O_2Web
User Manual

C++

Release 5.0 - April 1998

Information in this document is subject to change without notice and should not be construed as a commitment by O_2 Technology.

The software described in this document is delivered under a license or nondisclosure agreement.

The software can only be used or copied in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's own use.

Copyright 1992-1998 O_2 Technology.

All rights reserved. No part of this publication can be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopy without prior written permission of O_2 Technology.

O_2, O_2API, O_2C, O_2DBAccess, O_2Engine, O_2Graph, O_2Kit, O_2Look, O_2Store, O_2Tools, and O_2Web are registered trademarks of O_2 Technology.

SQL and AIX are registered trademarks of International Business Machines Corporation.

Sun, SunOS, and SOLARIS are registered trademarks of Sun Microsystems, Inc.

X Window System is a registered trademark of the Massachusetts Institute of Technology.

Unix is a registered trademark of Unix System Laboratories, Inc.

HPUX is a registered trademark of Hewlett-Packard Company.

BOSX is a registered trademark of Bull S.A.

IRIX is a registered trademark of Siemens Nixdorf, A.G.

NeXTStep is a registered trademark of the NeXT Computer, Inc.

Purify, Quantify are registered trademarks of Pure Software Inc.

Windows is a registered trademark of Microsoft Corporation.

All other company or product names quoted are trademarks or registered trademarks of their respective trademark holders.

Who should read this manual

This manual describes how to develop a World Wide Web server using the O_2 system. It describes how to write in HTML and program an O_2Web server. The manual contains a comprehensive list of O_2Web methods and commands.

Other documents available are outlined, click below.

See O2 Documentation set.

Table of Contents

TABLE OF CONTENTS

1 Introduction

GENERAL OVERVIEW OF O_2WEB

Congratulations! You are now a user of O_2Web. You will find that building your WWW service on top of the O_2 database system has several key benefits.

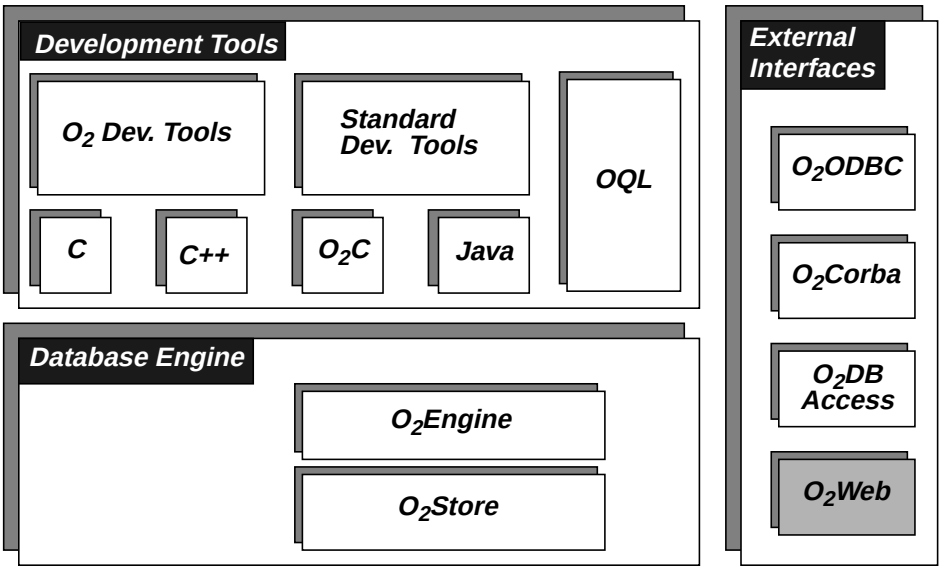
This chapter introduces O_2Web and is divided into the following sections:

- System overview
- The World Wide Web
- Features and advantages
- Manual overview

1 Introduction

1.1 System overview

O_2's system architecture is illustrated in the following figure.



O_2 System Architecture

The O_2 system consists of the database engine, development tools, and external interfaces.The database engine provides all the capabilities of a database and an object-oriented system. This engine is accessible by using programming languages, O_2 development tools, and other standard development tools. Numerous external interfaces are also provided.

The database engine consists of the following:

- O_2Store — The database management system (DBMS) provides low-level facilities, through O_2Store API, to access and manage databases: disk volumes, files, records, indexes, and transactions.
- O_2Engine — The object database engine provides direct control of schemas, classes, objects, and transactions through O_2Engine API. It provides full text indexing and search capabilities with O_2Search and spatial indexing and retrieval capabilities with O_2Spatial. It includes a Notification manager for informing other clients connected to the same O_2 server that an event has occurred, a Version manager for handling multiple object versions and a Replication API for synchronizing multiple copies of an O2 system.

Programming Languages:

You can create and manage O_2 objects by using the following programming languages:

- C — O_2 functions can be invoked by C programs.
- C++ — ODMG-compliant C++ binding.
- Java — ODMG-compliant Java binding.
- O_2C — An object-oriented fourth-generation language specifically for developing object database applications.
- OQL — ODMG-standard, SQL-like object query language capable of handling complex O_2 objects and methods.

O_2 Development Tools:

- O_2Graph — Creates, modifies, and edits any type of object graph.
- O_2Look — Designs and develops graphical user interfaces, provides interactive manipulation of complex and multimedia objects.
- O_2Kit — Library of predefined classes and methods for fast development of user applications.
- O_2Tools — Complete graphical programming environment for designing and developing O_2 database applications.

Standard Development Tools:

You can use any standard programming language system, such as Visual C++ and Sun Sparcworks and any supported operating system.

External Interfaces:

- O_2Corba — Creates O_2-Orbix servers for accessing O_2 databases with CORBA.
- O_2DBAccess — Connects O_2 applications to relational databases on remote hosts; invokes SQL statements.
- O_2ODBC — Connects remote ODBC client applications to O_2 databases.
- O_2Web — Creates O_2 World Wide Web servers for accessing O_2 databases through the internet network.

1.2 The World Wide Web

The World Wide Web (WWW or Web) is a protocol for exchanging and distributing hypermedia* information across the Internet network.

Since its creation at CERN in 1992, the Web has experienced a tremendous but unexpected success. Currently, more than one million Web users visit more than 10000 Web sites throughout the world.

Web servers provide interactive access to large amounts of complex multimedia, and distributed data including structured text, graphics, sound, and video.

*hypertext — linking elements, such as words and pictures in electronic documents to elements in other documents. When a user clicks on an linked element, the file linked to it displays.

hypermedia — an extension of hypertext that includes audio, video, and graphics.

1.3 Features and advantages

As an object database system, O_2 efficiently stores and manages large amounts of complex and multimedia data, including larg binary files.

O_2Web provides the benefits of database technology and object technology, in addition to a set of tools enabling rapid development and deployment of Web servers based on the O_2 system.

O_2Web provides clients with the ability to browse through hypermedia information stored in any O_2 database. The information presented to a Web client is a view of the objects in the database.

O_2 enables a client to address an unlimited number of objects. O_2 also implements management techniques such as buffering, indexing, clustering, and query optimization.

Based on O_2, your Web server benefits from the following additional capabilities:

- Information stored in O2 can be physically reorganized without any application modification. This physical/logical independence is ensured by the use of an associative query language, such as OQL.
- Data can be shared by several users connected through the Web or through other means [such as ?] (a standard O2 application).
- O2 enables fast recovery from any kind of software or hardware failure. [can we make such a claim without getting sued?]
- O2's development tools enable you to create, modify, and reuse classes and methods.
- O2 supports turnkey components to manage audio, video, and text data types and can interpret data from other sources.

1.4 Manual overview

This manual is divided into the following chapters:

- Chapter 1 - Introduction

  Introduces O_2Web and outlines some of its advantages.

- Chapter 2 - O_2Web Installation

  Describes how to install O_2Web.

- Chapter 3 - A World Wide Web Tour

  Gives an overview of the World Wide Web.

- Chapter 4 - O_2Web Overview

  Focuses on the differences between using the Web with file systems and using the Web with the O_2 system.

- Chapter 5 - Programming an O_2Web server

  Describes how to program an O_2Web server using exemples.

- Chapter 6 - O_2Web Reference

  Gives the full referential information for O_2Web.

2 O_2Web Installation

This chapter describes how to install O_2Web and is divided into the following sections:

- Requirements
- O2Web distribution
- Installation
- Launching O2Web

2 O2Web Installation

2.1 Requirements

In order to use O_2Web, you need a fully featured HTTP server.

O_2Web can be used with commercial servers (Netscape Communication Server, Netscape Commerce Server, Microsoft Information Server, etc.) and public domain servers (CERN httpd, NCSA httpd, etc.).

2.2 O2Web distribution

The O_2Web distribution contains the following:

- The **`o2web_gateway`** program. This is a CGI script that is called by the httpd server when an O_2 query is received.
- The **`o2open_dispatcher`** program. This program manages O_2Web activity. It knows every **`o2web_server`** running on your Local Area Network (LAN) and supplies each **`o2web_gateway`** with the address of the **`o2web_server`** best suited to answer a query.
- A library (**`libo2webserver.a`**) that is used by C++ programmers to build an O_2Web server.
- The **`o2webassistant`** library. This is provided as a C++ library located in **`$O2HOME/lib/libo2webassistant.a`**. The file **`o2web_CC.hxx`**, which is located in **`$O2HOME/include`**, contains the public interface of the library.
- A four step tutorial. This tutorial is installed in **`$O2HOME/samples/o2web/cplusplus`**. A README file in this directory explains how to use the examples. This tutorial is described in Chapter 5.

2.3 Installation

This section describes the different steps required to install O_2Web.

Specifying a port number to access o2open_dispatcher

In order to be reachable by **o2web_server** and **o2web_gateway**, a port number must be associated to the o2open_dispatcher service. This is done by adding an entry to the operating system file that contains the available services on the network.

The service name associated with **o2open_dispatcher** is **o2opendispatcher**. The port number can be any number that is not used by another service. Depending on the operating system (UNIX or Windows NT), the new entry must be inserted into one of the following files:

- **/etc/services** - for UNIX.
- **$WINDIR\system32\drivers\etc\services** - for Windows NT.

The new entry in the file is as follows:

```
o2opendispatcher                7999/tcp
```

Retrieving the dispatcher host name

To access the **o2open_dispatcher**, **o2web_gateway** and **o2web_server** must retrieve the machine on which **o2open_dispatcher** is running. This can be achieved using the following three techniques:

1. Looking for a command-line argument. This technique is only used by **o2web_server** and overrides any other means of retrieving the dispatcher host name.
2. Looking for the **O2OPEN_DISPATCHER** environment variable. This technique is used by **o2web_server** if the dispatcher host has not been retrieved by the command line. It can also be used by **o2web_gateway** if it is able to retrieve this variable. Certain HTTP servers only pass CGI variables to a CGI script. Thus, this technique cannot be used with these types of HTTP server.
3. Looking for the content of the file **/etc/o2openaccess** (UNIX) or **$WINDIR\system32\drivers\etc\o2openaccess** (Windows NT). This technique can be used by both **o2web_server** and **o2web_gateway** if the dispatcher host name has not been retrieved by any other means. This file must contain the dispatcher host name.

2.4 Launching O2Web

Running o2open_dispatcher

In order to start-up successfully, **`o2open_dispatcher`** needs the port number and the protocol used by the other programs to connect to it. This information is retrieved using the techniques described in the subsection Specifying a port number to access o2open_dispatcher.

This program has only one option (**`-v`**). This forces **`o2open_dispatcher`** to run in verbose mode.

It does not use any environment variables.

It must be running before **`o2web_server`** is launched.

Running o2server

Before running **`o2web_server`**, an **`o2server`** must be running. Consult the *O_2 System Administration Guide* for further details concerning **`o2server`**.

Running o2web_server

In order to start-up successfully, **`o2web_server`** needs the following information:

- The O_2 installation directory. This is given by the **`O2HOME`** environment variable (mandatory).
- The O_2 system to connect to. This is given by a directive in the .o2rc configuration file or by specifying a system name in the command line.
- The machine on which an **`o2server`** is running. This is given by a directive in the .o2rc configuration file or by specifying a server hostname in the command line.
- The machine on which an **`o2open_dispatcher`** is running. As explained in the subsection Retrieving the dispatcher host name, this information is retrieved by the **`o2web_server`** in a system-dependent file. It can be overridden by a directive in the .o2rc configuration file or by specifying a dispatcher hostname in the command line.
- The port number and the protocol used to connect to **`o2open_dispatcher`**. As explained in the subsection Specifying a port number to access o2open_dispatcher, this information is retrieved by the standard operating system mechanisms.

o2web_server can run on any machine on your LAN.

An O_2 server running on the same system must be active before **o2web_server** can be run. The **o2open_dispatcher** program must also be running before **o2web_server** can be run.

Running an HTTP server

To test your O_2Web service, you need an HTTP server. Any HTTP server can be used (commercial or public domain). Your server must be configured to call the **o2web_gateway** CGI script when a URL leading to O_2 is received. This installation is specific to each HTTP server and cannot be explained here. Refer to your HTTP server documentation for details about mapping URLs to CGI scripts.

After configuration, the HTTP server will run an **o2web_gateway** process each time a URL leading to O_2 is received.

To run successfully, **o2web_gateway** needs the following information:

- The machine on which the **o2open_dispatcher** is running. As explained in the subsection Retrieving the dispatcher host name, this information is retrieved by **o2web_gateway** in a system-dependent file and can overridden by the **O2OPEN_DISPATCHER** environment variable when running HTTP servers that transfer their own environment to CGI scripts.
- The port number and protocol used to connect to o2open_dispatcher. This information is retrieved by standard operating system mechanisms, as explained in the subsection Specifying a port number to access o2open_dispatcher.

3 A World Wide Web Tour

This chapter gives an overview of the World Wide Web.

It contains the following sections:

- The World Wide Web
- The HTML language
- Writing HTML documents
- Special characters in HTML text
- Special characters in URLs and form submissions
- HTML tags summary

3 A World Wide Web Tour

3.1 The World Wide Web

The World Wide Web (or Web or WWW) is a wide area client-server architecture for retrieving hypermedia information across the Internet. The Web has three main components:

- Universal naming scheme for documents. The Universal Resource Location (URL) syntax specifies documents in terms of the protocol to be used in order to retrieve them, their Internet host and path name.
- Use of available protocols for retrieving documents over the network, including FTP, NNTP, WAIS, Gopher, and HTTP. The latter is designed specifically for use with the WWW, and combines efficiency with an ability to flexibly exchange information between clients and servers.
- A document format (HTML) supporting hypertext links based on URLs which can specify documents anywhere on the Internet. HTML is designed for rendering on a wide variety of different display types and platforms.

3.2 The HTML language

The Hypertext Markup Language (HTML) is the language used to write documents for the Web. Applications designed for the Web (usually called Web browsers) can read HTML documents and format them with text, graphics, tables and links to other HTML documents.

HTML is a markup language - in fact it is a specific implementation of the Standard Generalized Markup Language (SGML) - and is concerned with the structure of documents rather than their appearance. This feature gives HTML documents portability across different platforms or media. It is up to the browser reading an HTML document to map the structure into a physical format.

An HTML document is made of structure commands and plain text. The structure commands are called tags. A tag begins with a **<** and ends with a **>**. HTML tags can be either separator tags or container tags. A container tag is made of two parts: the beginning tag **<X>** and the ending tag **</X>**. The command specified is then applied to the text between the two tags. A separator tag is a "one shot" command; for instance the **<hr>** tag, which inserts an horizontal rule line, is a separator tag whereas the **** tag, which makes the text contained between the **** tag and the **** tag bold, is a container tag.

3.3 Writing HTML documents

This section introduces the HTML language and the tags most commonly used for writing standard documents. It is not a complete HTML manual and is intended to help readers unfamiliar with the Web to understand the O_2Web basics.

An HTML document

Every HTML document has a common structure.

It declares itself as an HTML document with the **`<html>`** tag and ends with the **`</html>`** tag.

It comprises two main parts: the header and the body.

An HTML document has the following structure:

```
<html>
<head>
.....
</head>

<body>
.....
</body>
```

The header

The header content is not usually displayed by the browser in the document window. It contains information that can be either displayed in a separate window or when document information is requested. The most important command that must always be present in the header is the **<title>** command. This label is used, for example, when you store a reference to a document in your browser's "hotlist". Other information, such as the creator of the document, the creation date, etc., can also be written in the header.

```
<html>
<head>
<title> This is the title of my document </title>
</head>

<body>
.....
</body>
</html>
```

The body

The body of an HTML document contains both the structure and the content of the document.

Heading Levels

Heading tags are used to organize your document into a hierarchical structure. Different heading levels exist from level 1 to level 6. Each heading level puts text inside it with a particular font size, font attributes, etc.

Usually, the level 1 header (**<h1>**) is used for writing the title of your document, the level 2 header (**<h2>**) for the title of the document sections, the level 3 (**<h3>**) for the subsections, etc.

This kind of document structure is not forced by HTML, for which a header tag indicates only that the text between the header beginning tag and the header ending tag must have a particular style, but it is considered good practice to organize an HTML document in this way.

```
<html>
<head>
<title> This is the title of my document </title>
</head>

<body>
<h1> This is the title of my document </h1>

<h2> First Section </h2>

<h3> First SubSection </h3>

......
<h3> Second SubSection </h3>
......
<h2> Second Section </h2>
......
</body>
</html>
```

Paragraphs

The paragraph tag (**<p>**) is a separator tag. It cuts a piece of text into two different paragraphs. Most browsers when finding a paragraph tag, insert a blank line to separate the paragraphs. If you just want to break the current line, you can use the **
** tag.

```
<html>
<head>
<title> This is the title of my document </title>
</head>

<body>
<h1> This is the title of my document </h1>

<h2> First Section </h2>

This is the text of my first paragraph.
<p>
This is my second paragraph. It is cut here <br>
and continued on the next line.

......

</body>
</html>
```

Enumeration Lists

The list tags are very useful when the document must integrate enumeration of items. There are two kinds of lists: ordered lists and unordered lists.

When using ordered lists, each item appears preceded by its rank in the list; when using an unordered list, each item appears preceded by a marker (usually a bullet).

The tag corresponding to an ordered list is **<ol>** and the tag corresponding to an unordered list is **<ul>**. A separator tag (**<li>**) is used to indicate a new item in a list.

Enumeration list can be nested. In this case, visual outline effects are usually provided by most browsers.

```
<html>
<head>
<title> This is the title of my document </title>
</head>

<body>
<h1> This is the title of my document </h1>

<h2> First Section </h2>

......

<h3> First SubSection </h3>
<ol>
   <li> My first item
   <li> My second item
   <ul>
      <li> My first sub item
      <li> My second sub item
    </ul>
   <li> My third item
 </ol>

......

</body>
</html>
```

Definition Lists

A definition list is quite useful for expanding items in a list. The tag for defining a definition list is the **`<dl>`** tag. Each item in a definition list has:

a definition term (**`<dt>`**)

and definition data (**`<dd>`**) expanding or explaining the term

Definition lists are a very commonly used structure in HTML documents.

```
<html>
<head>
<title> This is the title of my document </title>
</head>

<body>
<h1> This is the title of my document </h1>

...

<h3> Second SubSection </h3>

<dl>

<dt> first item title
<dd> first item development or explanation

<dt> second item title
<dd> second item development. Note that the
definition       data (such as any html tag
content) can be as complex as you want and
contains any HTML construction. <p>
Here we just add a new paragraph.

 </dl>

......

</body>
</html>
```

Hypertext Links

Hypertext links are a key feature of HTML and are what makes the Web so exciting for many people. They are not only a way of pointing to another document stored on a disk but a way of pointing to another document located anywhere else on the internet network.

At this point, it is important to explain how a document located somewhere on the internet is retrieved by your favorite Web browser. The World Wide Web is based on a communication protocol, the HyperText Transfer Protocol (HTTP) and documents are referenced

using something called a Uniform Resource Locator (URL). A simple URL might have the following format:

```
protocol://<machine internet
address>[:port]/path/document
```

where:

- the **protocol** is usually HTTP but can also be another communication protocol such as FTP, NNTP, WAIS, Gopher, etc. In such cases, the URL is not used to retrieve an HTML document but for other purposes that do not enter into the scope of this manual,
- the **machine internet address** is an internet host on which an HTTP server is running,
- the optional **port** is the TCP port number on which the HTTP server is accessible. The standard HTTP port number is 80. This means that when accessing an HTTP server running on port 80, it is unnecessary to specify the port number in the URL,
- the **path** is something that will be mapped by the HTTP server (according to some configuration rules) into a physical path in its file system,
- the **document** is a reference to a file containing an HTML document.

To insert a link to another document in an HTML document you use the anchor tag (**<a>**). The anchor beginning tag contains an attribute specifying the URL of the linked document. The text between the anchor beginning tag and the anchor ending tag will be highlighted by the browser (usually underlined). The appropriate document will be retrieved when the user clicks on this text.

```
<html>
<head>
<title> This is the title of my document </title>
</head>

<body>
<h1> This is the title of my document </h1>

...

<h2> Second Section </h2>

The second section of the document is not here.
It can be retrieved <a
href="http://xx.xx.xx/doc/sec2.html"> here. </a>

</body>
</html>
```

Inline Images

Another exciting feature of the Web is its multimedia orientation. Although documents integrate images, sound or video, only images and bitmaps can be put inline in HTML documents. The only image format that can be displayed by all browsers is the GIF format. It is therefore recommended that you use GIF images in your HTML documents. However, another format (JPEG) is becoming increasingly popular and can be put inline by many recent browsers.

An image is inserted in an HTML document by means of the **`<img>`** tag. This tag contains an attribute (**`src`**) specifying the URL of the image file.

The following optional attributes can be used with the **`<img>`** tag:

- the **`align`** attribute specifies the position of the image relative to the text. The possible values for this attribute are **`top`**, **`middle`** or **`bottom`**. Some browsers recognize other values but these are nonstandard.
- the **`alt`** attribute specifies a text label to be displayed instead of the image when a browser can not display images or when a browser is configured to only show images on demand.
- the **`ismap`** attribute specifies the image as a clickable image. This attribute is discussed in more detail in the section "clickable images".

Do not forget that inline images, although greatly improving the look of your documents, considerably slow down the retrieval of such documents. Keep this in mind when designing documents.

```
<html>
<head>
<title> This is the title of my document </title>
</head>

<body>
<h1> This is the title of my document </h1>

...

<h2> Second Section </h2>

The second section of the document is not here but
can be retrieved <a
href="http://www.o2tech.com/doc/sec2.html">
here. </a>

What about using an <br>
<img src="http://xx.xx.xx/doc/step1.gif">
image separation line.

</body>
</html>
```

The complete document built so far has the following form:

```
<html>
<head>
<title> This is the title of my document </title>
</head>

<body>
<h1> This is the title of my document </h1>

<h2> First Section </h2>

This is the text of my first paragraph.
<p>
This is my second paragraph. It is cut here <br>
and continued on the next line.

<h3> First SubSection </h3>
<ol>
   <li> My first item
   <li> My second item
   <ul>
      <li> My first sub item
      <li> My second sub item
   </ul>
   <li> My third item
</ol>

<h3> Second SubSection </h3>

<dl>

<dt> first item title
<dd> first item development or explanation

<dt> second item title
<dd> second item development. Note that the
definition data (such as any html tag content)
can be as complex as you want and contains any
HTML construction. <p>
Here we just add a new paragraph.

</dl>
```

```
<h2> Second Section </h2>

The second section of the document is not here but
can be retrieved <a
href="http://ww.o2tech.com/doc/sec2.html"> here.
</a>
What about using an <br>
<img src="http://xx.xx.xx/doc/step1.gif">
image separation line.

</body>
</html>
```

This document will have the following appearance when viewed using a typical Web browser:

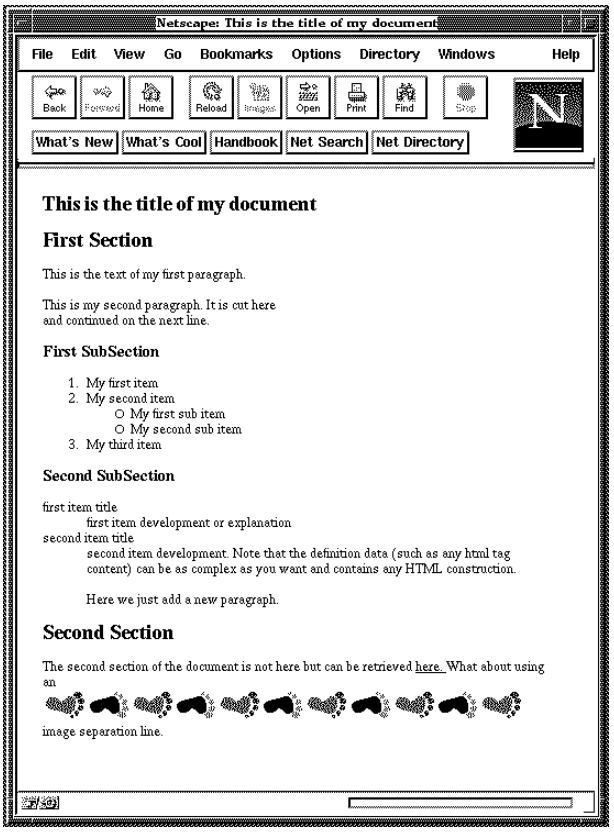


Figure 3.1: A simple HTML page

Clickable images

A clickable image (or imagemap) is an inline image in which sensitive zones are defined. A URL is associated with each of these zones. The appropriate URL will be resolved when the user clicks on one of these zones. An imagemap is declared by the **`ismap`** attribute of the **`<img>`** tag.

Zones of an image are described in a file called a map file. Such a file contains, on each line, the specification of a region and its corresponding URL. The syntax of a map file is as follows:

- **`point URL x,y`** specifies a clickable point on the image. This is useful if a user clicks on an undefined area because the closest defined point is used.
- **`circle URL x,y x,y`** specifies a circle by the coordinates of both its center and any point on its circumference.
- **`rect URL x,y x,y`** specifies a rectangle by its upper left and lower right corners.
- **`poly URL x,y x,y ...`** specifies a polygon of up to 100 sides. Each (x,y) pair is a point where two sides of the polygon meet. The last point is always connected to the first one.
- **`default URL`** defines the default URL to be used if a user clicks on a point outside a defined region. When a point region is defined in the image, the default is never used.

Forms

HTML forms are a way of getting information from a user connected to your Web server. Forms can be used to insert buttons, input text fields, menus, etc. A form is something included inside the **`<form>`** and **`</form>`** tags. Two main attributes can be given to the **`<form>`** tag:

- **`action`** is the URL of a program to which the form content will be submitted. If this attribute is missing, the current document URL will be used. Forms usually use a special kind of URL that contains a path to a program instead of a path to a file. Such a program is called a CGI (Common Gateway Interface) script; this is a program that understands a very simple protocol allowing it to get information from the HTTP server. Writing a CGI script is out of the scope of this manual. To know more about writing CGI scripts, you should consult one of the many courses available on the internet. A good starting point is the Web Consortium server (**`http://www.w3.org`**).
- **`method`** is the HTTP method used to submit the fill-out form to a query server. The **`method`** attribute values can be **`get`** or **`post`**. When using the **`get`** method, the fill-out form content is appended to the

URL; if the **post** method is used, the fill-out form contents are sent to the server in a data block. It is highly recommended to use post methods when writing forms as this technique does not have the limitations **get** methods can have when the fill-out form contents are large.

The form content is built using several tags:

- The input tag is used to specify a simple element inside a form. There is no corresponding ending tag. The possible attributes of the input tag are as follows:
  - **type** must be one of:

    text: a text entry field (default)

    password: a text entry field where typed characters are represented as asterisks.

    checkbox: a single toggle button; on or off.

    radio: a single radio button; on or off. Other radio buttons with the same name are grouped into "one of many" behavior.

    submit: a push-button that computes the fill-out form contents, formats it and resolves the URL in the **action** attribute of the form, appending the form contents to the URL (**get** method) or posting the form contents (**post** method).

    reset: a push-button that resets the different fields in the form to their default values.

    hidden: a hidden text field in the created HTML form. It is useful for contextual information.
  - **name** is the symbolic name for this input field. This is a mandatory attribute except for input tags of the type submit or reset. The name is used when building the formatted fill-out form contents transmitted to the server.
  - **value** can be used to specify the default value of a field (for a text or password input field). It can also be used to specify (for a checkbox or a radio button field) the value of a button when it is checked; the default value for a checkbox or radio button is "on". It can also be used to specify the label of a push-button.
  - **checked** specifies that a checkbox or a radio button is checked when displayed.
  - **size** is the size, expressed in characters, of text fields and password fields.
  - **maxlength** is the maximum number of characters accepted as input in text and password fields.
- The select tag is used to insert an options menu or scrollable lists into a form. It is used as follows:

```
<select name="my-menu">
     <option> first item
     <option> second item
     <option> third item
</select>
```

The possible attributes of the **`select`** tag are:

- **`name`** is the symbolic name for this element. This is a mandatory attribute of the tag. The name is used when building the formatted fill-out form contents transmitted to the server.
- **`size`** is a specified integer. When its value is 1 (or if the attribute is missing) then the form element will be represented as an options menu. If **`size`** is greater than 1, the form element will be represented as a scrollable list. The value of **`size`** determines how many items are visible in the window.
- **`multiple`** specifies that the user can make multiple selections ("n of many" behavior). This attribute forces the form element to be a scrollable list regardless of the **`size`** attribute value.

• The **`textarea`** tag can be used to insert a multi-line text area in a form.

• The attributes of a **`textarea`** are as follows:

- **`name`** is the symbolic name of the field
- **`rows`** is the number of rows in the text area
- **`cols`** is the number of columns in the text area.

The following example shows a complete form:

```
<html>
<head>
<title> Form Example </title>
</head>
<body>
<h1><center> A Form Example </center></h1>
<form method="POST"
action="http://www.site.domain/cgi/getperson">

<hr>
Mr  <input type ="radio" value="Mr" name="KIND" checked
>
Mrs <input type ="radio" value="Mrs" name="KIND">
<hr> <p>

<dl>
<dt>Enter your first name below:
<dd><input values="" size="40" name="FN">
<p>
<dt>Enter your last name below:
<dd><input values="" size="40" name="LN">
<p>
<dt>Enter your address below:
<dd><textarea rows=5 cols=40 name="ADDR"></textarea>
</dl>

<dl>
<dt> My hobbies are: <p>
<dd> <select name="HOBBIES" multiple size=3>
<option> skiing
<option> swimming
<option> playing golf
<option> playing tennis
<option> chess
</select> <p>
</dl>
<p> <hr> <p>
<p>
To submit the query, press this button:<br>
<input type="submit" value="Submit">
<hr>
</form>
</body>
</html>
```

This document will have the following appearance when viewed by a typical Web browser:

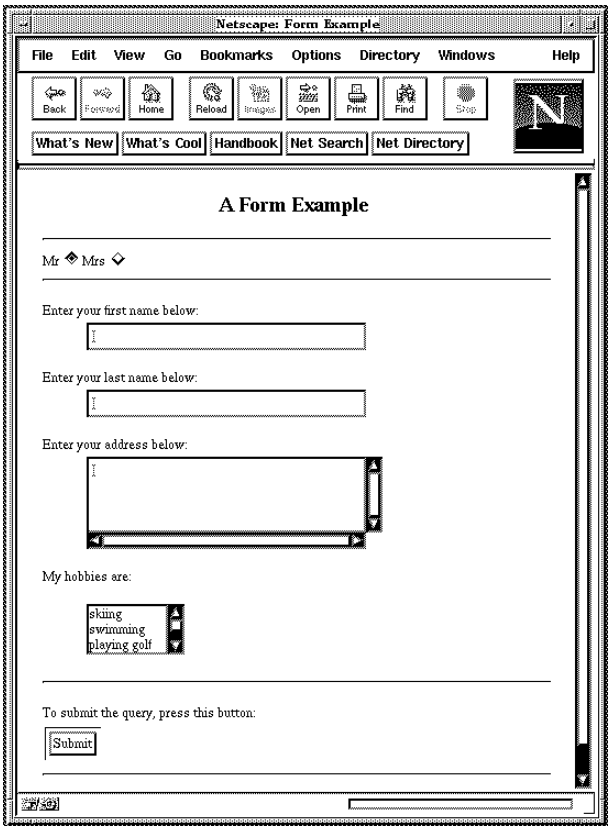


Figure 3.2: An HTML form example

In the above example, when the user clicks on the submit button, the URL specified in the action attribute will be resolved and the specified CGI script will be executed. As the method is a post method, a data block containing the result of the form will be sent to the server. It is up to the CGI script to read this data block. In this example, the following data will be sent:

```
KIND=Mr&FN=John&LN=Smith&ADDR=this%20is%20an%20ad
dress&HOBBIES=skiing&HOBBIES=chess
```

In the data block above, some specificities of the format generated by a fill-out form are as follows:

- Each value retrieved in the form is built as **`name=value`** where **`name`** is the attribute name of the field and **`value`** is either the value entered by the user or the default value. All the input field results are put together, separated by the **`&`** character. For a multiple list input field, each selected value is repeated **`name=value1&name=value2`**,
- Some characters are replaced by codes. This is discussed in the following section.

3.4 Special characters in HTML text

Some characters have special meaning within HTML and therefore cannot be used "as is" in text. These characters are, for example, the angle bracket or the quote characters.

As the Web is multiplatform, only a reduced set of characters can be typed in a text. Namely, only lower ASCII characters (all the characters of an English keyboard) can be used. Upper ASCII characters cannot be used directly and must be "escaped".

An escape sequence can be either character references or entity references.

A character reference has the format &#nnn, where nnn is a number that references the character.

An entity reference has the format &nnn, where nnn is a text string that references the character.

The following table summarizes the special characters in HTML text.

TABLE 3.1 Special characters in HTML text

Character	Reference	Entity
"	&#34	&quot
&	&#38	&amp
<	&#60	&lt
>	&#62	&gt
ª	&#170	&ordf
«	&#171	&laquo
¬	&#172	&not
-	&#173	&shy
®	&#174	&reg
¯	&#175	&macr
°	&#176	&deg
	&#177	&plusmn
	&#178	&sup2
	&#179	&sup3
´	&#180	&acute

	&#181	&micro
¶	&#182	&para
·	&#183	&middot
¸	&#184	&cedil
	&#185	&sup1
º	&#186	&ordm
»	&#187	&raquo
	&#188	&frac14
	&#189	&frac12
	&#190	&frac34
¿	&#191	&iquest
À	&#192	&Agrave
Á	&#193	&Aaute
Â	&#194	&Acirc
Ã	&#195	&Atilde
Ä	&#196	&Auml
Å	&#197	&Aring
Æ	&#198	&AElig
Ç	&#199	&Ccedil
È	&#200	&Egrave
É	&#201	&Eacute
Ê	&#202	&Ecirc
Ë	&#203	&Euml
Ì	&#204	&Igrave
Í	&#205	&Iacute
Î	&#206	&Icirc
Ï	&#207	&Iuml
	&#208	&ETH
Ñ	&#209	&Ntilde
ò	&#210	&Ograve
ó	&#211	&Oacute
ô	&#212	&Ocirc
õ	&#213	&Otilde

ö	&#214	&Ouml
	&#215	
Ø	&#216	&Oslash
Ù	&#217	&Ugrave
Ú	&#218	&Uacute
Û	&#219	&Ucirc
Ü	&#220	&Uuml
	&#221	&Yacute
	&#222	&THORN
ß	&#223	&szlig
à	&#224	&agrave
á	&#225	&aacute
â	&#226	&acirc
ã	&#227	&atilde
ä	&#228	&auml
å	&#229	&aring
æ	&#230	&aelig
ç	&#231	&ccedil
è	&#232	&egrave
é	&#233	&eacute
ê	&#234	&ecirc
ë	&#235	&euml
ì	&#236	&igrave
í	&#237	&iacute
î	&#238	&icirc
ï	&#239	&iuml
	&#240	&eth
ñ	&#241	&ntilde
ò	&#242	&ograve
ó	&#243	&oacute
ô	&#244	&ocirc
õ	&#245	&otilde
ö	&#246	&ouml

	&#247	
ø	&#248	&oslash
ù	&#249	&ugrave
ú	&#250	&uacute
û	&#251	&ucirc
ü	&#252	&uuml
	&#253	&yacute
	&#254	&thorn
ˇ	&#255	&yuml

3.5 Special characters in URLs and form submissions

Section 3.4 was only concerned with characters inside HTML text. Another kind of encoding must be used inside a URL or to decode a string resulting from an HTML form submission.

In a URL, characters may be encoded by a triplet consisting of the character " % " followed by the two hexadecimal digits which form the hexadecimal value of the characters. A character must be encoded if it has no corresponding graphic character in the US-ASCII coded character set, if the use of the corresponding character is unsafe, or if the corresponding character is reserved for a specific purpose.

No corresponding graphic in US-ASCII

The characters between 7F and FF hexadecimal and control characters (between 00 and 1F) must be encoded.

Unsafe characters

The unsafe characters are:

```
space < > « # % { } | \ ^ ~ [ ]  `
```

Reserved characters

The reserved characters are:

```
; / ? : @  = &
```

3.6 HTML tags summary

Previous sections introduced the main tags in the HTML 2.0 specification. Many others tags exist and many new ones may appear as HTML evolves. For example, many recent browsers such as Netscape Navigator© introduce tags that are part of the HTML 3.0 specification. These tags relate to tables, text centering and font size.

The table below summarizes the main tags in the HTML 2.0 specification.

TABLE 3.2 The main tags in the HTML 2.0 specification

Tag name	Description
HTML	Defines the file as an HTML document
HEAD	Defines the heading for the document
TITLE	Defines the title of the document
BODY	Defines the document body
H1...H6	Defines the six levels of headings
P	Defines a paragraph
OL	Defines an ordered list
UL	Defines an unordered list
DIR	Defines an unordered list with several items per line
MENU	Defines an unordered list, typically with one line per item
LI	Defines an individual item in a list
DL	Defines a definition list
DT	Defines an individual definition term of a definition list
DD	Defines individual definition data of a definition list

EM	Emphasizes characters (usually italic)
STRONG	Strongly emphasizes characters (usually bold)
CODE	Displays text in a fixed-width font (usually courier)
B	Displays text in bold
I	Displays text in italics
TT	Displays text in typewriter-like font
IMG	Inserts an image in the document
HR	Inserts a horizontal rule line
BR	Breaks the current line
A	Defines an anchor to another document
PRE	Inserts preformatted text
ADDRESS	Inserts text in address-like formatting (italic, smaller font)
BLOCKQUOTE	Defines text quoted from another source

3 A World Wide Web Tour

4 O2Web Overview

This chapter focuses on the differences between using the Web with file systems (introduced in the previous section) and using the Web with the O_2 system.

It contains the following sections:

- Principles
- O2Web Architecture
- Building your O2Web server

4 O2Web Overview

4.1 Principles

A user connecting to a Web server built on top of O_2Web will directly access objects in an O_2 database rather than files.

In order to specify an object to be retrieved in the database, a Web client specifies an OQL query rather than a directory path to a file.

OQL is the standard object query language, defined by the ODMG (Object Database Management Group) to query object databases. OQL is an object extension of SQL, it allows complex object databases to be queried and object methods to be invoked.

For example, the query:

```
select distinct employee.salary
from employee in All_employees
where employee.name like "Mac*"
```

returns the salary (or salaries) of all the employee(s) whose name begins with **`Mac`**.

For further information about OQL, please refer to either the OQL User Manual from O_2 or the ODMG-93 standard[1].

To make OQL queries enter in the framework of URLs, O_2Web uses the same kind of URL as the one used by HTML fill-out forms. These URLs have the following form:

```
http://host[:port]/path/script/
[extrapath][?search]
```

This is the more general form of URLs where:

- **`host`** is the internet host on which an HTTP server is running.
- **`port`** is the TCP port number on which the server is accessible.
- **`path`** is a logical path translated by the HTTP server into a physical path on the file system.

1. The Object Database Standard: ODMG - 93. Atwood, Duhl, Ferran, Loomis and Wade. Edited by R.G.G. Cattell. Copyright 1994 Morgan Kaufmann Publishers.

- **`script`** is the name of a program compliant with the CGI protocol.
- **`extrapath`** is a set of strings (separated by the / character) that can be retrieved by the script.
- **`search`** is a string that can be retrieved by the script.

When a Web server is built on top of O_2Web, a URL that accesses this server must comply with the above URL form. The **`script`** component is a program (**`o2web_gateway0`**) provided with O_2Web, the **`extrapath`** component contains the name of an O_2 system and the name of an O_2 database. The **`search`** component contains an OQL query. The result of this query is an HTML view of an object in the database.

A key feature of O_2Web is that it can be used at different levels. Starting from a completely automatic mode where HTML is generated for any object of the database, the programmer can progressively improve the Web service by overloading the generic mode for some classes of a schema.

4.2 O2Web Architecture

Three elements comprise O_2Web. These elements are used with a standard ful- featured WWW server (Netscape server, CERN HTTPD, NCSA HTTPD, etc.):

- An O_2Web gateway.
- An O_2Web server, which you must build.
- An O_2Web dispatcher.

An O_2Web server may be accessed from any standard Web client (such as Mosaic or Netscape).

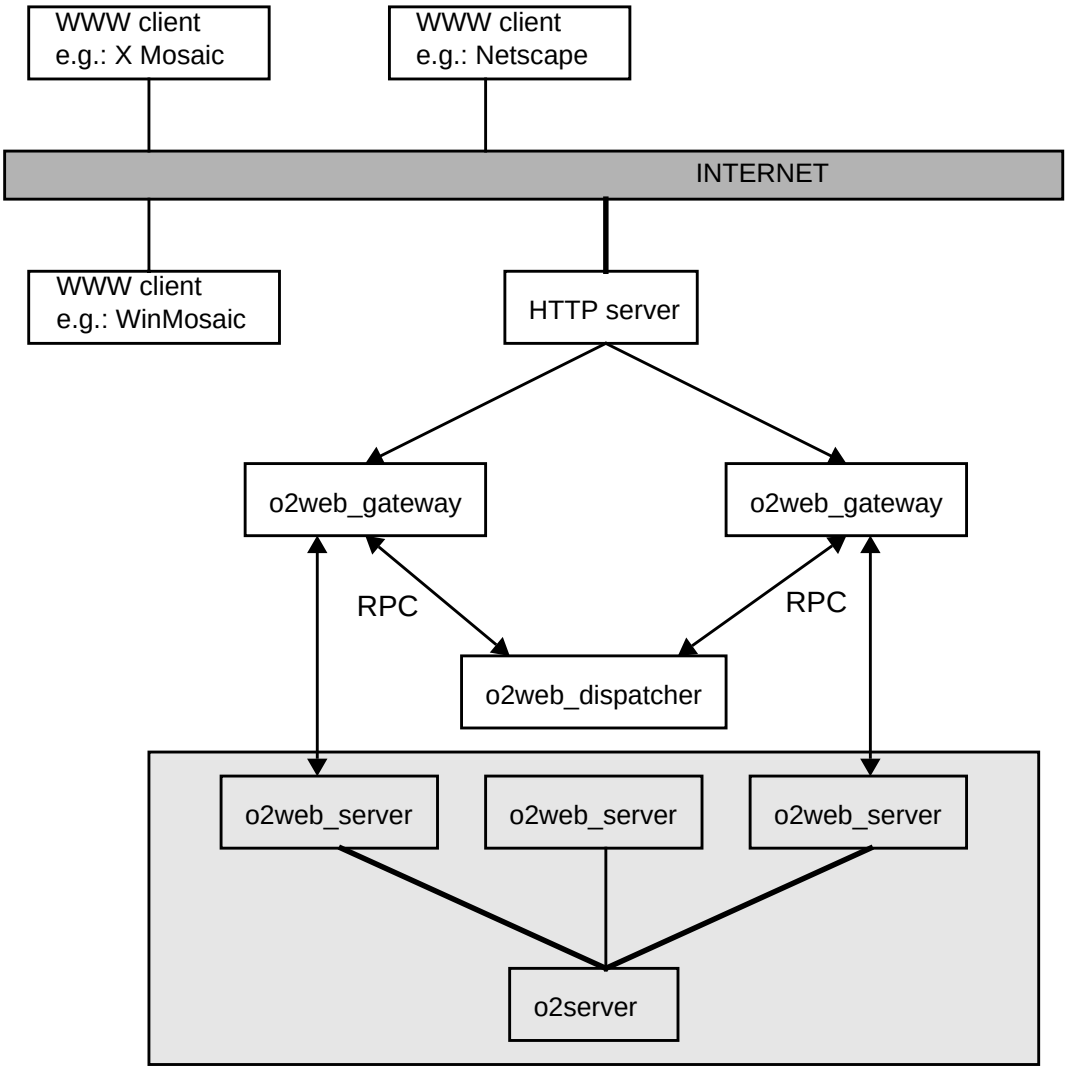


Figure 4.2: O_2Web Architecture

The O_2Web system works in the following way:

- A Web client sends a URL in HTTP format. This URL contains an OQL query.
- The HTTP server passes the query to the O_2Web gateway.
- The O_2Web gateway connects to an O_2Web dispatcher running on your local area network.
- The O_2Web dispatcher tells the O_2Web gateway which O_2Web server to connect to.
- The O_2Web gateway connects to the appropriate O_2Web server.
- The O_2Web server runs the query specified in the URL and transforms the result of the query into HTML (or other formats such as GIF and BIT-MAP when required).
- The resulting data is sent back to the Web client.

4.3 Building your O2Web server

The O_2Web product allows you to use the O_2 ODMG C++ binding to develop your Web applications.

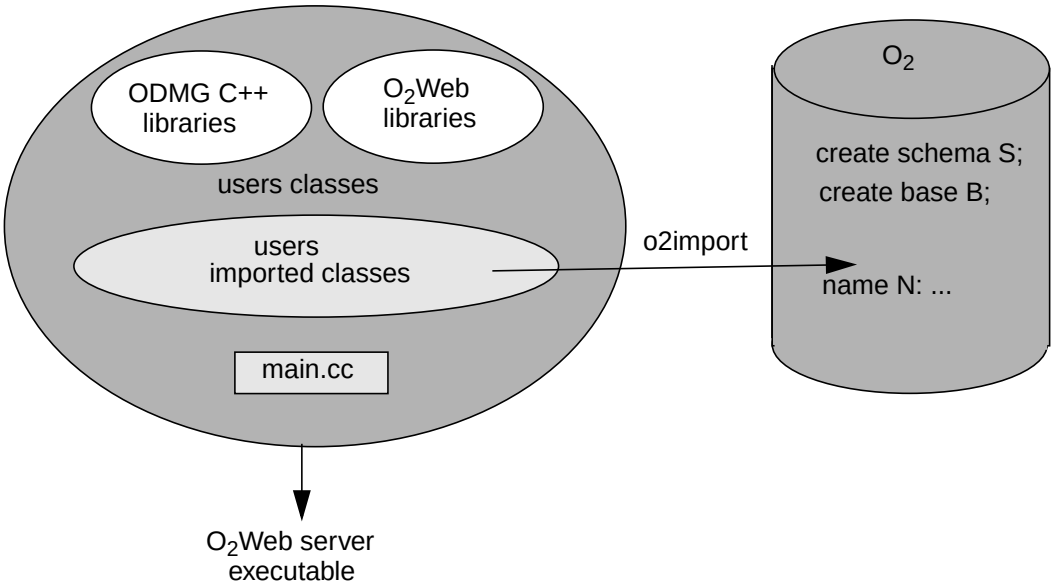


Figure 4.2: Components of an O_2Web application

To implement your O_2Web server you build a ODMG C++ application using the following:

- user classes
- ODMG C++ libraries
- O_2Web libraries

You must build an O_2Web executable. This executable is the result of a file which contains the main function and the application files. This main function uses the **`o2_Web`** class. The O_2Web server is linked with the **`o2webassistant`** (if necessary), **`o2link`**, **`o2sql`**, and **`o2`** libraries.

Some user classes must be imported into O_2. For an object to be published on the Web, it must belong to a persistent capable class.

5 Programming an O2Web Server

This chapter describes how to program an O_2Web server.

It is divided into the following sections:

- Introduction
- Summary
- Generic mode
- Global Personalizations
- Local Personalizations
- Summary

5 Programming an O2Web Server

5.1 Introduction

O_2Web can be used at different levels.

The first (and simplest) level is to let O_2Web generate HTML for the result of the query contained in the URL. This is the generic mode of O_2Web.

The second level allows programmers to globally change parts of the HTML generation. The data retains the same generic look, but HTML text may be inserted at the top or at the bottom of pages. This level also allows you to react to some events such as a connection or a disconnection to the server (in order to maintain a log in the database), or when an error occurs (in order to personalize the error message the client will receive).

The last level allows total control of HTML generation for each class of the working schema. This means that a programmer can specify HTML text for the objects of each class. As this text is built by a member function, it can be made to depend on the values of objects. The programmer can also define, for each class, the HTML text that will be inserted at the top and at the bottom of each page when an object of this class is the entry point of an HTML report. To use this level you must use the C++ library **`$O2HOME/lib/libo2webassistant.a`**, which is delivered with the O_2Web product.

5.2 Summary

HTML code responding to URL queries is made of five parts:

- prolog (protocol-specific)
- header
- body
- footerr
- epilog (protocol-specific)

Default HTML code for producing these parts is predefined by the O_2Web server. You can accept the generic code, or you can redefine all or part of it.

The following illustration describes the makeup of HTML production code

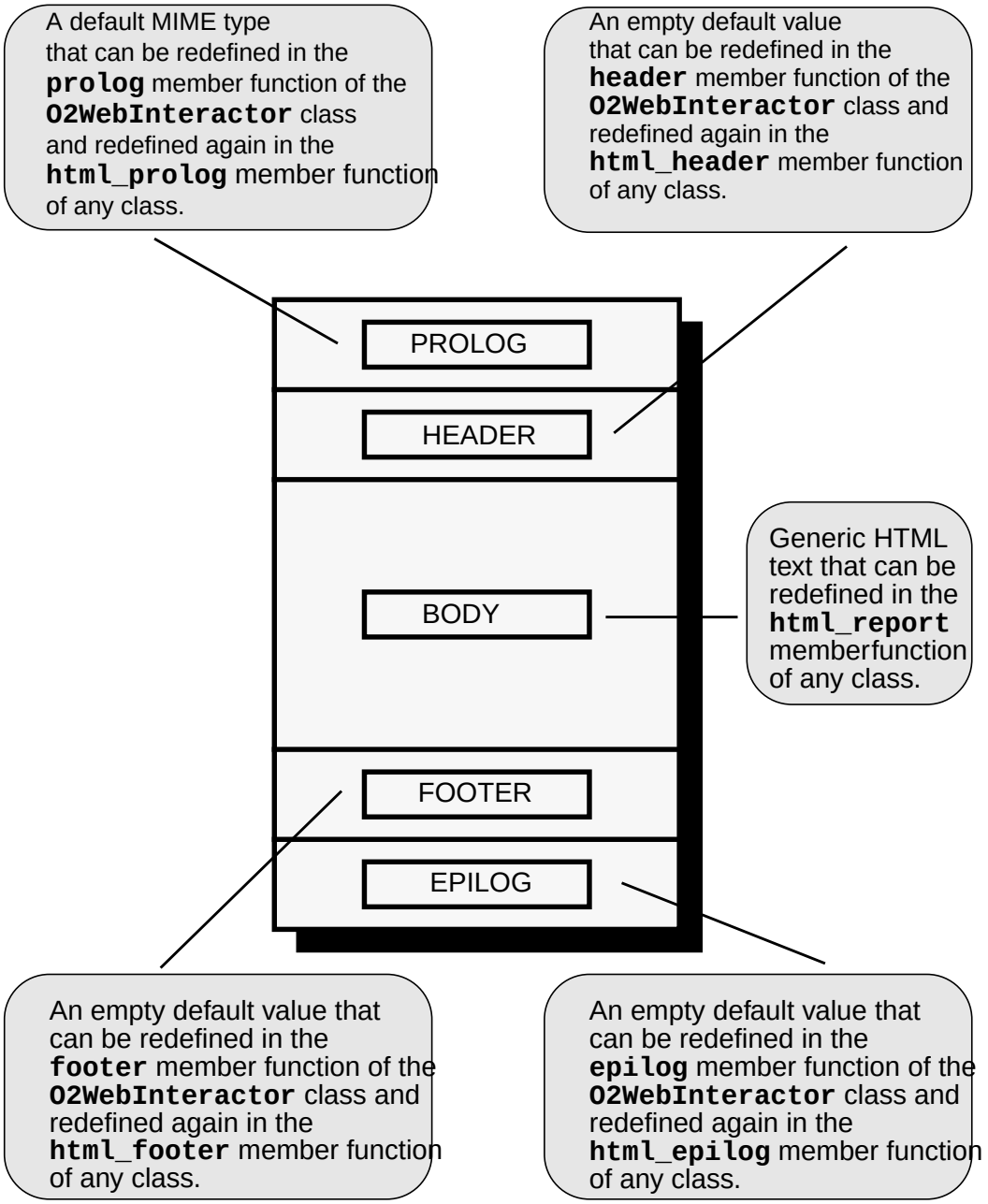


Figure 5.1: HTML production code makeup

5.3 Generic mode

Simple Browsing

The first step when one wants to provide a Web interface to an O_2 database is very simple. Just install O_2Web on your system (refer to Chapter 1 for installation details). You are ready to browse your database with a Web browser.

To begin browsing, you must provide your browser with a URL (or click on an already existing link found somewhere on the internet). The kind of URLs given to start a browsing session usually point to a persistent root of the database. Such a URL could be:

```
http://xx.xx.xx/cgi/o2web_gateway/sysname/basename?rootname
```

This URL will return an HTML view of the specified root of persistence. You can now click on the links on this page to continue browsing.

How does it work

Let us now explain how the generated HTML is built.

System-supplied member functions can generate HTML for any object in the database, however complex it is. The generated HTML is based on the structure type of the object being processed. The exact behavior of the system-supplied member functions is as follows:

- atomic values: integers, chars, booleans, reals, strings, bytes. The value is printed except for bytes which are not displayed.
- tuple values: A **`<ul>`** HTML list is used, each attribute of the O_2 tuple being an item in the list. Each item is composed of an attribute name and the result of the recursive call to an HTML production member function on an attribute value.
- collection values: lists, sets. A **`<ul>`** HTML list is used, each element in the O_2 collection being printed as an item of the list. Each item is composed of the result of the recursive call to an HTML production member function on a collection element.
- objects: An object is printed by the recursive call to an HTML production member function on the encapsulated value.

- sub-objects: An HTML anchor is generated with a reference to a URL that contains an OQL query returning this sub-object.

  You obtain the text of the anchor by calling the **`html_title`** member function on the subject. If this member function does not exist, the name of the class is used.

A Guided Example

Let us take an example. We define in the database the schema of a very simple phone book. The aim of this example is to demonstrate how bases of this schema can be made accessible from the Web. The schema and the member function code of this simple application can be found in the O_2 installation in the **`$O2HOME/samples/o2web/cplusplus/step1/`** directory. The implementation of some member functions, unnecessary for the purposes of this example, will not be detailed in this manual. For further information concerning these member functions, please refer to the samples directory of O_2.

First we create the **`Directories`** class. This is the container of all the directories we will create in the database.

```
class Directories {
      public
            d_Set<d_Ref<Directory> > directories;

            d_String html_title();
            d_Ref<Directory> add_directory(char* name);

            Directories(){};
            ~Directories(){};
};
```

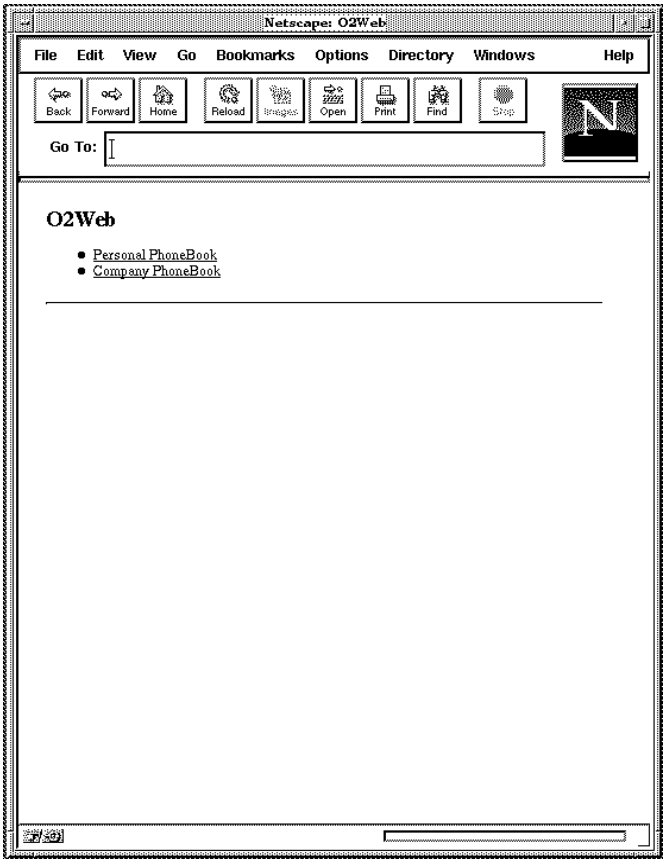


Figure 5.2: The Directories class

We now define the **`Directory`** class. This has a name and an attribute called **`entries`**. This attribute refers to the **`Entries`** class whose type is an enumeration of **`Entry`**.

```
class Directory {
      public:
            char* name;
            d_Ref<Entries> entries;

            d_String html_title();
            void new_entry(char* name,
                  d_ref<Picture>,
                  char* address,
                  char* phone,
                  char* e_mail);

            Directory();
            Directory(char* name);
            ~Directory();
            Directory & operator=(const Directory & dir);
}


class Entries {
      public:
            d_List<d_Ref<Entry> >Entries_value;

            d_String html_title();
            void insert(d_Ref<Directory> d,
            d_Ref<Entry> entry);

            Entries(){};
            ~Entries(){};
};
```

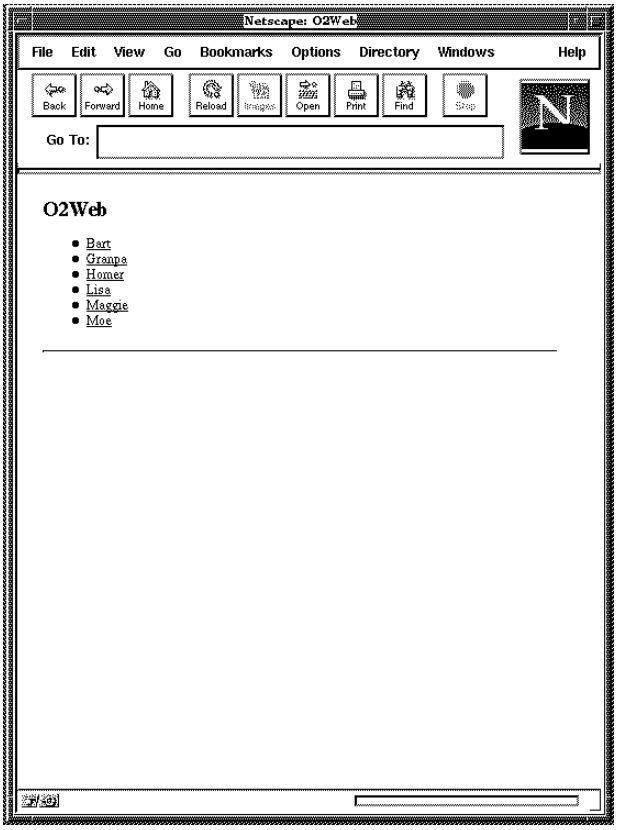


Figure 5.3: The Entries class

The **`Entry`** class contains an entry in a directory. An entry has a name, a photo, an address, a phone number, an e-mail address, and three attributes (**`previous`**, **`next`** and **`up`**) which respectively refer to the previous entry in the directory, the next entry in the directory and the directory itself. These attributes will be used to browse through a directory.

```
class Entry {
      public:
            char* name;
            d_Ref<Picture> photo;
            char* address;
            char* phone;
            char* e_mail;
            d_Ref<Entry> previous;
            d_Ref<Entry> next;
            d_Ref<Directory> up;

            d_String html_title();

            Entry();
            Entry(char* name,
            d_ref<Image> & photo,
            char* address,
            char* phone,
            char* e_mail);
            ~Entry();
            Entry & operator = (const Entry & ent);

};
```

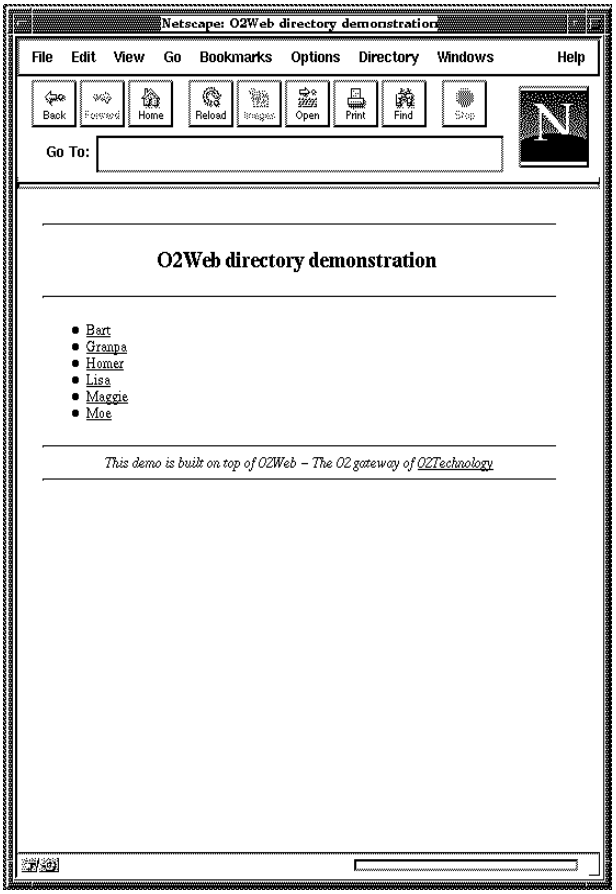


Figure 5.4: The Entry class

The **Picture** class is a class that manages GIF images. It is defined as follows:

```
class Picture {
private:
      d_Bits bits;
      int width;
      int height;
      char *name;
public:

      Picture();
      Picture(char *s);
      ~Picture();
      Picture & operator = (const Picture & p);

      int get_width();
      int get_height();
      char *get_name();
      int load(char* file, int w, int h);

      d_String html_title ();
      d_Bits html_prolog (char *query, char *userdata);
      d_Bits html_header (char *query, char *userdata);
      d_Bits html_footer (char *query, char *userdata);
      d_Bits html_epilog (char *query, char *userdata);
      d_Bits html_report (char *query, char *userdata);
};
```

Details of the member function bodies for this schema are not given here. You can consult these member functions in the **opt/o2web/samples/cplusplus/step1** directory of the O_2 distribution.

O2Web server main

The **main** function performs the following:

- Creates a **o2_Web** class object.
- Reads the default environment (**set_default_env**).
- Gets the command line parameters.
- Initializes o2_Web (**begin**).
- Starts o2_Web (**init**).

- Starts a main loop (**loop**).
- Finishes (**end**).

The main loop waits for requests sent by the web client. Each request is analyzed. The OQL query is processed and an HTML page is generated. The **begin**, **loop**, **init**, and **end** functions are in the **o2web_server** library.

When started, the **o2web_server** establishes a connection with an O_2 Web dispatcher (**o2open_dispatcher**), which must already be running and establishes a connection with a named O_2 database system using **o2server**, which must already be running.

The main function can be found in **$O2HOME/samples/o2web/cplusplus/step1/main.cc**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "o2web_CC.hxx"
int main(int argc, char** argv)
{
        short  i, error=0;
        o2_Web *o2web = new o2_Web();

        o2web->set_default_env();
        error = o2web->begin(argc, argv);
        if (error) {
                return(1);
        }
        o2web->init();
        o2web->loop();
        o2web->end();

        delete o2web;
        return (0);
```

Building your O2Web Server

Now that the application is written, we have to perform the following six steps:

1 Initialize a system and run **o2server**.

2 Create an O_2 schema.

3 Import the C++ classes into O_2 and create the Web server.

4 Create persistent roots.

5 Populate the database.

6 Test your server.

Initializing a system and running o2server

```
> $O2HOME/bin/o2dba_init -system your_system
> $O2HOME/bin/o2server -system your_system
```

Creating an O_2 Schema

```
> $O2HOME/bin/o2dsa -system your_system
Type your command with ^D
# "load_schema.o2"
^D
Type your command with ^D


with load_schema.o2:
      schema step1_schema;
      base step1_base;
      commit;
```

Importing classes and creating the o2web_server executable

After schema creation, you import the classes and member functions of your application. Then, you build your o2web_server executable. The configuration file of the sample application (step1_webserver.cf) can be used to generate the appropriate makefile.

```
> $O2HOME/bin/o2makegen step1_webserver.cf
> make
```

This imports the classes and creates an executable named **step1_webserver**.

Creating persistent roots

```
> $O2HOME/bin/o2dsa_shell -system your_system
Type your command with ^D
# "load_names.o2"
^D
Type your command with ^D
```

Populating the database

You need to populate your new database before you can test the Web server. A small ODMG C++ application has been written for this purpose. This application is located at **$O2HOME/samples/o2web/cplusplus/step1/init_data.cc**, with the corresponding configuration file at **$O2HOME/samples/o2web/cplusplus/step1/step1_init_data.cf**. To build this application:

```
> $O2HOME/bin/o2makegen
$O2HOME/samples/o2web/cplusplus/step1/step1_init_data.cf
> make
```

To run this application, type:

```
> step1_init_data -system your_system -server your_server
```

Testing the server

You can now run your O_2Web server by typing:

```
> step1_webserver -system your_system -server your_server
```

Note

In order to run successfully, ensure that:

- An HTTP server is running.
- An o2open_dispatcher is running.
- The o2web_gateway is properly installed.

You are now ready to browse through the phone book. You can see that the result is a good starting point to help you evaluate your final service.

5.4 Global Personalizations

The generic mode of O_2Web can be considered as a great prototyping tool in the process of building a large scale Web service. However, there is a need for more sophisticated mechanisms in order to build a personalized Web service. This section explains how to globally personalize parts of a generic service.

The global personalizations are handled by means of a named object whose name is **`TheO2WebInteractor`**. This object must belong to the **`O2WebInteractor`** class or one of its subclasses. Global personalizations are achieved by defining member functions in the **`O2WebInteractor`** class. O_2Web, in the process of generating HTML, will call these member functions on the **`TheO2WebInteractor`** root if they are defined. In order to be actually used by O_2Web, the class **`O2WebInteractor`** and its member functions must be imported in O_2.

Adding a header to the top of each page

To add a constant header to the top of each page, the programmer must define a member function called **`header`** on **`TheO2WebInteractor`** root. O_2Web checks for this member function definition before calling the generic HTML generation. If it is defined the result is inserted before the standard HTML.

Adding a footer to the bottom of each page

To add a constant footer to the bottom of each page, the programmer must define a member function called **`footer`** on **`TheO2WebInteractor`** the root. O_2Web checks for that member function definition before calling the generic HTML generation. If this is defined the results are inserted before the standard HTML.

Changing the default prolog and epilog

The default prolog and epilog can be modified. This is usually unnecessary to do so because the default prolog contains the MIME type for the returned document and should only be changed globally for a class returning a specific MIME type (a class that handles postscript documents for instance). However, this member function can be overloaded just in case the HTML prolog format changes in future versions of HTML.

React to a connection

When an O_2Web server is contacted to answer a query, the existence of the **connect** member function at the **TheO2WebInteractor** root is verified. If it exists, **connect** is called to allow the programmer to perform certain actions. The member function **connect** must be defined as follows:

```
char connect(char* query, char* userdata);
```

where

- **query** is the query given to the O_2Web server and for which a HTML report has to be built.
- **userdata** is a string a programmer can store in a URL to be returned when certain member functions are called. It allows the programmer to store context dependent information when a URL is created, and to enable specific processing according to this information when the URL is resolved (for more information about **userdata**, see the **make_url** member function in the o2webassistant library).

If this member function returns **true**, the connection is accepted. A return value of **false** will reject the connection.

React to a disconnection

After generating HTML text in response to a user query, O_2Web checks for the existence of the **disconnect** member function on the **TheO2WebInteractor** root. If it exists, **disconnect** is called allowing the programmer to perform certain logging actions. The member function **disconnect** must be defined as follows:

```
void disconnect(int report_size, char* report_kind);
```

where

- **report_size** is the number of bytes returned for that connection and
- **report_kind** is the MIME type of the returned text.

Make your own error messages

When an error occurs, an error message is returned to the Web browser. You may want to change these messages for your server or enhance them by adding images to the error message text.

This can be done by defining the **error** member function on the **TheO2WebInteractor** root. If defined, this member function must return the complete HTML text, including the content type, to the client. The signature of this member function is as follows:

```
d_Bits error(int k);
```

The **k** parameter is one of the possible error codes returned by O_2Web. These codes are defined in the file **o2web.h**.

A Guided Example

We will now modify our previous example in order to add text to the top and bottom of every page. We will also personalize the error messages. The extra-code for this example can be found in the **opt/o2web/samples/cplusplus/step2** directory of the O_2 distribution.

In all the next examples, we will use the **o2_WebStream** class which is provided in the **o2webassistant** library and which implements a number of character string manipulations.

We create the class **O2WebInteractor**. Three member functions are defined in the **O2WebInteractor** class. These member functions are automatically called by O_2Web at the beginning of HTML generation (**header**), at each ending of HTML generation (**footer**) and each time an error will occur (**error**). This class and its methods must be imported in O_2.

We also define a persistent root called **TheO2WebInteractor**; It is mandatory to create this name for the member functions of the class to be called.

Let us have a look at these three member functions.

The **header** member function returns the HTML text to be inserted before each new HTML generation.

This is a simple header containing formatted HTML text.

```
d_Bits O2WebInteractor::header()
{
      o2_WebStream st;

      st << "<html><header>\n";
      st << "<title> O2Web directory demonstration
</title>\n";
      st << "<body>\n";
      st << "<HR>\n<CENTER>\n";
      st << "<H2> O2Web directory demonstration </H2>\n";
      st << "</CENTER>\n<HR>\n";

      d_Bits str( st.data() );
      return( str );
}
```

In the **`footer`** member function, we build a text containing an address and insert an anchor pointing to an O_2Technology Web server.

```
d_Bits O2WebInteractor::footer()
{
      o2_WebStream st;

      st << "<HR>\n<ADDRESS>\n<CENTER>\n";
      st << "This demo is built on top of O2Web - The O2 gateway
of ";
      st << "<a href=http://www.o2tech.com> O2Technology
                                          </a></CENTER>\n";
      st << "<HR>\n</body></html>";

      d_Bits str( st.data() );
      return( str );
}
```

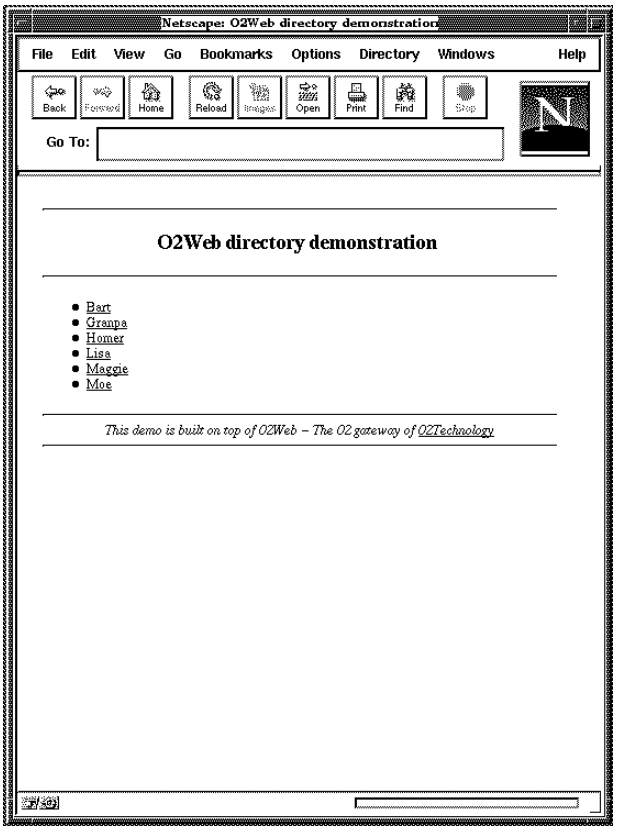


Figure 5.5: The global header and footer

The **error** member function is defined below. Notice the CGI header, containing the MIME type of the returned document, in the HTML text.

```
d_Bits O2WebInteractor::error( int k )
{
      o2_WebAssistant toolbox;
      o2_WebStream st;

      st << toolbox.get_http_prolog("text/html");
      st << "<HR>\n<CENTER>\n";
      st << "<H2>Hypertext Documentation Error</H2>\n";
      st << "</CENTER>\n<HR>\n";
      st << "<H3>\n An error has occurred, please try again.\n";
      st << "<p>If the error persists, contact Mr. Patch
(patch@rescue.com)</H3>";
      st << "<HR>\n";

      d_Bits str( st.data() );
      return( str );
}
```

5.5 Local Personalizations

Local personalizations allow a programmer to control all aspects of the HTML returned by a query. Local personalization involves the definition of a member function in a class that will be in charge of producing a part of the HTML.

With the exception of the events handling member functions (**`error`**, **`connect`** and **`disconnect`**), all the other member functions that can be defined in the **`O2WebInteractor`** class can be overloaded locally in any class of a schema.

In order to be useed by O_2Web, all these methods must be imported in O_2.

Adding a header to the top of a page

The programmer can decide that all queries received by an O_2Web server returning an object of a particular class must have a specific header that depends on the data contained in the object or that is well adapted to the meaning of that class.

This is done by defining a member function called **`html_header`** in a class of the schema. If there was a **`header`** member function defined in the **`O2WebInteractor`** class, it is not used for classes in which a specific **`html_header`** member function is defined.

Adding a footer to the bottom of a page

The programmer can decide that all queries received by an O_2Web server returning an object of a particular class must have a specific footer that depends on the data contained in the object or that is well adapted to the meaning of that class.

This is done by defining a member function called **`html_footer`** in a class of the schema. If there was a **`footer`** member function defined in the **`O2WebInteractor`** class, it is not used for classes in which a specific **`html_footer`** member function is defined.

Changing the prolog and epilog

Suppose that you have created a class that encapsulates data of a specific format (postscript document, JPEG image, MPEG movie, etc.).

You must inform the HTTP server that the data you send to it has a specific format. The member function **`html_prolog`** can be defined in this class and returns the MIME type corresponding to the format handled by the class.

For instance, a class managing MPEG movies will define a member function **`html_prolog`** returning the string:

```
Content-type: movie/mpeg\n\n
```

Having a specific member function to handle CGI headers allows a programmer to define a class managing any binary format and to create a subclass for each format. Thus, the only member function defined in subclasses is the **`html_prolog`** member function.

Building the body of a report

The body of a report can only be redefined locally in a class. The programmer can define a member function called **`html_report`** in a class and the result of this member function will be used instead of calling the generic HTML production.

Optimizing the query generation

When the programmer lets O_2Web generate queries for the sub-objects of an object, these queries are built starting from the query leading to the current object and adding to it a selection predicate or the selection of an attribute. In certain classes, the programmer may decide to generate the queries leading to the instances of the classes. This is achieved by defining a member function called **`get_query`**.

A guided example

In this section, we will improve our example in two phases: in the first phase, we optimize the generation of queries of the generic mode. In the second phase, we completely customize the HTML generation.

Optimizing the query generation

In the two previous examples (generic mode and global personalization), the query generated and inserted in the anchors grew in size each time a user browsed. A query inserted in a link was built automatically by O_2Web, starting from the current query and adding a selection predicate or an attribute name to it. This can lead O_2Web to generate very large queries.

This problem can be avoided if the programmer knows how to directly reach objects of a class from the persistent roots of the schema. In this case, we define **get_query** member functions in the classes **Directories**, **Directory** and **Entry**. When **get_query** is applied to an object, the text of an OQL query returning the object is returned. O_2Web uses this query instead of generating a query.

The code for these member functions can be found in the **$O2HOME/samples/o2web/cplusplus/step3** directory of the O_2 distribution.

```
d_String Directories::get_query ()
{
      d_String str( "DIRECTORIES" );
      return( str );
};

d_String Directory::get_query ()
{
      o2_WebStream st;

      st << "element("
            << "select d"
                  << "from d in DIRECTORIES.directories "
             << "where d->html_title = \""
                  << html_title () << "\")";
      d_String str( st.data() ); );
      return( str );
};

d_String Entry::get_query ()
{
      o2_WebStream st;

      st << "first(select e "
            << "from e in " << up->get_query()
                  << "->entries.Entries_value "
                        << where e->html_title = \""
                              << html_title + "\")";

      d_String str( st.data() ); );
      return( str );
};
```

After defining these three member functions, you can now browse a document and see that the queries contained in the HTML anchors no longer grow when you click on the links.

A complete customization

The code of the complete customization can be found in the **`$O2HOME/samples/o2web/cplusplus/step3`** directory of the O_2 distribution.

Class Directories

We define the **`html_header`** member function in the **`Directories`** class in order to personalize the header when retrieving objects of this class.

```
d_Bits Directories::html_header(char * query, char *
userdata)
{
      o2_WebStream st;

      st << "<html> <header>\n<title>";
      st << "directories\n";
      st << "</title>\n</header>";

      st << "<body>\n<CENTER><H2>\n";
      st << "The Available Directories \n";
      st << "</H2></CENTER>\n<HR>";

      d_Bits str( st.data() );
      return( str );
}
```

Class Directory

We now add two member functions to the **`Directory`** class. The first one (**`html_header`**) is used to personalize the header of objects of the **`Directory`** class.

```
d_Bits Directory::html_header(char * query, char *
userdata)
{
      o2_WebStream st;

      st << "<html> <header>\n<title>";
      st << html_title();
      st << "</title>\n</header>";

      st << "<body>\n<CENTER><H2>\n";
      st << html_title();
      st << "</H2></CENTER>\n<HR>";

      d_Bits str( st.data() );
      return( str );
}
```

The second one (**`html_report`**) is used to personalize HTML generation. We change the standard HTML generation and give the user the choice whether to browse the directory or search for an entry in the directory.

The first choice is associated with a URL containing a query leading to the **`entries`** field of the class. Notice that the URL is created using the **`make_url`** member function in the **`o2_WebAssistant`** class. The **`userdata`** parameter specifies the name of the directory. This is returned to the **`html_header`**, **`html_footer`** and **`html_report`** member functions which are called when a user clicks on the created anchor. The **`userdata`** parameter is detailed in the **`make_url`** member function of the o2webwassistant library. This is a means of storing a context in the returned HTML and retrieving it when a user clicks on an anchor. This could have been achieved using other techniques such as hidden fields or cookies.

The second choice is associated with a URL leading to an object called **`TheDirectorySearcher`**. This object belongs to a new class

DirectorySearch. This class handles the creation of an HTML form and the retrieval of user information. This class will be discussed below.

```
d_Bits Directory::html_report(char * query, char * userdata)
{
      o2_WebAssistant toolbox;
      o2_WebStream st, tmp;

      tmp << get_query() << "->entries";

      st << "<dl>\n";

      st << "<dt> <h3>";
      st <<
toolbox.make_anchor(toolbox.make_url(tmp.data(),"",(const
                            char*)html_title(), 0),
                            "Browse the Directory");
      st << "</h3>\n";
      st << "<dd>\n";
      st << "Click above to consult the directory by navigating
inside
                            it\n";

      st << "<dt> <h3>";
      st <<
toolbox.make_anchor(toolbox.make_url("TheDirectorySearcher",
                            "",(const char*)html_title(), 0),
                            "Search the Directory");
      st << "</h3>\n";
      st << "<dd>\n";
      st << "Click above to search a specify entry in the
directory\n";

      st << "</dl>\n";

      d_Bits str( st.data() );
      return( str );
}
```

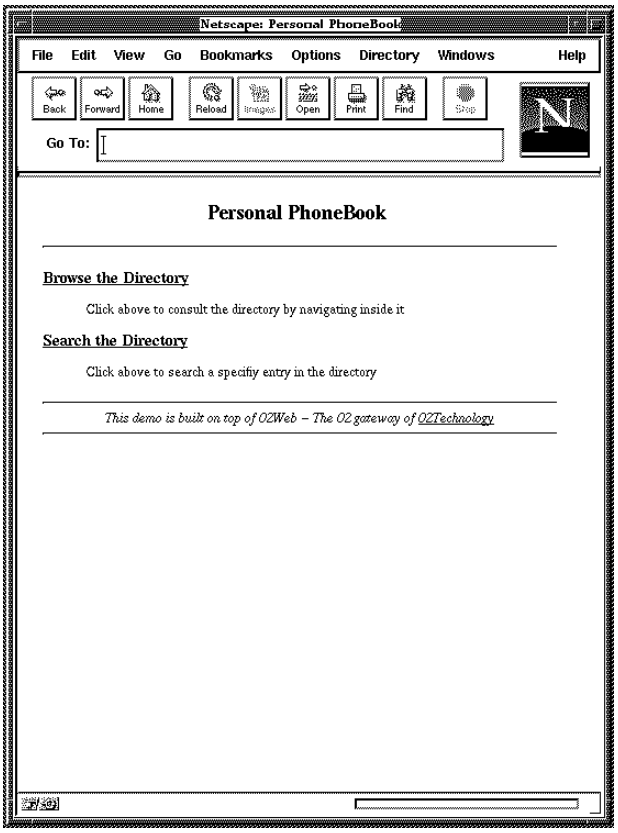


Figure 5.6: The customized Directory class

Class Entries

We personalized the header for an object of the class **`Entries`**. You may be wondering why we chose to define a class to manage the entries in a directory rather than defining an attribute of the **`Directory`** class to contain the list of entries. This choice was made to enable the personalization of the pages containing the list of entries.

Notice that the **`userdata`** parameter is used in the member function below. It contains a string, given by the programmer in the **`html_report`** member function of the **`Directory`** class when the anchor pointing to an object of the **`Entries`** class was created. This string contains the name of the directory.

```
d_Bits Entries::html_header(char * query, char *
userdata)
{
      o2_WebStream st;

      st << "<html> <header>\n<title>";
      st << userdata << " entries";
      st << "</title>\n</header>";

      st << "<body>\n<CENTER><H2>\n";
      st << "Entries of the " << userdata << "
Directory\n";
      st << "</H2></CENTER>\n<HR>";

      d_Bits str( st.data() );
      return( str );
}
```

Class Entry

We define two member functions in the **`Entry`** class. First, we add a specific header. Previously, the **`previous`**, **`next`** and **`up`** attributes of the class **`Entry`** were part of the report and appeared as anchors. We now want to remove these attributes from the report and add a navigation bar to the header defined in this class.

The navigation bar provides direct access to the previous and next entries of the current phone book as well as access to the phone book (**`up`**) and the list of phone books (**`top`**).

```
d_Bits Entry::html_header(char * query, char * userdata)
{
      o2_WebStream st;
      o2_WebAssistant toolbox;
      d_Ref<Directories> dir("DIRECTORIES");

      st << "<html> <header>\n<title>";
      st << name << " Directory";
      st << "</title>\n</header>";

      st << "<body>\n<CENTER><H2>\n";
      st << name << " Directory";
      st << " </H2></CENTER>\n<HR>";

      if(dir != NULL ) {
            st <<
toolbox.make_anchor(toolbox.make_url("DIRECTORIES",
                                    "","", 0), "[top] " );
      }
      st << " ";
      if( up != NULL ) {
            st << toolbox.make_anchor(toolbox.make_url
                        (up->get_query(), "","", 0), "[up] ");
      }
      st << " ";
      if( previous != NULL ) {
            st << toolbox.make_anchor(toolbox.make_url
                        (previous->get_query(), "","", 0),
                              "[previous] ");
      }
      st << " ";
      if( next != NULL ) {
            st << toolbox.make_anchor(toolbox.make_url
                        (next->get_query(),"","", 0), "[next] ");
      }
      st << "<hr>";

      d_Bits str( st.data() );
      return( str );
}
```

Now we define a specific report member function that formats an entry in the directory. Note the use of the **`<table>`** tag in the code below. Tables are part of the HTML 3.0 specification and are not recognized by all browsers.

```
d_Bits Entry::html_report(char *, char *)
{
      o2_WebStream st, tmp;
      o2_WebAssistant toolbox;
      o2_WebImageAttributes * format = new o2_WebImageAttributes;

      tmp << get_query() << ".photo";

      st << "<center>\n";
      st << "<table border=5 cellpadding=5>\n";

      st << "<tr align=center>\n";
      st << "<td><h3>" << name << "</h3></td>";

      st << "<tr align=center>\n";
      st << "<td>";
      st << toolbox.make_inline_image(tmp.data(),0, 0, format, 0,
                                      "Entry Photo");
      st << "</td></table>\n";
      st << "</center><p>\n";
      st << "<p>Address : " << address;
      st << "<p>Phone   : " << phone;
      st << "<p>Email   : <a href=\"mailto:" << e_mail << " \"> "
                                      << e_mail << "</a>\n";

      d_Bits str( st.data() );
      return( str );
}
```

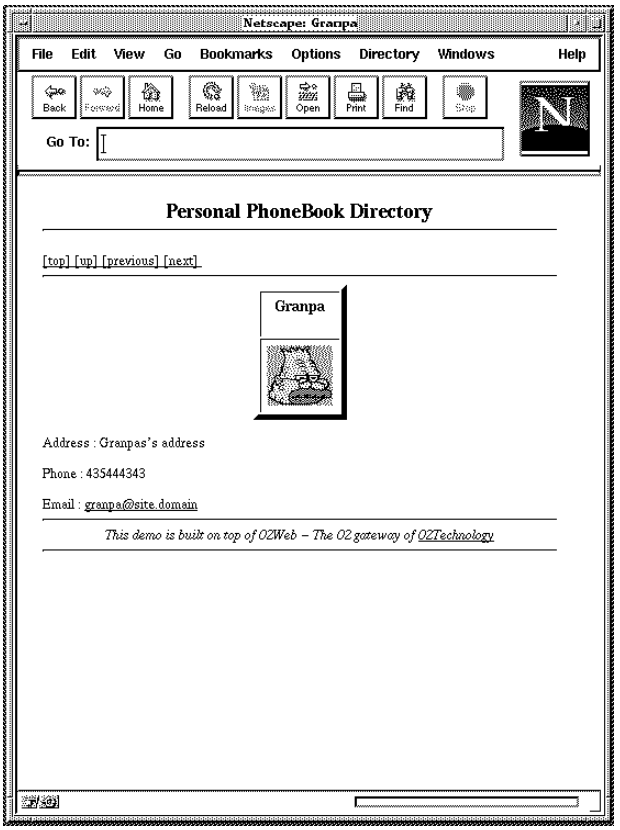


Figure 5.7: The customized Entry class

Class DirectorySearch

We saw in the **`html_report`** member function of the **`Directory`** class that the user has the choice of browsing the directory or searching for a specific entry in the directory. The following class, called **`DirectorySearch`**, will be used to display an HTML form that gives the user an interface for searching a directory. The creation of the form is handled in the **`html_report`** member function and the analysis of the user answer is handled by the **`search`** member function. A named object **`TheDirectorySearcher`** is created for this class.

```
class DirectorySearch{
public:


     d_Bits html_header(char* query, char* userdata);
     d_Bits html_report(char* query, char* userdata);
     d_String search(char* params);

     DirectorySearch(){};
     ~DirectorySearch(){};
};
```

The **`html_header`** member function is used to personalize the header for objects of this class.

```
d_Bits DirectorySearch::html_header(char * query, char * userdata)
{
     o2_WebStream st;

     st << "<html> <header>\n<title>";
     st << userdata << " search";
     st << "</title>\n</header>";

     st << "<body>\n<CENTER><H2>\n";
     st << "Search the " << userdata << " Directory";
     st << "</H2></CENTER>\n<HR>";

     d_Bits str( st.data() );
     return( str );
}
```

The **`html_report`** member function creates an HTML form with two fields: the first field is used by the user to enter a name to search for in the directory; the second field is a hidden field (invisible on the screen) that retains the name of the directory, which was searched, in the HTML produced.

The action of the form is defined as:

TheDirectorySearcher->search($0)

This means that when a form is submitted, the member function **search** is triggered on the **TheDirectorySearcher** root. The values of the form fields are given to the **search** member function by substituting the string $0 with the result of the form.

```
d_Bits DirectorySearch::html_report(char * query, char * userdata)
{
      o2_WebStream st;
      o2_WebAssistant toolbox;

      st << "<form method=post action=";
      st << toolbox.make_url("TheDirectorySearcher->search($0)",
                             "", "", 0);
      st << ">\n";
      st << "Enter a name : <input name=\"pattern\" value=\"\">\n";
      st << "<input type=hidden name=\"directory\" value=\"";
      st << userdata << "\"></form>";

      d_Bits str( st.data() );
      return( str );
}
```

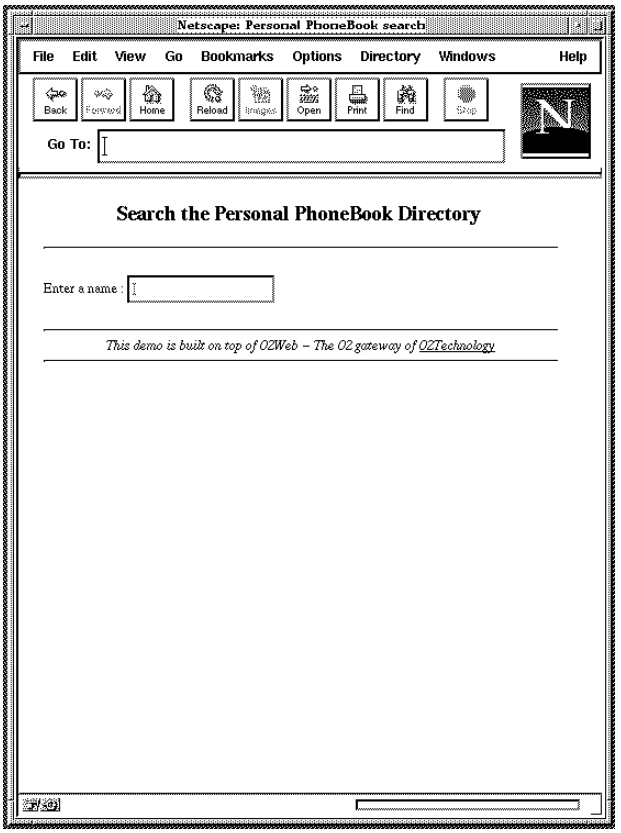


Figure 5.8: An interface for searching

The **`search`** member function retrieves the form result using the **`o2_WebFormAnalyser`** class of the o2webassistant library. Then it searches for the entry and displays the result. Notice, as with the **`error`** member function of the class **`O2WebInteractor`**, a member function called as an action of a form must return a complete HTML text, including the CGI header. This is why the **`search`** member function creates a return value containing a call to **`get_http_prolog`** when the entry does not exist in the directory. When the entry is found, the result of the member function is the result of the call to the **`make_report`** member function of the **`o2_WebAssistant`** class. To insert the CGI header, the **`make_report`** member function is called with the **`RP_WITH_HEADER`** parameter.

```
d_Bits DirectorySearch::search( char * params )
{
   int i;
   unsigned long nb_entries;
   d_Iterator< d_Ref<Directory> > iter;
   char * name;
   char * dir_name;
   o2_WebAssistant toolbox;
   o2_WebStream st;
   o2_WebFormAnalyser formtool( params );
   d_Array<o2_WebFormItem> formitems;
   d_Array<o2_WebFormItem> dirformitems;
   d_Ref<Directories> dir("DIRECTORIES");
   d_Ref<Directory> directory;
   d_Ref<Entry> entry;

   formtool.get_values("pattern", formitems);
   if( (formitems[0].get_value()) != NULL ) {
      name = new char[strlen(formitems[0].get_value())+1];
      strcpy(name, formitems[0].get_value());
   }

   formtool.get_values( "directory", dirformitems );
   if( (dirformitems[0].get_value())  != NULL ) {
      dir_name = new
char[strlen(dirformitems[0].get_value())+1];
      strcpy(dir_name, dirformitems[0].get_value());
   }

   iter = dir->directories.create_iterator();
   while(iter.not_done()) {
      if(strcmp(iter.get_element()->html_title(), dir_name ) ==
0 ){
       directory = iter.get_element();
       break;
      }
      iter++;
   }
```

Continued on the following page.

```
   if (directory != NULL) {
     i = 0;
     nb_entries = directory->entries->Entries_value.cardinality();
     while(i<nb_entries) {
      if(strcmp(
directory->entries->Entries_value[i]->html_title(),
              name ) == 0 ) {
         entry = directory->entries->Entries_value[i];
         break;
      }
      i++;
     }
   }

   if( entry == NULL ) {
     st << toolbox.get_http_prolog("text/html");
     st << "<H3> There is no entry for ";
     st << name << " in " << dir_name;
     st << "</H3>";
   } else {
     st << toolbox.make_report(entry->o2_get_handle(),
                               entry->get_query(),
                               RP_DEFAULT,
                               RP_WITH_HEADER,
                               "" );
   }

   d_Bits str( st.data() );
   return( str );
}
```

5.6 Updating the database with O2Web

There are many occasions where a programmer might wish to update the database (log a connection in the database, store the result of an HTML form, etc.).

As O_2Web runs in program mode, a member function can update persistent objects of the database or create new persistent objects only if it starts a new transaction. It is the responsibility of the programmer to end the transaction after the updates in order to leave O_2Web in read-only transaction mode.

In O_2, a transaction is ended using either the **`validate`** or the **`commit`** member functions of the d_Transaction class. In O_2Web, a transaction can only be ended using **`validate`**. If a member function called by O_2Web performs a **`commit`**, the current request is discarded and the Web client receives an error message.

Suppose you want to log information concerning a connection in the database; you can define the **`disconnect`** member function of the **`O2WebInteractor`** class as follows:

```
void O2WebInteractor::disconnect(int report_size,
                                 char * report_kind)
     {
          TheConnections->add(report_size, report_kind);
     }
```

It is up to the programmer to start a new transaction in order to update the **`TheConnections`** persistent object. This must be done as follows:

```
void Connections::add(int size, char * kind)
     {
          d_Transaction trans;
          Connection conn = new Connection;

          conn->size = size;
          conn->kind = kind;

          trans.begin();
          this->insert_element_list(conn);
          trans.validate();
     }
```

Furthermore, as the member function above might produce a deadlock when more than one O_2Web server is accessed at the same time to answer requests, an explicit lock must be set on the persistent object before manipulating it in order to avoid deadlocks. Thus, the member function should be written as follows:

```
void Connections::add(int size, char * kind)
     {
          d_Transaction trans;
          Connection conn = new Connection;
          conn->size = size;
          conn->kind = kind;
          locks[0] = conn; locks[1] = 0;
          trans.begin(locks);
          this->insert_element_list(conn);
          trans.validate;
     }
```

For further information concerning transactions and deadlocks, please refer to the O_2 manuals.

5.7 Summary

In this section, we summarize the different techniques used to create an HTML text for a query submitted to an O_2Web server.

The HTML text produced for any query is made of five parts:

1 prolog - a protocol specific text

2 header - a constant page header

3 body - the body of a report

4 footer - a constant page footer

5 epilog - a protocol specific text

All these document parts have a generic implementation that can be redefined by the programmer.

The prolog, header, footer and epilog sections can be overloaded globally in the **`O2WebInteractor`** class by means of the **`prolog`**, **`header`**, **`footer`** and **`epilog`** member functions and overloaded locally in any class using the **`html_prolog`**, **`html_header`**, **`html_footer`** and **`html_epilog`** member functions.

The body of a report can only be overloaded locally by means of the **`html_report`** member function.

The generic implementation can also be improved by locally defining the **`get_query`** member functions in your classes.

Besides HTML production personalization, the programmer can personalize the reaction to certain events that might occur. These events are a connection, a disconnection and an error. The handlers associated with these events are member functions defined in the **`O2WebInteractor`** class.

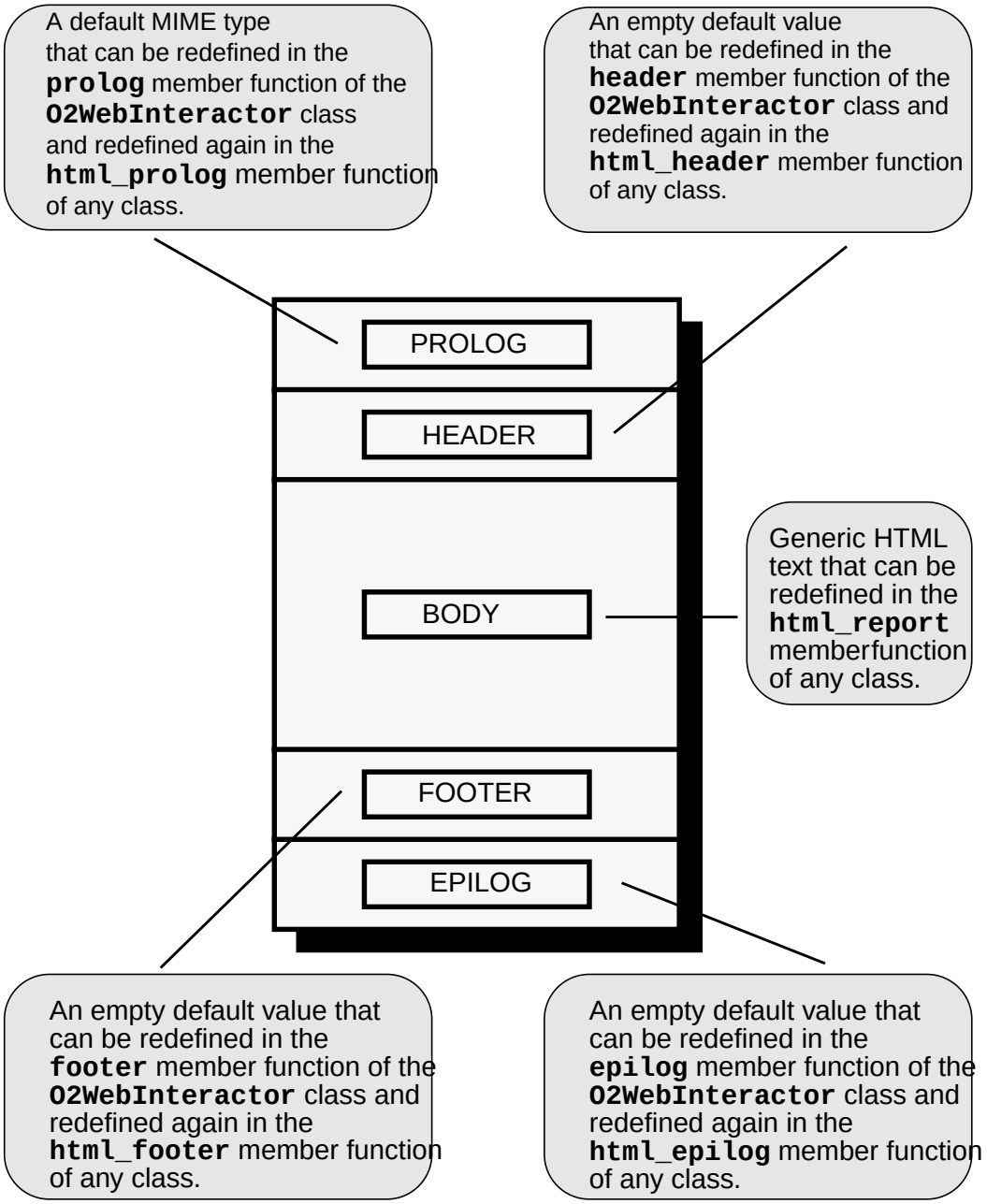


Figure 5.9: HTML production

6 O2Web Reference

This chapter gives the full referential information for O_2Web.

It is divided into the following sections:

- `O2WebInteractor` - Programmers can define some member functions in this class in order to overload O2Web behavior for all classes of a schema that do not have their own behavior. This class is also used by programmers to provide O2Web with member functions to call when some events occur (connection, disconnection, etc.).
- `User-Defined member functions` - Programmers can define member functions for each class of a schema, to overload the default behavior or the global redefinition of the O2WebInteractor class.
- `The o2webassistant library` - This is composed of several classes and member functions whose aim is to help the programmer in the process of redefining the generic HTML production.
- `The o2webassistant library` - This class is used by the programmer in the main of the application to start an O2Web server and begin the server loop.
- `O2Web Commands` - This section outlines the O2Web system commands.

6 O_2WEB REFERENCE

6.1 O2WebInteractor

`O2WebInteractor` is a class that can be created by the programmer. Some member functions can be defined in this class to globally change parts of the generic HTML production.

Other member functions can be defined to handle certain events such as connection, disconnection and error.

Member functions of the **`O2WebInteractor`** class will be called only if a persistent root called **`TheO2WebInteractor`** is defined in this class.

The rest of this section describes the following member functions:

- **`connect`**
- **`disconnect`**
- **`epilog`**
- **`error`**
- **`footer`**
- **`header`**
- **`prolog`**

connect

Summary A user defined member function called once for each connection.

Syntax `char connect(char* query, char* userdata);`

Arguments **`query`** The query to be submitted to the O_2Web server.

`userdata` A string that the programmer might have inserted in the anchor that produced the query. If this is the case, this string is returned to the programmer (for more information about **`userdata`**, see the **`make_url`** member function of the o2webassistant library).

Description This member function is called once for each connection if the **`TheO2WebInteractor`** persistent root is defined. There is no default implementation for this member function. It can be defined for desired tasks such as authorization checking.

Returns A char.

A value of 0 causes O_2Web to reject the connection. O_2Web will try to trigger the **`error`** member function on the **`TheO2WebInteractor`** root. The argument for this member function will be **`O2WEB_PROTECTION`**.

A non-zero value authorizes the connection.

disconnect

Summary A user-defined member function called once after each connection.

Syntax `void disconnect(int report_size, char* report_kind);`

Arguments `report_size` The size of the generated document.

`report_kind` The MIME type of the returned document.

Description This member function is called once after each connection if the `TheO2WebInteractor` persistent root is defined. There is no default implementation for this member function. It can be defined for desired tasks such as statistical analysis.

Returns Nothing.

epilog

Summary Changes the default epilog.

Syntax `d_Bits epilog();`

Arguments None.

Description This member function is called each time HTML is generated if the **`TheO2WebInteractor`** persistent root is defined and if the **`html_epilog`** member function has not been defined in the class to which belongs the object computed by the URL query. There is no default implementation for this member function.

Returns A d_Bits value.

error

Summary Redefines the error message.

Syntax `d_Bits error(int k)`

Arguments **k** An error code.

Description This allows the programmer to redefine the error message that will be returned to the Web client. It is called when an error occurs if the **`TheO2WebInteractor`** persistent root is defined. The possible values of **k** are:

- **`O2WEB_WRONG_BASE_NAME`**
- **`O2WEB_EMPTY_QUERY`**
- **`O2WEB_RUNTIME_ERROR`**
- **`O2WEB_COMMIT`**
- **`O2WEB_ABORT`**
- **`O2WEB_BAD_FORM`**
- **`O2WEB_PROTECTION`**

Some of the above errors can occur due to a problem in the HTTP configuration file.

Returns A d_Bits value that contains the HTML text, including the CGI header, to be sent to a Web client when an error occurs.

Example When an error occurs during O_2Web activity, default error messages are sent to the Web client. The following example shows how a programmer can define his own error messages.

```
d_Bits O2WebInteractor::error( int k )
{
   o2_WebAssistant toolbox;
   o2_WebStream st;

   st << toolbox.get_http_prolog("text/html");
   st << "<HR>\n<CENTER>\n";
   st << "<H2>Hypertext Documentation Error</H2>\n";
   st << "</CENTER>\n<HR>\n";
   st << "<H3>\n An error has occured, please try again.\n";
   st << "<p>If the error persists, contact Mr Patch
(patch@rescue.com)</H3>";
   st << "<HR>\n";

   d_Bits str( st.data() );
   return( str );
}
```

footer

Summary Adds a constant footer to the bottom of each page.

Syntax `d_Bits footer();`

Arguments None.

Description This member function adds a constant footer to the bottom of each page. It is called after each time HTML is produced if the **`TheO2WebInteractor`** persistent root is defined and if a member function **`html_footer`** has not been defined in the class to which belongs the object computed by the URL query.

There is no default implementation for this member function.

This member function is used to add constant elements to the bottom of each page returned by O_2Web.

Returns A d_Bits value containing a piece of HTML text to be inserted at the bottom of each page.

Example A simple footer can be written as follows:

```
d_Bits O2WebInteractor::footer()
{
   o2_WebStream st;

   st << "<HR>\n<ADDRESS>\n<CENTER>\n";
   st << "This demo is built on top of O2Web - The O2 gateway
of ";
   st << "<a href=http://www.o2tech.com>
O2Technology</a></CENTER>\n";
   st << "<HR>\n</body></html>";

   d_Bits str( st.data() );
   return( str );
}
```

header

Summary Adds a constant header to the top of each page.

Syntax **`d_Bits header();`**

Arguments None.

Description This member function adds a constant header to the top of each page. It is called each time HTML is produced if the **`TheO2WebInteractor`** persistent root is defined and if a member function **`html_header`** has not been defined in the class to which belongs the object computed by the URL query.

There is no default implementation for this member function.

This member function is used to add constant elements to the top of each page returned by O_2Web.

Returns A d_Bits value containing a piece of HTML text to be inserted at the top of each page.

Example A simple header can be written as follows:

```
d_Bits O2WebInteractor::header()
{
   o2_WebStream st;

   st << "<html><header>\n";
   st << "<title> O2Web directory demonstration
</title>\n";
   st << "<body>\n";
   st << "<HR>\n<CENTER>\n";
   st << "<H2> O2Web directory demonstration
</H2>\n";
   st << "</CENTER>\n<HR>\n";

   d_Bits str( st.data() );
   return( str );
}
```

prolog

Summary Changes the default prolog.

Syntax `d_Bits prolog();`

Arguments None.

Description This member function is called each time HTML is generated if the **`TheO2WebInteractor`** persistent root is defined and if the **`html_prolog`** member function has not been defined in the class to which belongs the object computed by the URL query. The default implementation of this member function returns a CGI header describing the type of the returned document.

Returns A d_Bits value that must contain a valid CGI header.

6.2 User-Defined member functions

The previous section described how to globally overload part of the HTML production. This section explains how to locally redefine (for a class) the HTML generation.

A local redefinition is performed by defining member functions in a class. Each defined member function overloads a part of the HTML generation.

The member functions that a programmer can define are:

- `get_query`
- `html_epilog`
- `html_footer`
- `html_header`
- `html_prolog`
- `html_report`
- `html_title`

get_query

Summary A user-defined member function that must return a valid OQL query.

Syntax `d_String get_query();`

Arguments None.

Description When using the generic mode of O_2Web, hypertext links are created to allow a user to browse the sub-objects contained in an object. These links are HTML anchors that encapsulate a URL containing a query leading to a sub-object. Such a query is generated automatically by O_2Web starting from the current query and adding to it a selection predicate or an attribute name. This can produce, after many clicks, very large queries.

This problem can be avoided if the programmer knows how to directly reach objects of a class from the persistent roots. In this case, the programmer can define the **`get_query`** member function in this class. The member function must return a valid OQL query, which when executed will return its receiver.

Warning !

Even if the get_query member function is identical in a class and its sub-class, it must be redefined in the sub-class and modified to contain a cast to the sub-class.

Returns A d_String value.

Example In the following example, the string returned by the member function will be used by the O_2Web automatic generation as the query to be used to create an anchor leading to an object of the **`Directory`** class.

```
d_String Directory::get_query ()
{
    o2_WebStream st;

    st << "element("
        << "select d "
        << "from d in DIRECTORIES.directories "
        << "where d->html_title = \"" << html_title() << "\")";

    d_String str( st.data() );
    return( str );
}
```

html_epilog

Summary Specifies the epilog for a class.

Syntax `d_Bits html_epilog(char* query, char* userdata);`

Arguments **`query`** The query to be submitted to the O_2Web server.

`userdata` A string that the programmer might have inserted in the anchor that produced the query. If this is the case, this string is returned to the programmer (for more information about **`userdata`**, see the **`make_url`** member function of the o2webassistant library).

Description This member function is called at the end of each new HTML production if it is defined on the class to which belongs the object computed by the URL query. In that case, it overloads the eventually defined **`epilog`** member function of the **`O2WebInteractor`** class. The result of this member function is inserted after the text returned by an **`html_footer`** member function defined in the same class or the **`footer`** member function of the **`O2WebInteractor`** class.

This member function is rarely useful.

Returns A d_Bits value.

html_footer

Summary Adds class-dependent elements to the bottom of each page.

Syntax `d_Bits html_footer(char* query, char* userdata);`

Arguments **`query`**The OQL query that returns the receiver of the member function. This can be guaranteed by O_2Web only if the **`html_report`** member function is called directly by O_2Web in response to a client query. If called directly by a programmer, it is its responsibility to provide the member function with a correct value for the query parameter.

`userdata`A string containing data stored in an anchor by the programmer when using the **`make_url`** member function of the **`o2_WebAssistant`** class. This parameter is only meaningful when **`html_report`** is called by O_2Web on the result of a client query.

Description This adds class-dependent elements to the bottom of each page resulting from a query leading to the class in which this member function is defined.

This member function can be defined in any class of an O_2 schema. It is called, at the end of each new HTML production, if it is defined in the class to which the object computed by the URL query belongs. In this case, it overloads the eventually defined **`footer`** member function of the **`O2WebInteractor`** class.

Returns A d_Bits value containing valid HTML.

html_header

Summary Adds class-dependent elements to the top of each page.

Syntax `d_Bits html_header(char* query, char* userdata);`

Arguments **`query`** The OQL query that returns the receiver of the member function. This can be guaranteed by O_2Web only if the **`html_report`** member function is called directly by O_2Web in response to a client query. If called directly by a programmer, it is its responsibility to provide the member function with a correct value for the query parameter.

`userdata` A string containing data stored in an anchor by the programmer when using the **`make_url`** member function of the **`o2_WebAssistant`** class. This parameter is only meaningful when **`html_report`** is called by O_2Web on the result of a client query.

Description This member function is used to add class-dependent elements to the top of each page resulting from a query leading to the class in which this member function is defined.

This member function can be defined in any class of an O_2 schema. It is called, at the beginning of each new HTML production, if it is defined in the class to which the object computed by the URL query belongs. In this case, it overloads the eventually defined **`header`** member function of the **`O2WebInteractor`** class.

Returns A d_Bits value containing valid HTML.

Example The following example defines a simple header in a class **`Directory`**.

```
d_Bits Directory::html_header(char * query,
                                        char *
userdata)
{
   o2_WebStream st;

   st << "<html> <header>\n<title>";
   st << html_title();
   st << "</title>\n</header>";

   st << "<body>\n<CENTER><H2>\n";
   st << html_title();
   st << "</H2></CENTER>\n<HR>";

   d_Bits str( st.data() );
   return( str );
}
```

html_prolog

Summary Specifies the prolog for a class.

Syntax `d_Bits html_prolog(char* query, char* userdata);`

Arguments **`query`** The query to be submitted to the O_2Web server.

`userdata` A string that the programmer might have inserted in the anchor that produced the query. If this is the case, this string is returned to the programmer (for more information about **`userdata`**, see the **`make_url`** member function of the o2webassistant library).

Description This member function is called at the beginning of each new HTML production if it is defined in the class to which belongs the object computed by the URL query. When called, it replaces the default CGI header or the one returned by the **`prolog`** member function in the **`O2WebInteractor`** class (if this has been defined).

Returns A d_Bits value containing a valid CGI header string.

html_report

Summary Replaces the default HTML generation.

Syntax `d_Bits html_report(char* query, char* userdata);`

Arguments **`query`**The OQL query that returns the receiver of this member function. This can be guaranteed by O_2Web only if the **`html_report`** member function is called directly by O_2Web in response to a client query. If called directly by a programmer, it is its responsibility to provide the member function with a correct value for the query parameter.

`userdata`A string containing data stored in an anchor by the programmer when using the **`make_url`** member function of the **`o2_WebAssistant`** class. This parameter is only meaningful when **`html_report`** is called by O_2Web on the result of a client query.

Description This replaces the default HTML generation for the class in which this member function is defined.

Returns A d_Bits value containing valid HTML.

Example The following example shows how the body of a report can be customized by a programmer.

```
d_Bits Directory::html_report(char * query, char * userdata)
{
   o2_WebAssistant toolbox;
   o2_WebStream st, tmp;

   tmp << get_query() << "->entries";

   st << "<dl>\n";

   st << "<dt> <h3>";
   st << toolbox.make_anchor(toolbox.make_url(tmp.data(),
                     "",(const char*)html_title(), 0),
                     "Browse the Directory");
   st << "</h3>\n";
   st << "<dd>\n";
   st << "Click above to consult the directory by navigating inside
it\n";

   st << "<dt> <h3>";
   st <<
toolbox.make_anchor(toolbox.make_url("TheDirectorySearcher",

"", (const char*)html_title(), 0),

"Search the Directory");
   st << "</h3>\n";
   st << "<dd>\n";
   st << "Click above to search a specify entry in the directory\n";

   st << "</dl>\n";

   d_Bits str( st.data() );
   return( str );
}
```

html_title

Summary Returns the text that will appear as the anchor (in the generic mode) when the receiver is a sub-object.

Syntax `d_String html_title();`

Arguments None.

Description When using the generic mode of Web or the **`make_report`** member function of the **`o2_WebAssistant`** class, a sub-object is represented by an anchors on which users must click to enter the sub-object.

This member function returns the text of the anchor. If this method has not been defined, the name of the class is used as the anchor text.

Returns A d_String.

6.3 The o2webassistant library

The o2webassistant library is a set of classes designed to help programmers in the process of building a World Wide Web service on top of O_2.

The classes belonging to the o2webassistant library are:

- **o2_WebAssistant**
- **o2_WebFormAnalyser**
- **o2_WebFormItem**
- **o2_WebImageAttributes**
- **o2_WebImageInliner**
- **o2_WebStream**

These classes do not need to be imported in O_2.

o2_WebAssistant

This class is a general purpose class that contains member functions used by programmers to perform various actions (retrieving a CGI variable value, creating an anchor, calling the generic O_2Web HTML generation, etc.).

This subsection presents the **`o2_WebAssistant`** class and the following member functions:

- **`get_http_prolog`**
- **`get_http_variable`**
- **`make_anchor`**
- **`make_index`**
- **`make_inline_image`**
- **`make_report`**
- **`make_url`**

```
class o2_WebAssistant {
public:

  o2_WebStream get_http_prolog(const char* MimeType);
  o2_WebStream get_http_variable(const char* name);
  o2_WebStream make_url(const char* index, const
                      char* user_data, int key);
  o2_WebStream make_index(const char* name,
                        const char* content);
  o2_WebStream make_inline_image(const char* query,
                              int width, int_height,
                              o2_WebImageAttributes* format,
                              int key, const char* alphalabel);
  o2_WebStream make_anchor(const o2_WebStream &url, const char*
content);
  o2_WebStream make_anchor(const char* url, const char* content);
  o2_WebStream make_report(Handle obj, const char* query,
                         O2RpMode generic, O2RpStatus header,
                         const char* userdata);

  o2_WebAssistant(){};
  ~o2_WebAssistant(){};
};
```

get_http_prolog

Summary Builds the proper CGI header for a MIME Type.

Syntax `o2_WebStream get_http_prolog(const char* MimeType);`

Arguments `MimeType` The MIME type.

Description This builds the proper CGI header for a MIME Type.

Returns An object of the `o2_WebStream` class.

get_http_variable

Summary Retrieves the environment variable values of a HTTP server.

Syntax `o2_WebStream get_http_variable(const char* name);`

Description This retrieves the values of the environment variables given by the HTTP server to the CGI script. The possible variable names are:

- **SERVER_NAME**
- **SERVER_PORT**
- **SERVER_PROTOCOLE**
- **SERVER_SOFTWARE**
- **REQUEST_METHOD**
- **PATH_INFO**
- **PATH_TRANSLATED**
- **QUERY_STRING**
- **SCRIPT_NAME**
- **CONTENT_TYPE**
- **CONTENT_LENGTH**
- **HTTP_ACCEPT**
- **HTTP_USER_AGENT**
- **AUTH_TYPE**
- **REMOTE_HOST**
- **REMOTE_ADDR**
- **REMOTE_USER**
- **REMOTE_IDENT**

For further information concerning these variables, consult the documentation for your HTTP server.

Returns An object of the **o2_WebStream** class.

make_anchor

Summary Creates a bits value containing the definition of an HTML anchor.

Syntax

```
o2_WebStream make_anchor(const o2_WebStream &url,
                         const char* content);
```

Arguments

`url`	The URL of the resource.
`content`	A string that will appear in the generated anchor.

Description This creates a bits value containing the definition of an HTML anchor.

Returns An object of the **`o2_WebStream`** class.

make_index

Summary Creates a string that contains an HTML index.

Syntax

```
o2_WebStream make_index(const char* name,
                        const char* content);
```

Arguments

name	The name of the created string.
content	The content of the created string.

Description This creates a string that contains an HTML index. This index can be referred to when creating a URL in order to scroll through the retrieved document until the index becomes visible.

Returns An object of the **o2_WebStream** class.

make_inline_image

Summary Creates an inline image.

Syntax

```
o2_WebStream make_inline_image(const char* query,
                int width, int_height,
                o2_WebImageAttributes* format,
                int key, const char* alphalabel);
```

Arguments

`query` A query leading to an image.

`width` The width of an image. If this is not equal to 0, the value is used by some browsers to reserve space for the image in order to continue displaying the text of the received document before loading the inline image.

`height` The height of an image. If its value is not equal to 0, it is used by some browsers to reserve space for the image in order to continue displaying the text of the received document before loading the inline image.

`format` An object of the **`o2_WebImageAttributes`** class containing directives to change the attributes (borders, alignments, clickable, etc.) of the image.

Giving nil for **`format`** indicates that the image is not clickable, the borders are null and the alignment of the image will be the default used by the browser.

`key` A short integer that encodes the query. It is inserted into the bits result of this member function. A value of 0 means no encoding.

`alphalabel` A string that is displayed instead of the inline image by text-oriented browsers or when a browser is configured to load images only on demand.

Description This creates an inline image.

Returns An object of the **`o2_WebStream`** class.

make_report

Summary Produces an HTML output of a complex O_2 value.

Syntax

```
o2_WebStream make_report(Handle obj,
                         const char* query,
                         O2RpMode generic,
                         O2RpStatus header,
                         const char* userdata);
```

Arguments

`obj` The value to be printed. It can be an object, a tuple, a collection, a string or a byte (but not an integer, a char, a boolean or a real).

`query` A query leading to the value **`obj`**.

`generic` This indicates whether **`make_report`** uses user-defined member functions (**`RP_DEFAULT`**) or generic member functions (**`RP_GENERIC`**) to build the report.

`header` This indicates whether O_2Web adds a CGI header (**`RP_WITH_HEADER`**) or not (**`RP_NO_HEADER`**).

`userdata` A string to be returned to the programmer in the **`html_header`**, **`html_footer`**, **`html_report`** member functions when the user clicks on an anchor associated with this URL

Description This produces an HTML output of a complex O_2 value. It embeds a generated report in a report.

Returns An object of the **`o2_WebStream`** class.

make_url

Summary Creates a formatted URL.

Syntax

```
o2_WebStream make_url(const char* query,
                      const char* index,
                      const char* user_data, int key);
```

Arguments

`query`	A query.
`index`	A string referring to the index name of the returned document.
`user_data`	A string to be returned to the programmer in the **`html_header`**, **`html_footer`** and **`html_report`** member functions when the user clicks on an anchor associated with the URL.
`key`	A short integer used to encode the query inserted in the URL. A value of 0 means no encoding.

Description This creates a formatted URL leading to the object result of **`query`**.

Returns An object of the **`o2_WebStream`** class.

o2_WebFormAnalyser

This class helps programmers to decode HTML form results. It provides a set of member functions that retrieve the keywords in a form, the number of keywords, the values retrieved for the keywords, etc.,. It is used in conjunction with the **o2_WebFormItem** class.

This subsection presents the **o2_WebFormAnalyser** class and the following member functions:

- **get_all_values**
- **get_keywords**
- **get_nb_values**
- **get_nth_value**
- **get_raw_data**
- **get_unique_keywords**
- **get_values**
- **is_decoded**

```
class o2_WebFormAnalyser {
protected:
    char * raw_data;
    char * type;
    char decoded;
    d_Array<o2_WebFormItem>  elements;

public:
    char is_decoded ();
    d_Array<o2_WebFormItem> & get_all_values();
    int get_nb_values();
    void get_values( char * name, d_Array<o2_WebFormItem> & values);
    int get_nth_value(int i, o2_WebFormItem & item );
    void get_keywords(d_Array<char *> & values);
    void get_unique_keywords(d_Array<char *> & values);
    o2_WebStream get_raw_data();

    o2_WebFormAnalyser();
    o2_WebFormAnalyser (char *  params);
    ~o2_WebFormAnalyser();

} ;
```

get_all_values

Summary Retrieves an array of all the values in an HTML form.

Syntax

```
d_Array<o2_WebFormItem> & get_all_values();
```

Arguments None.

Description This returns an array of all the values in an HTML form. When a multiple selection list is used in a form, this member function returns the multiple items as different elements in the list.

Each element in the list is an object of the o2_WebFormItem class that contains detailed information about a single item in the form.

Returns An array of objects from the o2_WebFormItem class.

get_keywords

Summary Returns all the keywords retrieved by an HTML form.

Syntax `void get_keywords(d_Array<char *> & values);`

Arguments values An array of values.

Description This returns all the keywords retrieved by an HTML form. The same keyword may appear more than once if multiple values have been retrieved for a keyword.

Returns Nothing.

get_nb_values

Summary Returns the number of values.

Syntax `int get_nb_values();`

Arguments None.

Description This returns the number of values.

Returns An integer representing the number of values.

get_nth_value

Summary Retrieves the nth value in an HTML form concerning a specific attribute.

Syntax `int get_nth_value(int i, o2_WebFormItem & item);`

Arguments

`item`	An item in a form.
`i`	An integer.

Description This returns the `i`th value in an HTML form concerning the `item` attribute. This member function is only meaningful for forms containing items with potential multiple values (multiple selection list).

The number of values for an item can be retrieved by counting the number of elements of the `d_Array` set by the `get_values` function.

Returns The `i`th value.

get_raw_data

Summary Returns the form content as a raw byte string.

Syntax `o2_WebStream get_raw_data();`

Arguments None.

Description This returns the initial data retrieved from a form. It is used when O_2Web failed to decode the form.

Returns The content of the form.

get_unique_keywords

Summary Gets the unique keywords retrieved by an HTML form.

Syntax `void get_unique_keywords (d_Array<char *> & values);`

Arguments values An array of keywords.

Description This gets all the unique keywords retrieved by an HTML form. Even if multiple values have been retrieved for a keyword, a keyword only appears once in the returned list.

Returns Nothing.

get_values

Summary Gets an array of values in an HTML form concerning a specific attribute.

Syntax

```
void get_values(char * name,
   d_Array<o2_WebFormItem> & values);
```

Arguments

name	A specific attribute
values	The values in the form.

Description This gets a list of all the values in an HTML form concerning the **name** attribute. When a multiple selection list is used in a form, this member function returns the multiple items as different elements in the list.

Each element in the list is an object of the o2_WebFormItem class that contains detailed information about a single item in the form.

Returns Nothing.

is_decoded

Summary Establishes whether the data retrieved from an HTML form is decoded.

Syntax `char is_decoded ();`

Arguments None.

Description This establishes whether the data retrieved from an HTML form has been decoded by O_2Web. O_2Web can decode HTML forms that return two kinds of data:

(1) `application/x-www-form-urlencoded`

(2) `multipart/form-data`

O_2Web retrieves the MIME type of the form data using the **`CONTENT_TYPE`** CGI variable.

(1) is usually used by all web browsers.

(2), recently introduced by Netscape, permits the retrieval of entire files from a client. Its specification conforms to RFC 1867.

If O_2Web is unable to decode data, you can get the data and decode it yourself using the **`get_raw_data`** member function. This member function returns the initially retrieved data.

Returns A boolean.

A value of **`true`** indicates the data has been decoded.

A value of **`false`** indicates the data has not been decoded.

o2_WebFormItem

The objects of this class are returned by member functions of the **o2_WebFormAnalyser** class.

This subsection presents the **o2_WebFormItem** class and the following member functions:

- **get_file**
- **get_name**
- **get_type**
- **set_align**

```
class o2_WebFormItem {
protected:
    char * name;
    char * type;
    char * file;
    char * value;

public:
    char * get_name ();
    char * get_type ();
    char * get_file ();
    char * get_value ();

o2_WebFormItem & operator= (const o2_WebFormItem & item);

    o2_WebFormItem();
    ~o2_WebFormItem();
};
```

get_file

Summary Ascertains whether the value retrieved is the contents of a file.

Syntax `char * get_file ();`

Arguments None.

Description This ascertains whether the value retrieved is the contents of a file. It is only meaningful when the form data was posted with **`multipart/form-data`** encoding.

Returns The file name from which the value is obtained or an empty string if the value does not come from a file.

get_name

Summary Returns the keyword for an item retrieved from an HTML form.

Syntax `char * get_name ();`

Arguments None.

Description This returns the keyword for an item retrieved from an HTML form.

Returns The keyword of an item.

get_type

Summary Returns the type of an item retrieved from an HTML form.

Syntax `char * get_type ();`

Arguments None.

Description This returns the type of an item retrieved from an HTML form. It contains the MIME type of the retrieved item (text/html etc.).

Returns The type of an item.

get_value

Summary Returns the value of an item retrieved from an HTML form.

Syntax `char * get_value ();`

Arguments None.

Description This returns the value of an item retrieved from an HTML form.

Returns The value of an item.

o2_WebImageAttributes

This class is used to specify the attributes of an image. It must be used in conjunction with the **make_inline_image** member function of the **o2_WebAssistant** class or with the **o2_WebImageInliner** class.

This subsection presents the **o2_WebImageAttributes** class and the following member functions:

- **set_align**
- **set_border**
- **set_clickable**
- **set_hspace**
- **set_vspace**

```
class o2_WebImageAttributes {
protected:
    int hspace;
    int vspace;
    int border;
    char * align;
    char clickable;

public:
    void set_hspace (int h);
    void set_vspace (int v);
    void set_border (int b);
    void set_align (char * s);
    void set_clickable (char v);
    char *get_report ();

    o2_WebImageAttributes ();
    ~o2_WebImageAttributes();
};
```

set_align

Summary Specifies the way an image is aligned with text.

Syntax `void set_align (char * s);`

Arguments `s` A value specifying the type of alignment to be used.

Description This specifies the way an image is aligned with the current line of text.

Some values are always valid whatever the browser you are using. These are **`bottom`**, **`top`** and **`middle`**. Other values can be used but are only recognized by certain browsers such as Netscape Navigator©. These values are **`left`**, **`right`**, **`texttop`**, **`absmiddle`**, **`baseline`**, **`absbottom`**. These relate to either floating images (**`left`**, **`right`**) or the implementation of Netscape inline images.

Returns Nothing.

set_border

Summary Sets the thickness of the border around an image.

Syntax `void set_border (int b);`

Arguments **b** The thickness of the border.

Description This sets the thickness of the border around an image.

Returns Nothing.

set_clickable

Summary Specifies that an inline image is an imagemap.

Syntax `void set_clickable (char v);`

Arguments **v** An inline image.

Description This specifies that an inline image is an imagemap.

Returns Nothing.

set_hspace

Summary Sets the space to be left to the left and right of a floating image.

Syntax `void set_hspace (int h);`

Arguments **h** The space to be left to the left and right of an image.

Description This sets the space to be left between the left and right of a floating image and the text wrapped around it.

Returns Nothing.

set_vspace

Summary Sets the space to be left at the top and bottom of a floating image.

Syntax `void set_vspace (int v);`

Arguments **v** The space to be left at the top and bottom of an image.

Description This sets the space to be left between the top and bottom of a floating image and the text wrapped around it.

Returns Nothing.

o2_WebImageInliner

This class is used by the **make_inline_image** member function of the **o2_WebAssistant** class. It is used to generate inline images.

This subsection presents the **o2_WebImageInliner** class and the following member functions:

- **set_format**
- **set_height**
- **set_key**
- **set_label**
- **set_query**
- **get_report**
- **set_width**

```
class o2_WebImageInliner {

protected:
    char * query;
    int width;
    int height;
    char * label;
    int key;
    o2_WebImageAttributes * format;

public:
    void set_query (const char * s);
    void set_width (int w);
    void set_height (int h);
    void set_label (const char * s);
    void set_key (int k);
    void set_format (o2_WebImageAttributes * f);
    o2_WebStream get_report ();

    o2_WebImageInliner();
    ~o2_WebImageInliner();
};
```

set_format

Summary Specifies the format of an inline image.

Syntax `void set_format (o2_WebImageAttributes * f);`

Arguments `f` An object of the o2_WebImageAttributes class.

Giving nil for f indicates that the image is not clickable, the borders are null and the alignment of the image will be the default used by the browser.

Description This specifies the format of an inline image.

Returns Nothing.

set_height

Summary Sets the height of an inline image.

Syntax `void set_height (int h);`

Arguments **h** The height of an inline image.

Description This sets the height of an inline image. It is used by some browsers to reserve enough space for an image. It also allows browsers to continue to display text before the entire image has been read.

Returns Nothing.

set_key

Summary Sets the key to be used to encode a query.

Syntax `void set_key (int k);`

Arguments **k** The value of the key. A value of 0 means no encoding.

Description This sets the key to be used to encode a query that will be inserted in the bits result of this member function. A value of 0 means no encoding.

Returns Nothing.

set_label

Summary Specifies an alpha-numerical label to be used instead of an inline image.

Syntax `void set_label (const char * s);`

Arguments

`s`	The alpha-numerical label to be used instead of an inline image.

Description This specifies an alpha-numerical label to be used instead of an inline image for browsers which either do not support images or are configured to load images only on demand.

Returns Nothing.

set_query

Summary Affects a query resulting in an image to the **`o2_WebImageInliner`** class.

Syntax **`void set_query (const char * s);`**

Arguments **`s`** A string.

Description This affects a query resulting in an image to the o2_WebImageInliner class.

Returns Nothing.

get_report

Summary Builds an inline image.

Syntax `o2_WebStream get_report ();`

Arguments None.

Description This builds an inline image according to the parameters given.

Returns An object of the o2_WebStream class.

set_width

Summary Sets the width of an inline image.

Syntax void set_width (int w);

Arguments

`w`	The width of an inline image.

Description This sets the width of an inline image. It is used by some browsers to reserve enough space for an image. It also allows browsers to continue to display text before the entire image has been read.

Returns Nothing.

o2_WebStream

This class describes a stream. It is used in the other classes in the **`libo2webassistant.a`** library.

This subsection presents the following operations:

- **`append`**
- **`caseCompare`**
- **`char*`**
- **`compareTo`**
- **`contains`**
- **`data`**
- **`first`**
- **`index`**
- **`insert`**
- **`isAscii`**
- **`isNull`**
- **`last`**
- **`length`**
- **`mblength`**
- **`o2_WebStream`**
- **`operator=`**
- **`operator+=`**
- **`operator[]`**
- **`operator+`**
- **`operator==`**
- **`operator!=`**
- **`operator<`**
- **`operator<=`**
- **`operator>`**
- **`operator>=`**
- **`operator<<`**
- **`prepend`**
- **`remove`**
- **`replace`**
- **`toLower`**
- **`toUpper`**

append

Summary Appends a stream.

Syntax

```
o2_WebStream& append(const char *cs);              (1)
o2_WebStream& append(const char *cs, size_t len);  (2)
o2_WebStream& append(const o2_WebStream& str);     (3)
o2_WebStream&append(consto2_WebStream&str,size_tlen);
                                                   (4)
o2_WebStream& append(char c, size_t rep);          (5)
```

Arguments

cs	The value with which the stream is appended.
len	The length of the value or stream with which the stream is appended.
str	The stream with which the stream is appended.
c	The value with which the stream is appended.
rep	The length of **c**.

Description

(1)	This appends the stream with the character string pointed to by **cs**.
(2)	This appends the stream with the first **len** characters pointed to by **cs**.
(3)	This appends the stream with the stream in **str**.
(4)	This appends the stream with the first **len** characters from the stream **str**.
(5)	This appends the stream with the value in **c**, which is repeated **rep** times.

Returns A reference to the appended stream.

caseCompare

Summary Specifies whether operations are case sensitive or not.

Syntax `enum caseCompare {exact, ignoreCase};`

Arguments

`exact`	Operation is case sensitive.
`ignoreCase`	Operation is case insensitive.

Description This specifies whether operations are case sensitive or case insensitive.

Returns Nothing.

char*

Summary Converts a stream into a char*.

Syntax

```
operator const char* () const { return data ();};
```

Arguments None.

Description This converts a stream into a char*.

Returns The converted data.

compareTo

Summary Compares two streams with each other.

Syntax

```
int compareTo(const char* cs2, caseCompare cmp) const;
                                                        (1)
int compareTo(const o2_WebStream& str, caseCompare
cmp) const;                                             (2)
```

Arguments

cs2	The character string with which the current stream is compared.
cmp	The type of case comparison.
str	The stream with which the current stream is compared.

Description

(1)	This compares **cs2** with self.
(2)	This compares **str** with self.

Returns

(1)	-1, 0, or 1 if cs2 is lexicographically less than, equal to, or greater than self.
(2)	-1, 0, or 1 if str is lexicographically less than, equal to, or greater than self.

contains

Summary Matches the pattern of a stream.

Syntax

int contains(o2_WebStream& str, caseCompare = exact)const; **(1)**

int contains(const char* cs, caseCompare = exact)const; **(2)**

Arguments

`str` A stream.

`cs` A character string.

Description

(1) This ascertains whether the stream contains the character string cs.

(2) This ascertains whether the stream contains the stream str.

Returns 0 if the stream contains str, else 1.

data

Summary Returns the data in a stream.

Syntax `char* data() const;`

Arguments None.

Description This returns the character string contained in a stream.

Returns The data in a stream.

first

Summary Returns the index of the first occurrence of a given character in a stream.

Syntax

```
size_t first(char c) const;
```

Arguments

c	The character.

Description This returns the index of the first occurrence of the character **c** in a stream.

Returns The index of the first occurrence of c in the stream or NPOS if c is not in the stream.

index

Summary Returns the index that matches a specified pattern.

Syntax

```
size_t index(const char* pat, size_t i=0,
   caseCompare = exact) const;                    (1)
size_t index(o2_WebStream& pat, size_t i=0,
   caseCompare = exact) const;                    (2)
size_t index(const char* pat, size_t patlen,
   size_t i, caseCompare cmp);                    (3)
size_t index(o2_WebStream& pat, size_t patlen,
   size_t i, caseCompare cmp) const;              (4)
```

Arguments

pat	The pattern to search for.
i	The starting index.
patlen	The length of **pat**.
cmp	The type of case comparison.

Description

(1)	This starts from **i** and returns the index of the match of the first occurrence of the pattern **pat**.
(2)	This starts from **i** and returns the index of the match of the first occurrence of the pattern **pat**.
(3)	This starts from **i** and returns the index of the match of the first occurrence of the first **patlen** characters in the pattern **pat**.
(4)	This starts from **i** and returns the index of the match of the first occurrence of the first **patlen** characters in the pattern **pat**.

Returns The index that matches the specified pattern or NPOS if no match is found.

insert

Summary Inserts characters in a stream.

Syntax

```
o2_WebStream& insert(size_t pos, const char* str);  (1)
o2_WebStream& insert(size_t pos, const char* str,
   size_t len);                                     (2)
o2_WebStream& insert(size_t pos, o2_WebStream& str);
                                                    (3)
o2_WebStream& insert(size_t pos, o2_WebStream& str,
   size_t len);                                     (4)
```

Arguments

pos	The position of the first character to be inserted.
str	A stream of characters to be inserted.
len	The number of characters to be inserted.

Description

(1)	This inserts the characters in the character string **str** at the position **pos**.
(2)	This inserts the first **len** characters of the character string **str**, beginning at the position **pos**.
(3)	This inserts the characters of the stream **str** at the position **pos**.
(4)	This inserts the first **len** characters of the stream **str**, beginning at the position **pos**.

Returns A reference to the updated stream.

isAscii

Summary Ascertains whether a stream consists of only ASCII characters.

Syntax `int isAscii() const;`

Arguments None.

Description This ascertains whether a stream consists of only ASCII characters.

Returns 0 if the stream contains only ASCII characters, else 1.

isNull

Summary Ascertains whether a stream is null.

Syntax `int isNull() const;`

Arguments None.

Description This ascertains whether a stream is null.

Returns O if the stream is null, else 1.

last

Summary Returns the index of the last occurrence of a given character in a stream.

Syntax `size_t last(char c) const;`

Arguments **c** The character.

Description This returns the index of the last occurrence of the character **c** in a stream.

Returns The index of the last occurrence of c in the stream or NPOS if c is not in the stream.

length

Summary Returns the length of a stream.

Syntax `int length() const;`

Arguments None.

Description This returns the number of bytes in a stream.

Returns The number of bytes in a stream.

mblength

Summary Returns the length of a stream.

Syntax `size_t mblength() const;`

Arguments None.

Description This returns the number of characters in a stream, taking into account the possible multi-byte characters in the stream.

Returns The number of characters in a stream.

o2_WebStream

Summary Constructs a stream.

Syntax

```
o2_WebStream ();                                    (1)
o2_WebStream (const char* cs);                      (2)
o2_WebStream (const char* a, size_t N);             (3)
o2_WebStream (const o2_WebStream& str);             (4)
o2_WebStream (char c);                              (5)
o2_WebStream (char c, size_t N);                    (6)
```

Arguments

cs	A pointer to the data to be copied.
a	A pointer to the data to be copied.
N	The number of characters to be copied.
str	The stream to be copied.
c	A single character.

Description

(1)	This constructs a null stream.
(2)	This constructs a stream using the data pointed to by **cs**, up to the first terminating null.
(3)	This constructs a stream by copying exactly **N** characters from the data pointed to by **cs**.
(4)	This constructs a stream by copying another stream **str**.
(5)	This constructs a stream that will contain the single character **c**.
(6)	This constructs a stream that will contain the single character **c**, which is repeated **N** times.

Returns Nothing.

operator=

Summary Assignment operator.

Syntax

```
o2_WebStream& operator=(const o2_WebStream& str);   (1)
o2_WebStream& operator=(const char* cs);            (2)
```

Arguments

str	The stream to be copied.
cs	A pointer to the data to be copied.

Description

(1)	This copies the data in **str** to the stream.
(2)	This copies the characters pointed to by **cs** to the stream.

Returns A reference to the updated stream.

operator+=

Summary Append operator.

Syntax

```
o2_WebStream& operator+=(const o2_WebStream& str);  (1)
o2_WebStream& operator+=(const char* cs);           (2)
```

Arguments

`str`	The stream to be copied.
`cs`	A pointer to the data to be copied.

Description

(1)	This appends the data in **`str`** to the stream.
(2)	This appends the characters pointed to by **`cs`** to the stream.

Returns A reference to the updated stream.

operator[]

Summary Returns the character at a given position in a stream.

Syntax

```
char& operator[] (size_t i);
char operator[] (size_t i) const;
```

Arguments

`i`	The position in the stream.

Description This returns the character at the position `i` in a stream.

Returns The character at a given position in a stream.

operator+

Summary Concatenate operator.

Syntax

```
o2_WebStream operator+(const o2_WebStream&,
   const o2_WebStream&);                                    (1)

o2_WebStream operator+(const o2_WebStream&, const
char*);
                                                            (2)

o2_WebStreamoperator+(constchar*,consto2_WebStream&);
                                                            (3)
```

Arguments The streams or strings to be added together.

Description

(1) This concatenates two streams.

(2) This concatenates a stream with a string.

(3) This concatenates a string with a stream.

Returns The concatenated stream.

operator==

Summary Compares the identities of two streams.

Syntax

```
inline int operator==(const o2_WebStream&,
   const o2_WebStream&);                                  (1)
int operator==(const o2_WebStream&, const char*);         (2)
inline int operator==(const char*, const o2_WebStream&);  (3)
```

Arguments The streams or strings to be compared.

Description

(1) This compares the identities of two streams.

(2) This compares the identities of a stream and a string.

(3) This compares the identities of a string and a stream.

Returns 0 if they are the same, else 1.

operator!=

Summary Compares the identities of two streams.

Syntax

```
inline int operator!=(const o2_WebStream&,
   const o2_WebStream&);                                    (1)

inline int operator!=(const o2_WebStream&, const char*);
                                                            (2)

inline int operator!=(const char*, const o2_WebStream&);
                                                            (3)
```

Arguments The streams or strings to be compared.

Description

(1) This compares the identities of two streams.

(2) This compares the identities of a stream and a string.

(3) This compares the identities of a string and a stream.

Returns 0 if they are not the same, else 1.

operator<

Summary Compares two streams.

Syntax

```
inline int operator<(const o2_WebStream&,
 const o2_WebStream&);                                    (1)

inline int operator<(const o2_WebStream&, const char*);
                                                          (2)

inline int operator<(const char*, const o2_WebStream&);
                                                          (3)
```

Arguments The streams or strings to be compared.

Description

(1) This compares two streams.

(2) This compares a stream and a string.

(3) This compares a string and a stream.

Returns

(1) 0 if the first stream is less than the second stream, else 1.

(2) 0 if the stream is less than the string, else 1.

(3) 0 if the string is less than the stream, else 1.

operator<=

Summary Compares two streams.

Syntax

```
inline int operator<=(const o2_WebStream&,
          const o2_WebStream&);                    (1)

inline int operator<=(const o2_WebStream&, const char*);
                                                   (2)

inline int operator<=(const char*, const o2_WebStream&);
                                                   (3)
```

Arguments The streams or strings to be compared.

Description

(1) This compares two streams.

(2) This compares a stream and a string.

(3) This compares a string and a stream.

Returns

(1) 0 if the first stream is less than or equal to the second stream, else 1.

(2) 0 if the stream is less than or equal to the string, else 1.

(3) 0 if the string is less than or equal to the stream, else 1.

operator>

Summary Compares two streams.

Syntax

```
inline int operator>(const o2_WebStream&,
const o2_WebStream&);                                    (1)

inline int operator>(const o2_WebStream&, const char*);
                                                         (2)

inline int operator>(const char*, const o2_WebStream&);
                                                         (3)
```

Arguments The streams or strings to be compared.

Description

(1) This compares two streams.

(2) This compares a stream and a string.

(3) This compares a string and a stream.

Returns

(1) 0 if the first stream is greater than the second stream, else 1.

(2) 0 if the stream is greater than the string, else 1.

(3) 0 if the string is greater than the stream, else 1.

operator>=

Summary Compares two streams.

Syntax

```
inline int operator>=(const o2_WebStream&,
   const o2_WebStream&);                                    (1)

inline int operator>=(const o2_WebStream&, const char*);
                                                            (2)

inline int operator>=(const char*, const o2_WebStream&);
                                                            (3)
```

Arguments The streams or strings to be compared.

Description

(1) This compares two streams.

(2) This compares a stream and a string.

(3) This compares a string and a stream.

Returns

(1) 0 if the first stream is greater than or equal to the second stream, else 1.

(2) 0 if the stream is greater than or equal to the string, else 1.

(3) 0 if the string is greater than or equal to the stream, else 1.

operator<<

Summary Output operator.

Syntax

```
inlineostream&operator<<(ostream&,consto2_WebStream&);
```
(1)

```
inline o2_WebStream& operator<<(o2_WebStream&,
          const o2_WebStream&);
```
(2)

```
inline o2_WebStream& operator<<(o2_WebStream&,
          const char*);
```
(3)

```
inline o2_WebStream& operator<<(o2_WebStream&, int);
```
(4)

```
inline o2_WebStream& operator<<(o2_WebStream&,
          unsigned char);
```
(5)

Arguments The objects of the class **`o2_WebStream`** class in which the other arguments will be input.

Description

(1) Outputs an o2_WebStream on an ostream.

(2) Outputs an o2_WebStream on another o2_WebStream.

(3) Outputs a string of characters on an o2_WebStream.

(4) Outputs an integer on an o2_WebStream.

(5) Outputs an unsigned character on an o2_WebStream.

Returns A reference to the updated stream.

prepend

Summary Prepends a stream.

Syntax

```
o2_WebStream& prepend(const char *cs);                    (1)
o2_WebStream& prepend(const char *cs, size_t len);(2)
o2_WebStream& prepend(const o2_WebStream& str);           (3)
o2_WebStream& prepend(const o2_WebStream& str, size_t len);  (4)
o2_WebStream& prepend(char c, size_t rep);                (5)
```

Arguments

cs	A pointer to some characters.
len	The length of **cs** or **str**.
str	The stream with which the stream is prepended.
c	A single character.
rep	The number of times **c** will be repeated.

Description

(1)	This prepends a stream with the characters pointed to by **cs**.
(2)	This prepends a stream with the first **len** characters pointed to by **cs**.
(3)	This prepends a stream with the stream **str**.
(4)	This prepends a stream with the first **len** characters of **str** or the length of **str**, whichever is less.
(5)	This prepends a stream with the character **c**, which is repeated **rep** times.

Returns A reference to the prepended stream.

remove

Summary Removes characters from a stream.

Syntax

```
o2_WebStream& remove(size_t pos);(1)
o2_WebStream& remove(size_t pos, size_t len);(2)
```

Arguments

pos	The position of the first character to be removed.
len	The number of characters to be removed.

Description

(1)	This removes the characters from the position **pos** to the end of the stream.
(2)	This removes a maximum of **len** characters, beginning at the position **pos**, from a stream.

Returns A reference to the updated stream.

replace

Summary Replaces characters in a stream.

Syntax

```
o2_WebStream& replace(size_t pos, size_t n1,
    const char* cs);                                    (1)

o2_WebStream& replace(size_t pos, size_t n1,
    const char* cs, size_t n2);                         (2)

o2_WebStream& replace(size_t pos, size_t n1,
    const o2_WebStream& str);                           (3)

o2_WebStream& replace(size_t pos, size_t n1,
    const o2_WebStream& str, size_t n2);                (4)
```

Arguments

pos The position of the first character to be replaced.

n1 The maximum number of characters to be replaced.

cs The character string containing the data to be inserted.

n2 The maximum number of characters to be inserted.

str The stream containing the data to be inserted.

Description

(1) This replaces a maximum of **n1** characters in a stream, starting at the position **pos**, by the characters pointed to by **cs**.

(2) This replaces a maximum of **n1** characters in a stream, starting at the position **pos**, by the first **n2** characters pointed to by **cs**.

(3) This replaces a maximum of **n1** characters in a stream, starting at the position **pos**, by the characters of the stream **str**.

(4) This replaces a maximum of **n1** characters in a stream, starting at the position **pos**, by the first **n2** characters of the stream **str**.

Returns A reference to the updated stream.

toLower

Summary Returns a lower-case version of a stream.

Syntax `o2_WebStream toLower();`

Arguments None.

Description This returns a lower-case version of a stream

Returns A lower-case version of the stream.

toUpper

Summary Returns an upper-case version of a stream.

Syntax `o2_WebStream toUpper();`

Arguments None.

Description This returns an upper-case version of a stream

Returns An upper-case version of the stream.

6.4 o2_Web

This section presents the **o2_Web** class and describes the following member functions:

- **begin**
- **init**
- **end**
- **loop**
- **enroll**
- **enroll_path**
- **get_option**
- **set...**
  - set_default_env
  - set_dispatchername
  - set_libname
  - set_libpath
  - set_servername
  - set_swapdir
  - set_sysdir
  - **set**_systemname
  - set_verbose

```
class o2_Web {
public:
  o2_Web();
  ~o2_Web();

  int  begin(int argc, register char *argv[]);
  int  begin(int argc, register char *argv[], const char *sysdir,
             const char *systemname, const char *servername,
             const char *dispatchername, int verbose);
  int  begin(int argc, register char *argv[], const char *sysdir,
             const char *systemname, const char *servername,
             const char *dispatchername, const char *swapdir,
             char * const *libpath, char * const *libname,
             int commitfrequency, int verbose);

  int init();
  int end();
  int loop();

  void set_systemname(const char *systemname);
  void set_servername(const char *servername);
  void set_sysdir(const char *sysdir);
  void set_swapdir(const char *swapdir);
  void set_dispatchername(const char *dispatchername);
  void set_commitfrequency(const char *commitfrequency);
  void set_commitfrequency(int commitfrequency);
  void set_verbose(int verbose);
  void set_libpath(char * const *libpath);
  void set_libname(char * const *libname);
  void set_default_env();
};
```

begin

Summary **Starts up a connection to the database.**

Syntax

```
int begin (int argc, register char *argv[]);

int begin (int argc, register char *argv[], const char
          *sysdir,
          const char *systemname,
          const char *servername,
          const char *dispatchername,
          int verbose);

int begin (int argc, register char *argv[], const char
          *sysdir,
          const char *systemname,
          const char *servername,
          const char *dispatchername,
          const char *swapdir,
          char * const *libpath,
          char * const *libname,
          int commitfrequency, int verbose);
```

Arguments

argc Number of arguments of the C++ executable.

argv List of arguments of the C++ executable.

systemname Name of database system as defined in the **systems** file of the O_2 installation directory. This information is mandatory. It can be given as a parameter or by calling **set_systemname(char *)** before beginning the session. It can also be set by **set_default_env()** in which case it is found in the environment variable O2SYSTEM.

servername Name of machine on which the O_2 server is running. If it is NULL, O_2 will find this information in the **systems** file of the O_2 installation directory. It can also be given by calling **set_servername(char *)** before beginning the session. It can also be set using **set_default_env()** in which case it is found in the environment variable O2SERVER.

sysdir Path to the directory where O_2 is installed. This information is mandatory. It can be given as a parameter,

or by calling **`set_sysdir(char *)`** before beginning the session. It can also be set by **`set_default_env()`** in which case it is found in the environment variable O2HOME.

`swapdir` Path to a directory where a swap file can be created if O_2 needs it. It can be NULL, in which case the swap directory in the O_2 directory is used (See the System Administration Guide).

`libpath` A NULL-terminated array of character strings, where each string gives a directory path. O_2 searches these directories for libraries named in **`libname`** if dynamic linking is needed. It may be NULL.

`libname` A NULL-terminated array of character strings, each specifying a library name to use when linking and loading functions dynamically. It may be NULL.

`dispatchername` The name of the machine on which o2open_dispatcher is running.

`commitfrequency` The frequency with which a commit is carried out.

`verbose` An integer specifying the session as a verbose session.

Description Starts up the connection to the database and connects up to the server.

Returns 0 if the connection was carried out successfully. If not, an error code is given.

Example

```
main (int argc, register char * [] argv){
o2_Web my_session;
if (my_session.begin (argc, argv, "smith",
"mick",
                    "/usr/bin/o2"))

    cerr <<"Error: cannot start the
connection"<<endl;
...
}
my_session.end();
my_session.set_systemname("my_system");
my_session.set_servername(0);
my_session.set_sysdir("/usr/bin/o2");
my_session.begin(argc, argv);
};
```

end

Summary Ends an O_2 session.

Syntax **`int end();`**

Arguments None.

Description Ends an O_2 session and the connection to the O_2 server. A commit is carried out automatically.

Returns 0 if the session was successfully ended. Else a non-zero value.

Example

```
main (int argc, register char * [] argv){
o2_Web my_session;
if (my_session.begin (argc, argv, "smith",
"mick",
                    "/usr/bin/o2"))
    cerr <<"Error: cannot start the
connection"<<endl;
...
my_session.end();
}
```

init

Summary Starts an O_2 session

Syntax `int end();`

Arguments None.

Description Starts and O_2 session.

Returns 0 if `init` is successful. Else a non-zero value.

loop

Summary	Creates a loop in the session.
Syntax	`int loop();`
Arguments	None.
Description	This creates a loop in the session.
Returns	0 if the operation was successful. Else a non-zero value.

enroll

Summary Registers an option to be recognized by the O_2 options manager.

Syntax

```
static int o2_Web::enroll (const char * const name,
                           const char * const confname,
                           const char * const optname,
                           char *dflt,
                           const OptionType t,
                           const char * const desc,
                           const OptionMode
                           mode=Replace);
static int o2_Web::enroll (const char * const name,
                           const char * const confname,
                           const char * const optname,
                           long dflt,
                           const OptionType t,
                           const char * const desc,
                           const OptionMode
                           mode=Replace);
static int o2_Web::enroll (const char * const name,
                           const char * const confname,
                           const char * const optname,
                           char dflt,
                           const OptionType t,
                           const char * const desc,
                           const OptionMode
                           mode=Replace);
static int o2_Web::enroll (const char * const name,
                           const char * const confname,
                           const char * const optname,
                           double dflt,
                           const OptionType t,
                           const char * const desc,
                           const OptionMode
                           mode=Replace);
```

Arguments

name A string that indicates the name of the option. This name is used for retrieving the value of the option.

confname A string that indicates under which name the value of this option can be given in a configuration file.

`optname` A string that indicates under which name the value of this option can be given in the environment variable or at the command line.

`dflt` The default value of the option. This value is retrieved if the end user does not provide a value for this option.

t — A flag from the OptionType enumeration

NoValue	The option represents a boolean value. No value must follow this option else this is an error and the usage is dislayed.
OptionalValue	The option can have an associated value, but this is not mandatory.
MandatoryValue	The option must have an associated value. If none is given, the usage is displayed.

desc — A string describing the option. This string is displayed when the **o2_Web::usage** function is called or when a parsing error is detected.

mode — A flag from the OptionMode enumeration.

Add	If the option is repeated, the associated values are stored in an array that you can retrieve with **o2_Web::get_option()**.
Append	If the option is repeated, the associated values are concatenated to form a single string that can be retrieved with **o2_Web::get_option()**. This flag is meaningful only if the option values are considered as strings.
Replace	By default, only one value is associated to this option. If the option is repeated the last value is used.

Description

These member functions allow you to register new options on the O_2 options manager. See the *ODMG C++ Binding Guide* for explanations on the option mechanism and a complete example.

Each of these functions allow you to enroll one option. There is one function for each type of option value: string, character, integer or real.

Returns

1 if successful.
0 if the option could not be enrolled.
-1 if there was an internal error in the option manager. if successful.

Example

See the *ODMG C++ Binding Guide*.

enroll_path

Summary Allows you to register hierarchical options.

Syntax

```
static int o2_Web::enroll_path (const char * path);
```

Description This member function allows you to register hierarchical options in your configuration file (called **.o2rc** by default). Hierarchical options are described as a path, i.e., an ordered list of options such as:

```
system.base.loadname
```

The path passed to **enroll_path** is composed of the internal names of the options (first parameter of **o2_Web::enrol()**), separated by the "." character.

Returns 0 if successful.
-1 if there was an internal error.

Example For option "-base" to be specific of a given system, write:

```
o2_Web::enroll_path("system_name.base_name");
```

where **base_name** is the internal name of option **-base** and **system_name** is the internal name of option **-system**.

Then you can write in your configuration file:

```
my_system.base = CustomerBase
```

If you had not called **o2_Web::enroll_path()**, you could only have written:

```
base = CustomerBase
```

in your configuration file.

get_option

Summary Retrieves the value of an option.

Syntax

```
static int o2_Web::get_option ( const char *name,
                                char *&value,
                                int ind = -1);

static int o2_Web::get_option ( const char *name,
                                long &value,
                                int ind = -1);

static int o2_Web::get_option ( const char *name,
                                double &value,
                                int ind = -1);

static int o2_Web::get_option ( const char *name,
                                char &value,
                                int ind = -1);
```

Arguments

name A string that indicates the internal name of the option as defined in the corresponding **o2_Web::enroll** member function.

value This argument points to the returned value. If no value can be retrieved from the command line, the environment variable (**O2OPTIONS**) or the configuration file (.**o2rc**), the default value given in **o2_Web::enroll()** is used.

ind An index that is used if the user enters an option several times. If you have registered the option with the replace or append mode, you should set this argument to -1.

If the index is -1, the last value entered by the end-user is returned. If the index is >= 0, the **ind**-th value is returned. If the index is too large, the returned value is NULL.

Description This member function allows you to retrieve the value of the registered options. This function should only be called for options that are registered.

This function may be used in the check function, which can be registered by the **o2_Web::begin** member function. See the *ODMG C++ Binding Guide* for information on the option mechanism.

Returns 0 if successful.
-1 if the option was notregistered with **o2_Web::enroll()**.

set...

Summary Sets the various session arguments.

Syntax

```
void set_systemname(const char *systemname);

void set_servername(const char *servername);

void set_sysdir(const char *sysdir);

void set_swapdir(const char *swapdir);

void set_dispatchername(const char *dispatchername);

void set_commitfrequency(const char
*commit_frequency);

void set_commitfrequency(const char
*commit_frequency);

void set_verbose(int verbose);

void set_libpath(char * const *libpath);

void set_libname(char * const *libname);

void set_default_env();
```

Description Explicitly sets various session parameters before beginning the session with **`begin(argc, argv, mode);`**.

`set_default_env();` sets:

system name to O2SYSTEM,

server name to O2SERVER and

O_2 installation directory to O2HOME.

Returns Nothing.

Note

Refer to **`begin()`** for additional information.

6.5 O2Web Commands

This section outlines the following O_2Web system commands:

The programs called by these commands can be found in the **`bin`** subdirectory of the O_2 installation directory.

The commands are:

- **`o2open_dispatcher`**
- **`o2web_gateway`**

o2open_dispatcher

Summary Starts an O_2Web dispatcher.

Syntax **`o2open_dispatcher [-v]`**

Description This command starts a new O_2Web dispatcher. An O_2Web dispatcher registers all the O_2Web servers running on a LAN and is queried by the O_2Web gateway to get the address of a server able to answer an OQL query.

When choosing an O_2Web server to answer a gateway query, the dispatcher uses heuristics. A score is computed for each server running and the server with the best score is returned to the gateway. The following elements enter into the computation of the score:

- `a server is running on the same host as the gateway.`
- `a server is already connected to the database to which the query is asked.`
- `the current load of each server (the number of queries treated).`

Options **`-v`** display additional information on the **`o2open_dispatcher`** activity.

Files **`/etc/services`** (UNIX) or **`$WINDIR\system32\drivers\etc\services`** (Windows NT) a file containing the port number and the protocol used by other programs to access the o2open_dispatcher.

See Also **`o2server`**, **`o2web_gateway`**

o2web_gateway

Summary Starts an O_2Web gateway.

Syntax **`o2web_gateway`**

Description This command starts a new O_2Web gateway. A gateway is not launched by a user but by an HTTP server in order to answer an OQL query. The **`o2web_gateway`** program complies with the CGI protocol.

Once launched, the O_2Web gateway has to find and query the O_2Web dispatcher to get the address of an O_2Web server that can answer the query. The gateway finds the dispatcher host name with the **`O2OPEN_DISPATCHER`** environment variable or in a system-dependent file (**`/etc/o2openaccess`** for UNIX and **`$WINDIR\system32\driers\etc\o2openaccess`** for Windows NT). The gateway connects to the dispatcher using the TCP port found by querying the system for the port of the **`o2open_dispatcher`** service (Refer to 2.3 for details of how you can specify this information).

This program is generally installed in a special directory of the HTTP server containing the CGI scripts.

Environment variables The O_2Web gateway environment is built by the HTTP server, particularly the CGI environment variables.

`O2OPEN_DISPATCHER`the dispatcher host name (not with all HTTP servers.

Files **`/etc/o2openaccess`** (UNIX) or **`$WINDIR\system32\drivers\etc\o2web`** (Windows NT) a file containing the dispatcher host name.

`/etc/services` (UNIX) or **`$WINDIR\system32\drivers\etc\services`** (Windows NT) a file containing a list of TCP and UDP services.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX