the object data of the dynamically created children. Each child object has clearly to satisfy the specification of the general remote object which the gateway designer has defined. The objects are separated by a CRLF.

An example

HTTP/1.1 200 OK	status line
Content-Length:3735	entity header
Content-Type:hgobject	
RefNo:1	
CRLF	
Objectdata of children (1)	entity body
CRLF	
Objectdata of children (2)	
...	
Objectdata of children (n-1)	
CRLF	
Objectdata of children (n)	

3.1.7 AN ERROR OCCURRED

If an error occurs the Status-Line corresponds clearly to the definition in 7.2.2, but with a Status-Code, and a Reason Phrase which indicates an error. All error codes listed in the Internet draft are on principal allowed, but the dcserver does not understand the meaning of all error codes, but he understands at least the meaning of the calls of each status code. For special reasons also extensions to the HTTP error codes are allowed. Here is an example of an Error Response .

HTTP/1.1 500 Internal Server Error

3.1.8 PERL5 HGI GATEWAYS

3.1.8.1 REQUIREMENTS

To run perl5 HGI gateways you need:

perl5	perl 5.001 patchlevel m or higher
HGI_handler.pm	Version 0.7 or higher
dcserver	Version 1.46 or higher
WaveMaster	Version 2.99 or higher

You can find the file `HGI_handler.pm` in the directory `$DIRdcs/hgi/gateways`.

Note*: If you want to run HGI gateways in other locations than* `$DIRdcs/hgi/gateways` *of the local Hyperwave Information Server you have to copy the file* `HGI_handler.pm` (`$DIRdcs/hgi/gateways/`) *to the perl5 lib directory or to the directory of your gateway program.*

3.1.8.2 THE HGI_HANDLER

The HGI_handler is a perl module which provides the basic functionality of a HGI gateway. It consists of a number of subprograms.

start_HGI

This is the key subprogram of the HGI_handler. It must be invoked at the beginning of each HGI gateway with a number of parameters which specify the characteristics of the gateway as described in the HGI specification.

Parameter style:

```
start_HGI(-prot=>'protocol_name',
     -type=>3,
     -port=>8015',
     -attr=>['attr1','attr2','attr3','attr4*'],
     -key=>['attr1','attr5','Session_Key'],
     -timeout=>1200
     -log_file=>'/dev/tty/',
     -debug_level=>1);
```

-prot	required	Specifies the protocol type the gateway is used for
-type	required	Specifies the type of the gateway in accordance with the types described in Gateway Types.
-port	required	Specifies the port number of the gateway. It must correspond with the port number in the file `hgi_config`, which initiates the dcserver to connect to the gateway.
-attr	optional	includes the object’s attribute names which values are needed later in the gateway program. If more than one instance of an attribute are allowed this must be specified with a ’*’ after the attribute name attr4*. If the -attr parameter is omitted no value of any attribute is available in the gateway program.
-key	required type3, with others no effect	The key which enables the dcserver to identify the right child process if a gateway of type 3 is used, is built with its content. The term Session_Key signals the dcserver that for each user session the requests are performed by at least one child process (depends on the other attributes of the key).
-timeout	optional type3, with	specifies the time in seconds a child process waits before it terminates if no requests are made. The default value is 1200.

	others no effect	
-log_file	optional	necessary if you do not want to use the default log file. The default log file resides in the $DIRdcs directory and is named protocol_name.log. Old log files are stored in the same way as other Hyperwave log files. (protocol_name.log.hexnumber - hexnumber number of non-leap seconds since January 1, 1970, UTC)
-debug_level	optional	tells the gateway what information the log file should include. Two debug levels are supported (1,2). Level 1 represents normal debug information and level 2 stands for detailed debug information.

***Note**: The order of the parameters in the parameter list is not important.*

log & log_con

These two subprograms enable the output of log messages in the log file. The difference between log and log_con is that log_con writes the number of the user-session in the log-file entry.

Parameter style:

```
log('log_message');
log_con('log_message');
```

debug & debug_con

These subprograms enable the output of debug information in the log file, if a debug level is specified. The difference between debug and debug_con is the same as above. An additional parameter specifies the debug level of the debug message.

Parameter style:

```
debug(1,'debug_message');
debug_con(1,'debug_message');
```

Exported variables

Some variables are exported from the HGI_handler to make them accessible outside the module.

$object

As described above the parameter -attr makes it possible to specify some attributes to use their values in your gateway code. The variable $object is a reference to a hash-table which includes the appropriate values. The values can be referenced as follows:

One instance	$object->{'attr_name'}
One or more instances (*)	$object->{'attr_name'}[nr_of_instance]

$HGI_err

The HGI_handler supports three error classes:

class 0	no error
class 1	no fatal error - do not kill child process
class 2	fatal error - kill child process

The section 'Implement a gateway using the HGI_handler' shows how to use this variable.

$HGI_errstr

If an error occurs this variable should include the reason.

The section ’Implement a gateway using the HGI_handler’ shows how to use this variable.

Forms and HGI

If you invoke your HGI gateway via an HTML form you probably want to use the data entered in the form. To make it easy to switch from CGI to HGI the way to get the form data is exactly the same.

The difference between CGI and HGI is how the output of the gateway is sent back to the client. In contrast to CGI where the output is sent to STDOUT you have to assign the output to a variable and send the content of the variable as described in the next section.

See http://www.hyperwave.com/hg_cgi for further information.

Implementing a gateway using the HGI_handler

The HGI_handler makes it very easy to implement a HGI gateway. Here is the body of all perl HGI gateways using the HGI_handler:

```
#!/usr/local/bin/perl -w
use HGI_handler;
start_HGI(-prot=>’protocol_name’,
       -type=>3,
       -port=>8015’,
       -attr=>[’attr1’,’attr2’,’attr3’,’attr4*’],
       -key=>[’attr1’,’attr5’,’Session_Key’],
       -timeout=>1200
       -log_file=>’/dev/tty/’,
       -debug_level=>1);
sub getremote {
}
sub getremotechildren {
}
sub disconnect {
}
```

The perl script must contain three subroutines:

sub getremote & sub getremotechildren

Remember the HGI specification, two request types are possible:

GetRemote	data are expected
GetRemoteChildren	child objects are expected

One of these subprograms is invoked if the gateway gets an request. You can write any perl code in these subprograms you want, but it is very important that the subprograms return the right data. Two cases must be distinguished:

no error occurred

GetRemote	the subprogram must return the MimeType of the data and the data itself -> the data must correspond to the MimeType. `return($MimeType,$data)`
GetRemoteChildren	the subprogram must return child objects corresponding to the MimeType hgobject `return(’hgobject’,$data)`

an error occurred

The subprogram must return 0 and it must set the variables $HGI_err and $HGI_errstr. $HGI_errstr must contain 1 or 2 and the variable $HGI_errstr should contain the reason for the error.

sub disconnect

If you have implemented a gateway which connects to an external source in some cases it is important that the gateway disconnects from these sources in a correct way. The subprogram disconnect enables this.

***Note**: If one of these subprograms is not needed by your gateway write no code in the subprogram.*

3.1.9 INSERTING A NEW GATEWAY

1. Create a new entry in the file `hgi_config`.
2. Kill the dcserver to request it for reading the changed version of the hgi-config file. If you do not know why follow the link above.
3. Insert the appropriate general remote object (see page 72). (Example: simple test form)

```
Type=Document
DocumentType=Remote
Protocol=tryit
Title=en:A simple test form
Name=simple_test
MimeType=text/html
```

Now you are able to test the gateway.

3.1.10 THE LOG_FILE

The log_file gives you detailed information about what goes on when the gateway executes. If this information is not enough set a higher debug_level (possible values: 0,1,2):

- Change the parameter `-debug_level` of the start_HGI procedure of your gateway program -> (kill and restart your HGI gateway)
- In case of the SQL gateway, kill the SQL gateway and restart it with the additional parameter `-debug` *debuglevel*.

***Note:** If you change* `-prot`, `-port`, `-type` *or* `-key` *you have to restart the dcserver and change* `hgi_config`*!*

3.2 SQL GATEWAY

To run the HGI SQL gateway you have to fulfill the following requirements:

· Requirements to run HGI gateways (see page 81).

- The perl5 DBI/DBD module for Oracle databases
- Install the HGI SQL gateway (copy the file sql.hgi - default location: $DIRdcs/hgi/gateways/) on the same machine as you have installed the DBI/DBD module and look at the section Gateway Insertion and Debugging

Example 1

You have installed the perl5 module DBI/DBD on the same machine as your Hyperwave Information Server. (Host = hyperhost) Perform the following steps:

- Think about the port number you want to connect your HGI-gateway (e.g. 8100)
- Make an entry in the hgi_config file.

```
Host=localhost
Port=8100
Protocol=sql
```

- Start the SQL gateway in the directory `$DIRdcs/hgi/gateways/` (at least with the parameter `-port`).

  Type sql.hgi -h to get further information

```
sql.hgi -port 8100 &
```

- Kill the dcserver
- Gateway is ready !!!

Example 2

You have installed the perl5 module DBI/DBD on a machine other than the one your Hyperwave Information Server is on. (Hyperwave is on hyperhost; module DBI/DBD is on dbhost) Perform the following steps:

- Think about the port number you want to connect your HGI gateway (e.g. 8100)
- Make an entry in the `hgi_config` file.

```
Host=dbhost
Port=8100
Protocol=sql
```

- Copy the files `HGI_handler.pm` and `sql.hgi` (`$DIRdcs/hgi/gateways`) to any directory you want of the dbhost.
- Start the SQL gateway from this directory

```
sql.hgi -port 8100 &
```

- Kill the dcserver
- Gateway is ready

***Note:** If you use the attribute* `Exec` *in the file* `hgi_config` *there is no need to start the SQL gateway because the dcserver does it for you.*

If there are any problems take a look at your log file

3.2.1 SQL GATEWAY VIA THE HYPERWAVE GATEWAY INTERFACE HGI

This chapter describes the SQL Gateway, implemented in accordance with the Hyperwave Gateway Interface - HGI specification. With this gateway it is possible to represent parts of a relational database in the sense of the Hyperwave navigational paradigm and format database data as HTML or HTF documents.

General considerations

Design requirements:

- Provide a program-free interface. If you want to offer data from a database via the Common Gateway Interface - CGI, you have to write a CGI program, which connects to the corresponding database, waits for the results and finally formats the result and outputs it. With the SQL Gateway it should be possible to connect to a number of different databases, whereby you need only know the host name, the port number, what RDBMS product (Oracle, Informix, Sybase, ...) and in some cases one or two additional parameters. Unfortunately, it is necessary to be a little bit familiar with the database language SQL to formulate the database queries, but the formulation of SELECT statements, used to query the database, is not too difficult.
- It should not be necessary that each request to the SQL gateway initiates a new connection to the database. The connections to the databases should stay open for a certain amount of time, waiting for new requests. This significantly speeds up the SQL Gateway, since the establishing of a connection could be time-consuming. In contrast to this on each request to a CGI-gateway a connection needs to be established.
- For the user it should not be visible that the data source is a database. Thus a user need not learn new features.
- For the information provider it should be easy to present the data stored in a database in a clear and easy manner.
  - Easy creation of collection hierarchies which represent parts of a database.
  - Show data, stored in the database, as part of HTML or HTF documents with the possibility to design HTML or HTF pages of his own, where the data is inserted with help of placeholders.

Because of the design requirements a gateway of type 3 must be used which is able to hold several open connections at the same time.

The SQL gateway performs the following functions.

- Parse the request received from the dcserver, to get:
  - which database the data is stored;
  - the SELECT statement to query the database and
  - some additional parameters how to format the result
- Look if there exists an open connection to the requested database and if there is not, connect to the database
- Query the database
- Format the result and send it to the dcserver

Note: *The SQL gateway can be optionally installed with every Hyperwave Information Server either at installation time or added on to an existing server.*

3.2.2 THE GENERAL REMOTE OBJECT FOR THE SQL GATEWAY

As described in the HGI specification the first step when designing a General Remote object for an HGI gateway is to specify a Protocol-Type. For the SQL gateway the Protocol-Type is named 'SQL'.

Protocol=SQL

The following additional attributes are necessary to fulfill the design requirements described above:

· Database: To specify the RDBMS product and eventually additional parameters

· SQLStmt: To define the SELECT statements to query the database

· Uid,Pwd: To specify the username and password to connect to the database

Now let us consider the following Hyperwave General Remote object with Protocol=SQL

```
ObjectID = 0x0000001b
Type = Document
Document Type = Remote
Author = hgsystem
Time Created = 95/03/09 11:35:10
Time Modified = 95/04/28 09:10:11
Title = en:List of Employees and their Salary
Protocol = SQL
Uid = scott
Pwd = tiger
Host = fstghp70.tu-graz.ac.at
Port = 1523
Database = oracle:t:edvz
SQLStmt = select ename,sal from emp
MimeType = text/html
```

The next section will focus on the object attributes that are important for this object type. In the following the General Remote object with Protocol=SQL is shortly named SQL object.

Host and **Port**

The attributes Host and Port contain the hostname and the portnumber of the database server. For example, fstghp20.tu-graz.ac.at stands for the hostname. The author can also write the IP address instead of the canonical hostname or the DNS alias. The Port attribute denotes the port number of the SQL listener demon.

Database

The SQL gateway can connect to different products of relational databases. For commercial SQL servers like Sybase, Oracle or Informix it is often necessary to use a vendor specific protocol. The syntax for this attribute is database:type:databasename. For example, this attribute can have the value oracle:t:edvz. The first word oracle specifies that the remote database is an Oracle database. The second word, in this case the letter t, is specific to the type of Oracle installation. The t stands for transparent network substrate. This information can be obtained from the system administrator or the database administrator. Finally, the third component edvz stands for the name of the database.

Uid and **Pwd**

The attributes Uid and Pwd are required for the authentication process of the SQL server. Uid stands for the username and Pwd for the password.

The author of the object can fill in the values for the attributes Uid and Pwd. These pre-set values are used for connecting to the SQL server. This is useful when the author wishes to provide anonymous (or public) access. Of course, public access must be used with caution. The author can additionally set the access rights to the object and thereby adjust the access rights to the database.

Both attributes or only Pwd attribute can be left empty. In this case, the user would have to supply the necessary authentication information during the session, in order to obtain data from the remote database. Web clients display a dialogue box for the user to type in username and password or the password. (not implemented yet)

The last possibility is to provide them with help of the Hyperwave identification scheme. A Hyperwave user can work as anonymous or as identified user. So, if a certain user selects a General Remote Object, the Hyperwave Information Server searches for a previously stored username password combination, using the Hyperwave identification and the contents of the attributes Host and Port. This enables different authorizations depending on the user rights. (not implemented yet)

MimeType

The value of this attribute defines what kind of data the client has to expect if an SQL object is selected.

If this attribute is empty or absent the client expects child objects created on the fly, thus an SQL object containing several instances of the attribute SQLStmt must not contain any value in the attribute MimeType or must be absent, otherwise an error occurs. In this case the MimeType attribute is added by the SQL-gateway automatically, see 3.5. Thus a filled out MimeType attribute initiates the client to expect data and is only a allowed when the SQL-object contains one SQLStmt attribute. It specifies how the result of the query is to be displayed, whether as HTML text, HTF text or whatever, the possible values conform to the MIME standard. For further information see the HGI specification.

SQLStmt

The attribute SQLStmt contains the actual SQL statements that are to be executed at the SQL server in order to retrieve data. The author can specify any number of SQL statements. The purpose of supplying several SQL statements is rather specific to Hyperwave, particularly to the structural organization of information to enable a user to navigate through a database using the Hyperwave navigational paradigm. The author can determine how the data are organized and presented to the user.

Several instances of attribute SQLStmt

An author can insert several instances of the attribute SQLStmt to perform dynamically created collection hierarchies. Consider the following instances:

```
SQLSTmt =1=     select table1.col1, table1.col2, from table1

SQLStmt =2=     select table2.col1, table2.col2 from table2 where
table2.col4=!table1.col1

SQLStmt =3=     select table2.col1, table2.col2, table2.col3, table2.col4,
table2.col5 from table2  where table2.col4=!table1.col1 and
table2.col1=!table2.col1
```

Each instance of this attribute represents one level when building the collection hierarchy. As you can see, each entry begins with a number followed by an equal sign. The number represents the level that the SELECT statement belongs to. After the equal sign the real SELECT statement begins. The specification of the level is required, except there is only one level, then the specification is optional.

When you are authoring a SQL object which operates with more than one level take into account that you have to use parts of the information retrieved by the SELECT statement of the previous levels to restrict the scope of the SELECT statement of the following level. This is done by a WHERE clause, comparison operators and placeholders. A placeholder consists of the name of a column that has been received in one of the previous levels with a '!' in front.

When such an SQL object is selected for the first time , the first statement is executed and a 'virtual' collection with dynamically generated child objects is created, where each child object corresponds to a single row of the query result. These child objects are also SQL objects that are created on the fly by the SQL gateway. Each child object contains the following entries in the SQLStmt attributes:

```
SQLSTMT  =1=          select  table2.col1,  table2.col2  from  table2  where
table2.col4=<<value Table1.col1 of row N>>
```

```
SQLSTMT =2=       select table2.col1, table2.col2, table2.col3, table2.col4,
table2.col5 from table2 where table2.col4=<<value Table1.col1 of row N>> and
table2.col1=!table2.col1
```

Note that the placeholder !TABLE1.COL1 has been substituted by the value of the corresponding column from the previous SELECT statement of each row of the query result.

Selecting a child created on the fly leads to a further 'virtual' collection with dynamically created children with the following entry in the SQLStmt attribute:

```
SQLSTMT =1=     select table2.col1, table2.col2, table2.col3, table2.col4,
table2.col5 from table2 where table2.col4=<<value Table1.col1 of row N>> and

table2.col1=<<value table2.col1 of row M>>

MimeType = text/html
```

If only one instance of the SQLStmt attribute is left, the attribute MimeType is added by the SQL-gateway automatically, to show the client that it has to expect data and to show it the type of data which will be retrieved by this SELECT statement. The default value of the MimeType attribute if text data will be retrieved is text/html.

An author can specify commands, separated from the SELECT statement by a semicolon, to hand over additional information to the gateway, which is in fact not in accordance with the SQL standard but very useful for this purpose. An author has the following supplementary possibilities.

SHOW

When dynamically generated child objects are created, a title for each object is needed. At default the value of the first column is used. With the command SHOW the author has the possibility to use values of any column, fetched with the SELECT statement. Thus the author can specify which column values he wants to use as children's object title. The syntax of the command SHOW is the BNF format

SHOW(1#ColumnNr)

Example: Treat the SQLStmt of the first level. If the author wants to use the values of both columns as title, then the attribute SQLStmt must look like the following.

```
SQLSTMT =1=     select table1.col1, table1.col2, from table1;SHOW(1,2)
```

One instance of attribute SQLStmt

This case arises if an object with several instances of the attribute SQLStmt is in process and a level is reached where only one attribute SQLStmt is left or if an author inserts a SQL-object which contains only one attribute SQLStmt. Note the attribute MimeType must be filled out.

```
SQLStmt =       SELECT COL1, COL2, COL3 FROM TABLE1
```

Only text data are retrieved

The SELECT statement specified delivers only text data, thus the author has the following possibilities to represent the output.

Default

The SQL gateway is able to display the received data in a default format whether as HTML or as HTF page. The value of the MimeType attribute is responsible for the format being used. If a SQL object with several instances of SQLStmt attributes is in process the attribute is added automatically and is set to text/html. If you would like to change this to text/htf you have to change the default values of your SQL - gateway.

MimeType=text/html

The SQL - gateway receives one row:

The SQL - gateway receives several rows:

MimeType = text/htf

The SQL gateway receives one row: The result is equivalent to the result above. (MimeType = text/html).

The SQL gateway receives several rows: It must be remarked that in HTF no tags exist to build a table.

Defined by the author

An author has the possibility to design a HTML or a HTF page to represent the data, received from the database, in accordance with his own wishes. Only one row must be received by the SQL - gateway to use an author-defined format. To insert column values in the HTML or HTF page you have to use placeholders. A placeholder consists of one of the column names of the SELECT statement with a ’!’ in front. Thus an author hs to prepare a HTML or HTF page with placeholders and store this page (file extensions ’html’ or ’htf’ must be used) in the following directory. There exists an environment variable named $SQL_FORMAT which includes the path to store user-defined pages. This is the entry point for the SQL gateway if such pages are required. To tell the gateway that you want to use your own format you have to add a command behind the SELECT statement, named FORMAT, separated by a semi colon - FORMAT(’relative Path/file’). The relative path begins in the path specified in $SQL_FORMAT.

Assume you do not want to display the data of the example above in a default format. Thus you have to prepare an HTML page, in this example named ’test.html’, and store it for example directly in the base directory where user defined pages reside. Then the SELECT statement of the last level would look like the following.

```
SQLStmt =3=     select table2.col2, table2.col3,
                table2.col4, table2.col5 from table2
                where table2.col4=!table1.col1
                and table2.col1=!table2.col1;FORMAT(’TEST.HTML’)
```

Example to show how column values are inserted:

```
<TITLE>A simple Test</TITLE>
<H1>A simple test</H1>
The value of the first column: !TABLE2.COL2
etc. ...
```

When authoring a page you must be conscious of the fact that most users use Netscape when browsing Hyperwave Information Servers. Thus there arise some things which the author must take into account.

Inserting links in your HTML page:

Links to WWW servers do not cause any problems, but links to Hyperwave Information Servers can lead to the following problem. When a user browses a Hyperwave Information Server via a WWW Browser, the Hyperwave WWW gateway - WaveMaster - forks off a child process to satisfy the request. It stores session specific information (a Cookie or a Session Key) to recognize if the same user takes a further request. So the WaveMaster is able to send the request to the process forked with the first request. Now the problem is that in most cases the WaveMaster cannot recognize (clearly only here) whether the user has a WaveMaster process and forks off a child even when there exists such a process, thus a user can invoke several WaveMaster processes which can lead to performance problems.

Inserting Header information in your HTML page:

The WaveMaster ignores the header information of your HTML page, so it is not possible to include any header information.

4 EXTERNAL IDENTIFICATION

Hyperwave provides an interface for incorporating external identification mechanisms. For example, one would probably not want to manually insert a user object for every user stored in an X.500 database. Rather one would implement a simple gateway which the Hyperwave Information Server contacts in order to ask for identification parameters which are then applied to the server. This gateway would, for example, in the case of a positive answer, return the user name, group membership information, account information, and other information to be specified in the future.

To make use of this identification method, there must either be objects present on the server which have the appropriate user-specific rights, or, in the case that the identification gateway supplies group membership information for a user, objects that have group-specific rights. For example, if the gateway returns a positive reply with the user name U1, the client who identified that way is permitted to access all objects which have appropriate rights for the user U1. As another example, if the gateway returns a user U2 where no objects exist in the database with any rights for U2, but supplies group membership information G1 and G2, and there are objects with appropriate group rights, the client may access these objects accordingly.

This identification relates only loosely to the other identification methods "automatic" and "manual" in that it is completely transparent to the client, and is applied only if manual identification is used *and* there are identification gateways configured. If one client types her/his user name/password combination, the server asks its configured gateways to map these things onto some other things (and group membership, if any), and then identifies over an internal mechanism which is explained in detail below.

4.1 CONFIGURING IDENTIFICATION GATEWAYS

First you have to tell the Hyperwave Information Server where to look for the information. All it needs to accomplish that task is an identifier for each external identification mechanism (for example, "yp" for "Yellow Pages"), and a target where to connect to (TCP and Unix domain stream sockets are currently implemented). The list of identification gateways may be arbitrarily long, and the server stops running through the list after the first successful request.

There is one reserved identifier, `local`, to interleave the list of external gateways with an identification request to the local Hyperwave database.

If none of the entries in the list of configured identification gateways is successful in its attempt to identify, the local user database in `dbserver` is always tried unless it is explicitly configured in the list of gateways using the `local` identifier and thus has already been checked.

Configuration is done via the `.db.contr.rc` file in the home directory of the user that runs the server. For each gateway add a line of the form

`HGSERVER::AUTHENTICATION[`<*identifier*>`]=`<*target*>

where the <*identifier*> is the identifier mentioned above, a string containing no spaces, and <*target*> is one of the following:

`TCP` <*host*> <*port*>

with the meaning that you may expect these to have (<*port*> being a decimal integer), or

`TCP` <*port*>

with the same meaning, the host is localhost implicitly, or

`UNIX` <*filename*>

to connect over a Unix domain socket.

The gateways are visited in the order their entries are listed in the file. The reason the scope of the line is `HGSERVER` is simply because that task is done by server module `hgserver`, which has the predefined process tag `HGSERVER`.

4.2 STARTING THE GATEWAYS ALONG WITH THE HYPERWAVE INFORMATION SERVER

It is possible to use `dbserver.control` to control arbitrary processes, including your identification gateways. Suppose you have written a gateway called `my.own.gateway`, which is invoked with the option `-p` and a port argument, and which is responsible for supplying identification from the "my" domain on the TCP port 6666. You may have `dbserver.control` start it along with the server by adding the lines

```
MAIN::PROCESS=MYPROCESS
MYPROCESS::COMMAND=my.own.gateway -p 6666
```

to the server configuration file `.db.contr.rc`.

Of course, you still have to configure the server to talk to it by also adding the line

```
HGSERVER::AUTHENTICATION[my]=TCP 6666
```

The identifier, in this case “my”, can be almost any string (not containing spaces) you want it to be, but it should be unique in order to distinguish the gateways from one another.

4.3 THE PROTOCOL

The basic procedure from the point of view of the Hyperwave Information Server is:

1. Connect using the specified mechanism (TCP or Unix domain socket)
2. Issue a request in some format
3. Wait for the reply which will come in some similar format
4. Read the reply
5. Disconnect
6. Identify the user if the reply was positive (else go to the next thing, if any)

The corresponding gateway:

1. Listens at the specified service
2. Accepts a connection
3. Reads the request
4. Thinks
5. Sends a reply

Note that the gateway must not rely on the fact that the Hyperwave Information Server disconnects after receipt of a reply. It must be able to deal with persistent connections. Currently the reason for `hgserver` to close has mainly to do with performance. The hgserver spawns a process for each client, so every client's process has to establish a connection to every gateway on an identification request. If these connections stayed open after the request was done, the system would soon run out of file descriptors. However, efforts are being made to re-implement (parts of) hgserver as a single multithreaded process. There it makes sense to share one connection to a gateway between all threads.

The request and the reply are in the Hyperwave object format, and have designated fields. This format is simply a name-value pair on each line, with the name and the value separated with a "=" character. Lines are separated by a linefeed character.

Both request and reply have a protocol version field. The current version is 1.0, so the line that has to be present in an object is "PVer=1.0".

4.3.1 THE REQUEST FORMAT

In addition to the protocol version, a request must have the following fields:

- **PVer=1.0**. The (currently constant) protocol version.
- **RefNo**=<*integer*>. This line has to be echoed by the gateway in order for the server to be able to distinguish between multiple replies when multiple requests are outstanding. This feature is not yet implemented by the server, but, as stated above, this is just a matter of time.
- **Req**=<*request*>. The only request currently used is "identify", so **Req=identify** is the only meaningful line up till now.
- **UName**=<*username*>. <*username*> is the name as supplied by the client.
- **Passwd**=<*password*>. The (unencrypted) password as supplied by the client.

4.3.2 THE REPLY FORMAT

In addition to the obligatory protocol version, a reply has the following fields:

- **PVer=1.0**. The (currently constant) protocol version.
- **RefNo**=<*integer*>. See the request format description.
- **Ack**=<*yes/no*>. If yes, the following fields are valid, and the client should be considered identified. If no, the identification was not successful.
- **UName**=<*username*>. Here <*username*> is the name under which the client should be identified on the server. Ideally, in order to not confuse the client, this would be the same as the name the client typed, but this is not a restriction. A gateway may perform any kind of mapping.
- **Group**=<*groupname*>. This line may occur zero or more times in the reply object. It denotes group membership.

4.4 DETAILS OF OPERATION

Basically, at least regarding identification, the Hyperwave Information Server actually consists of two servers which talk to each other. The first one, `hgserver`, spawns a process for each client. It does not maintain any persistent information. The second one, `dbserver`, maintains every kind of object, including the local Hyperwave user objects. Regular identification is done by that module, and the user information after identification is also maintained within that single process. So, access rights checking is centralized into one common place, thus making integrity and all these things simpler. An `hgserver` process only sends the client information it has (user name/password, and some additional information which is not essential for identification tasks) to `dbserver`. In the case of normal (local) identification `dbserver` checks if there is a corresponding user object, and, if so, the user is identified.

4.5 EXISTING GATEWAYS

There is an existing YP gateway, written by Michael Klemme, available on the Hyperwave CD. The file is called `YP.tgz` and is found in `/UNIX/`*<your platform>*`/EXTRAS/YP`.

5 APPENDIX A

5.1 HYPERWAVE COPYRIGHT NOTES

The software on this CD is copyright protected. Some packages like Perl are protected by the GNU public license. This license requires the inclusion of the source code for the distributed packages. For these programs the source code can be found in the "*<path_to_CD>*`/unix/src`" directory, the source code includes also the exact text of the license.

In addition this CD contains the Acrobat Reader from Adobe to display the PDF documentation. The official copyright statement follows:

Acrobat © Reader Copyright © 1987-1996 Adobe Systems Incorporated.

All rights reserved. Adobe and Acrobat are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

Hyperwave is the trademark of Hyperwave Software Inc., the Hyperwave logo is a trademark of Hyperwave Information Management GmbH. Hyperwave Information Server, Copyright 1998 Hyperwave Information Management, GmbH. All rights reserved.

5.2 NETSCAPE COPYRIGHT STATEMENT

Portions © Netscape Communications Corporation 1996, All Rights Reserved

INDEX