CGI Extensions for the ClearSilver CGI Kit

Thomas J. Moore

Version 1.0 Revision 196
2010

Abstract

This document describes and implements a few enhancements to the CGI library provided by ClearSilver, in addition to what is provided by the *Generic ClearSilver and GLib Module Build Support* document.

This document was generated from the following sources, all of which are attached to the original electronic forms of this document:

```
$Id: build.nw 197 2012-12-06 05:02:53Z darktjm $
$Id: cs-glib.nw 198 2012-12-07 05:40:28Z darktjm $
$Id: cgi-supt.nw 196 2012-12-06 05:02:09Z darktjm $
```

Contents

Chapter 1

Introduction

The CGI kit included with ClearSilver works well enough for most applications, but some improvements could be made. First, like most CGI libraries, file upload control is inadequate: all uploads are placed in the same temporary directory (which bypasses per-user quotas, among other things), and the only provided callback does not get access to already-known CGI parameters or file attributes. Second, there is no direct FastCGI[1] support. FastCGI allows the sharing of long setup times among multiple connections; for example, connections can be made to a database before the user ever connects to the web server. Third, there is no macro library for common form operations. This is probably for the better, since limitations in such a library tend to prompt either rewrite or convoluted workarounds. Fourth, there is no support for WebDAV[2] request processing. This is normally done in a web server module, but there is no reason not to do it in (Fast)CGI.

In addition to the basic parsing enhancements and FastCGI support, a simple session management infrastructure is implemented, using either files or a database for session storage. This is then used to help implement authentication caching for the Central Authentication Service[3].

3a ⟨*(build.nw) Version Strings* 3a⟩≡

```
"$Id: cgi-supt.nw 196 2012-12-06 05:02:09Z darktjm $\n"
```

3b ⟨*(build.nw) Sources* 3b⟩≡

```
$Id: cgi-supt.nw 196 2012-12-06 05:02:09Z darktjm $
```

3c ⟨*(build.nw) Common NoWeb Warning* 3c⟩≡ (4b 8d)

```
# $Id: cgi-supt.nw 196 2012-12-06 05:02:09Z darktjm $
```

[1] http://www.fastcgi.com
[2] http://www.webdav.org
[3] http://www.jasig.org/cas

Chapter 2

HTML Form Support Macros

Using macros to display widgets makes the HTML harder to edit with normal tools, and also makes it more dependent on the macro language. The conditional HTML is complex enough that the use of macros is worth the potential problems.

4a ⟨*(cs-glib.nw) CS files* 4a⟩≡

```
html_macros.cs \
```

4b ⟨*html_macros.cs* 4b⟩≡

```
<?cs ⟨(build.nw) Common NoWeb Warning 3c⟩ ?><?cs
⟨Generic ClearSilver HTML Macros 4d⟩
# ?>
```

4c ⟨*(build.nw) Install other files* 4c⟩≡ 7i▷

```
⟨(cs-glib.nw) Install ClearSilver templates (imported)⟩
```

First, to start out a form, the form header must be printed.

4d ⟨*Generic ClearSilver HTML Macros* 4d⟩≡ (4b) 4e▷

```
def:form(fname,hasfiles)
  ?><form name="<?cs var:fname ?>" action="<?cs
       var: CGI.ScriptName + CGI.PathInfo ?>"<?cs
          if:hasfiles ?> enctype="multipart/form-data"<?cs /if ?>><?cs
/def ?><?cs
```

For a standard text widget, only the variable name and default value are required. As with all of these macros, the CGI parameter value is retrieved from the `Query` variable hierarchy.

4e ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁4d 5a▷

```
def:text(varn,default)
  ?><input type="text" size="32" name="<?cs var:varn ?>" value="<?cs
    if: ?Query[varn] ?><?cs
      var:Query[varn] ?><?cs
    else ?><?cs
      var:default ?><?cs
    /if ?>" /><?cs
/def ?><?cs
```

Repeated text widgets are the same as regular text widgets, but a sequence number is attached to the variable name. The hierarchy is expected to go one level lower, using the sequence number. The first widget (0) is actually just like a normal text widget.

5a ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁4e 5b▷

```
def:rtext(varn,seq,default)
  ?><input type="text" size="32" name="<?cs var:varn ?>" value="<?cs
    if:?Query[varn][seq] ?><?cs
      var:Query[varn][seq] ?><?cs
    else ?><?cs
      if:seq == "0" && ?Query[varn] ?><?cs
        var:Query[varn] ?><?cs
      else ?><?cs
        var:default ?><?cs
      /if ?><?cs
    /if ?>" /><?cs
/def ?><?cs
```

Hidden parameters are just printed once for every available value.

5b ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁5a 5c▷

```
def:hidden(varn)
  ?><?cs
    if:len(Query[varn]) ?><?cs
      each:v = Query[varn]
       ?><input type="hidden" name="<?cs var:varn ?>" value="<?cs var:v ?>" />
<?cs  /each ?><?cs
    elseif:?Query[varn]
      ?><input type="hidden" name="<?cs var:varn ?>" value="<?cs
                        var:Query[varn] ?>" />
<?cs
    /if ?><?cs
/def ?><?cs
```

File selection widgets do not have defaults. Even if the variable is provided by the CGI, it does not necessarily match a local file name.

5c ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁5b 5d▷

```
def:file(varn)
  ?><input type="file" size="32" name="<?cs var:varn ?>" value="<?cs
                                        var:Query[varn] ?>" /><?cs
/def ?><?cs
```

Radio selections are checked if the CGI parameter matches the value, or if this is the default and the CGI parameter either does not exist or has no value.

5d ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁5c 6a▷

```
def:radio(varn, val, isdef)
  ?><input type="radio" value="<?cs var:val ?>" name="<?cs var:varn ?>"<?cs
    if:?Query[varn]["0"] ?><?cs
      each:i = Query[varn] ?><?cs
        if: i == val ?> checked="checked" <?cs /if ?><?cs
      /each ?><?cs
    else ?><?cs
      if: Query[varn] == val ?> checked="checked" <?cs
      else ?><?cs
        if:isdef && (!?Query[varn] || Query[varn] == "") ?> checked="checked" <?cs /if?><?cs
      /if ?><?cs
    /if ?> /><?cs
/def ?><?cs
```

Plain checkboxes always default to unchecked, as it is too hard to tell the difference between unchecked and missing. If a hidden value were added, it would be a little easier, but possibly inconsistent.

6a ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁5d 6b▷

```
def:checkbox(varn)
  ?><input type="checkbox" value="Y" name="<?cs var:varn ?>"<?cs
     if:Query[varn] ?> checked="checked" <?cs /if ?>/><?cs
/def ?><?cs
```

Date widgets should really have a JavaScript calendar-based date selection button, but instead they are currently just plain text widgets. However, they are assigned an easy-to-identify class name to make adding the popup with dynamic HTML easier.

6b ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁6a 6c▷

```
def:date(varn, def)
  ?><input type="text" class="datesel" size="12" name="<?cs var:varn ?>" value="<?cs
    if: ?Query[varn] ?><?cs
      var:Query[varn] ?><?cs
    else ?><?cs
      var:default ?><?cs
    /if ?>" /><?cs
/def ?><?cs
```

Submit buttons display the value as the button text, so the value cannot be used to determine the action. Instead, the parameter name is used. The label is the configuration value of `labels.`*action*, and the parameter name is `action.`*action*.

6c ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁6b 7a▷

```
def:submit(action)
  ?><input type="submit" name="action.<?cs var:action ?>" value="<?cs
                                        var:labels[action] ?>" /><?cs
/def ?><?cs
```

Finally, a facility is provided for showing or hiding various HTML divisions using a selection widget (aka drop-down menu). If JavaScript is not supported by the browser, then all divisions will be displayed. The selection widget itself should not be displayed, either, so it is rendered using JavaScript. The less-than and greater-than symbols for the HTML tags are also escaped, so that simplistic HTML parsers don't get confused. Note that only one division set, and one selection widget are supported per page.

In order to remain independent of web servers, this could be included entirely in-line. However, as a general rule, it is better to keep anything more complex than a function call in a separate JavaScript file to avoid quoting issues, and enable preparsing optimizations by the browser.

6d ⟨*JavaScript Files* 6d⟩≡ (7g)

```
html_prefix.js \
```

6e ⟨*html_prefix.js* 6e⟩≡

```
function showdiv(divno)
{
  var mydiv, i;

  for(i = 1; (mydiv = document.getElementById('div' + i)); i++)
    mydiv.style.display = divno == i ? "block" : "none";
}
```

7a ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁6c 7b▷

```
def:optdiv_onload()
  ?>showdiv(document.forms[0]._ignore.selectedIndex);<?cs /def ?><?cs
```

7b ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁7a 7c▷

```
def:optdiv_start() ?><script language="javascript" type="text/javascript">
//<![CDATA[
document.writeln('\x3Cselect name="_ignore" onchange="showdiv(this.value);"\x3E');
document.write('\x3Coption value="0"\x3E<?cs var:labels.optdiv ?>');
document.writeln('\x3C/option\x3E');<?cs /def ?><?cs
```

7c ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁7b 7d▷

```
def:optdiv_opt(n, v)
  ?>document.write('\x3Coption value="<?cs var:v ?>"\x3E<?cs var:n ?>');
    document.writeln('\x3C/option\x3E');<?cs
     /def ?><?cs
```

7d ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁7c 109b▷

```
def:optdiv_end()
  ?>document.writeln('\x3C/select\x3E');
//]]>
</script><?cs /def ?><?cs
```

This does require that yet another location must be chosen for installation. The HTML can still be kept generic enough for easy run-time override of the installation location.

7e ⟨*CGI Support Configuration* 7e⟩≡ (149b) 10e▷

```
# Location of JavaScript on the server
#server_script_path = /js
```

7f ⟨*(build.nw) makefile.config* 7f⟩≡ 29e▷

```
# Installation directory for javascript files
JS_DIR:=/var/www/html/js
```

7g ⟨*(build.nw) makefile.vars* 7g⟩≡ 8f▷

```
JS_FILES = ⟨JavaScript Files 6d⟩
```

7h ⟨*(build.nw) Plain Files* 7h⟩≡ 116a▷

```
$(JS_FILES) \
```

7i ⟨*(build.nw) Install other files* 4c⟩+≡ ◁4c

```
mkdir -p $(DESTDIR)$(JS_DIR)
cp -p $(JS_FILES) $(DESTDIR)$(JS_DIR)
```

7j ⟨*html_prefix.cs* 7j⟩≡

```
<script language="javascript" type="text/javascript" src="<?cs
       alt:server_script_path ?>/js<?cs /alt ?>/html_prefix.js"></script>
```

Chapter 3

FastCGI Support

FastCGI support is implemented using the mostly undocumented direct functions of the `libfcgi` API. The standard I/O overrides are better documented, but intrusively override many standard functions. There is no standard method provided by `libfcgi` to find the libraries and include files, so standard `LDFLAGS` and `CFLAGS` overrides must be used if needed.

8a ⟨*(cs-glib.nw) Library* `cs-supt` *Members* 8a⟩≡ 97a▷

```
cgi.o
```

8b ⟨*(build.nw) Common C Includes* 8b⟩≡ 12a▷

```
⟨CGI Prerequisite Headers 8c⟩
#include "cgi.h"
```

8c ⟨*CGI Prerequisite Headers* 8c⟩≡ (8b) 30a▷

```
#include <fcgiapp.h>
```

8d ⟨*cgi.h* 8d⟩≡

```
/*
   ⟨(build.nw) Common NoWeb Warning 3c⟩
*/
#ifndef _CGI_H
#define _CGI_H
⟨CGI Support Global Definitions 8g⟩
#endif /* _CGI_H */
```

8e ⟨*cgi.c* 8e⟩≡

```
⟨(build.nw) Common C Header (imported)⟩

⟨CGI Support Variables 10c⟩
⟨CGI Support Functions 9b⟩
```

8f ⟨*(build.nw) makefile.vars* 7g⟩+≡ ◁7g 29d▷

```
EXTRA_LDFLAGS += -lfcgi
```

Each FastCGI session is like an independent CGI program. This implies that no global variables should be used; instead, a master state structure is passed around.

8g ⟨*CGI Support Global Definitions* 8g⟩≡ (8d) 12b▷

```
typedef struct cgi_state_t cgi_state_t;
⟨CGI State Dependencies 9c⟩
struct cgi_state_t {
  ⟨CGI State Members 9e⟩
};
```

9a ⟨*(build.nw) Known Data Types* 9a⟩≡ 9d▷

```
cgi_state_t,%
```

The general program structure supported here is to perform global initializations before the main loop, and in the main loop spawn a thread for every incoming connection. The threads get the `cgi_state_t` structure for the current CGI execution. An initialization function is provided for the global initializations, and a generic main loop with a callback is provided for spwan loop. The callback is called from within the thread rather than as the main body of the thread so that cleanup is automatic. That means that the callback must be stored in the CGI state for the thread to call. In addition, a generic cleanup callback is provided; this should be overridden using a statically stored chain for each overrider to ensure that all cleaners are called. Calling from within the thread also allows the return code from the callback to be used; if an error is returned, that error is printed.

9b ⟨*CGI Support Functions* 9b⟩≡ (8e) 9f▷

```
void init_cgi(void)
{
  ⟨Perform global CGI support initialization 10b⟩
}
```

9c ⟨*CGI State Dependencies* 9c⟩≡ (8g) 24e▷

```
struct cgi_state_t;
typedef NEOERR *(*cgi_callback_t)(struct cgi_state_t *cgi_state);
typedef void (*cgi_cleanup_cb)(struct cgi_state_t *cgi_state);
```

9d ⟨*(build.nw) Known Data Types* 9a⟩+≡ ◁9a 22d▷

```
cgi_callback_t,cgi_cleanup_cb,%
```

9e ⟨*CGI State Members* 9e⟩≡ (8g) 18c▷

```
cgi_callback_t callback;
cgi_cleanup_cb cleanup;
```

9f ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁9b 10a▷

```
static void *run_cgi_thread(void *_state)
{
  cgi_state_t *cgi_state = _state;
  NEOERR *nerr = (*cgi_state->callback)(cgi_state);
  if(cgi_state->cleanup)
    (*cgi_state->cleanup)(cgi_state);
  if(nerr != STATUS_OK) {
    ⟨Display error from CGI callback 18a⟩
  }
  ⟨Clean up after CGI callback 11c⟩
  return NULL;
}
```

10a ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁9f 10d▷

```
void run_cgi(cgi_callback_t callback)
{
  while (1) {
    cgi_state_t *cgi_state;

    ⟨Allocate CGI state 11f⟩
    cgi_state->callback = callback;
    ⟨Prepare for CGI thread 18d⟩
    ⟨Spawn CGI thread 10g⟩
  }
  ⟨Wait for remaining CGI threads 11d⟩
}
```

The most obvious global initializations are the memory pool for the CGI state structures, and the FastCGI subsystem.

10b ⟨*Perform global CGI support initialization* 10b⟩≡ (9b) 10f▷

```
#if GLIB_MINOR_VERSION < 10 /* GMemChunk is deprecated */
state_pool = g_mem_chunk_create(cgi_state_t, 16, G_ALLOC_AND_FREE);
#endif
FCGX_Init();
```

Rather than spawn a thread every time, which is what FastCGI is meant to avoid in the first place, a thread pool is used. The pool is generated in non-exclusive mode to prevent all of them from being created at once. GLib provides no control over how many idle threads to create initially. Only the maximum number of threads is configurable.

10c ⟨*CGI Support Variables* 10c⟩≡ (8e) 11a▷

```
static GThreadPool *cgi_thread_pool = NULL;
```

10d ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁10a 13b▷

```
static void run_cgi_thread_pool(gpointer data, gpointer user_data)
{
  run_cgi_thread(data);
}
```

10e ⟨*CGI Support Configuration* 7e⟩+≡ (149b) ◁7e 18b▷

```
# Maximum number of FastCGI threads
#max_cgi_threads = 64
```

10f ⟨*Perform global CGI support initialization* 10b⟩+≡ (9b) ◁10b 12e▷

```
cgi_thread_pool = g_thread_pool_new(run_cgi_thread_pool, NULL,
                                    getconf_int("max_cgi_threads", 64),
                                    FALSE, NULL);
```

10g ⟨*Spawn CGI thread* 10g⟩≡ (10a) 11b▷

```
g_thread_pool_push(cgi_thread_pool, cgi_state, NULL);
```

Keeping track of whether or not the threads are finished can be done several ways. The simplest is to just keep a count; in fact, the count can be retrieved from the pool. However, there is no signal associated with the pool's thread count, so waiting for the pool to empty is not possible. Instead, an integer thread counter is used, along with a lock and signal.

11a ⟨*CGI Support Variables* 10c⟩+≡ (8e) ◁10c 11e▷

```
static pthread_mutex_t cgi_state_lock = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cgi_state_sig = PTHREAD_COND_INITIALIZER;
static int cgi_thread_count = 0;
```

11b ⟨*Spawn CGI thread* 10g⟩+≡ (10a) ◁10g 12f▷

```
pthread_mutex_lock(&cgi_state_lock);
++cgi_thread_count;
pthread_mutex_unlock(&cgi_state_lock);
```

11c ⟨*Clean up after CGI callback* 11c⟩≡ (9f) 11g▷

```
pthread_mutex_lock(&cgi_state_lock);
--cgi_thread_count;
pthread_cond_signal(&cgi_state_sig);
pthread_mutex_unlock(&cgi_state_lock);
```

11d ⟨*Wait for remaining CGI threads* 11d⟩≡ (10a)

```
pthread_mutex_lock(&cgi_state_lock);
while (cgi_thread_count > 0)
  pthread_cond_wait(&cgi_state_sig, &cgi_state_lock);
pthread_mutex_unlock(&cgi_state_lock);
```

Allocating a CGI state from a pool requries locking. The thread count lock can be reused for this purpose.

11e ⟨*CGI Support Variables* 10c⟩+≡ (8e) ◁11a 12d▷

```
#if GLIB_MINOR_VERSION < 10 /* GMemChunk is deprecated */
static GMemChunk *cgi_state_pool;
#endif
```

11f ⟨*Allocate CGI state* 11f⟩≡ (10a)

```
#if GLIB_MINOR_VERSION < 10 /* GMemChunk is deprecated */
pthread_mutex_lock(&cgi_state_lock);
cgi_state = g_chunk_new0(cgi_state_t, cgi_state_pool);
pthread_mutex_unlock(&cgi_state_lock);
#else
cgi_state = g_slice_new0(cgi_state_t);
#endif
```

11g ⟨*Clean up after CGI callback* 11c⟩+≡ (9f) ◁11c

```
⟨Free CGI state members 18e⟩
#if GLIB_MINOR_VERSION < 10 /* GMemChunk is deprecated */
pthread_mutex_lock(&cgi_state_lock);
g_chunk_free(cgi_state, cgi_state_pool);
pthread_mutex_unlock(&cgi_state_lock);
#else
g_slice_free(cgi_state_t, cgi_state);
#endif
```

Since FastCGI programs are so similar to regular CGI programs, they should be expected to fall back to normal CGI behavior if invoked without FastCGI enabled. To support this, a global flag is used to distinguish the execution types, as well as macros to select the appropriate input and output funtions.

12a ⟨*(build.nw) Common C Includes* 8b⟩+≡ ◁8b 13a▷

```
#include <stdarg.h>
```

12b ⟨*CGI Support Global Definitions* 8g⟩+≡ (8d) ◁8g 21d▷

```
extern gboolean is_cgi;
#define cgi_puts(x) (is_cgi ? fputs(x, stdout) : \
                             FCGX_PutS(x, cgi_state->req.out))
#define cgi_putc(x) (is_cgi ? putchar(x) : \
                             FCGX_PutChar(x, cgi_state->req.out))
#define cgi_status(x) do { \
  if(!is_cgi) \
    FCGX_SetExitStatus(x, cgi_state->req.out); \
} while(0)
#define cgi_raw_gets(x, l) (is_cgi ? fgets(x, l, stdin) : \
                                     FCGX_GetLine(x, l, cgi_state->req.in))
#define cgi_raw_read(x, l) (is_cgi ? fread(x, 1, l, stdin) : \
                                     FCGX_GetStr(x, l, cgi_state->req.in))
#define cgi_write(x, l) (is_cgi ? fwrite(x, 1, l, stdout) : \
                                  FCGX_PutStr(x, l, cgi_state->req.out))
#define cgi_getenv(x) (is_cgi ? getenv(x) : \
                                FCGX_GetParam(x, cgi_state->req.envp))
#define cgi_printf(f, ...) (is_cgi ? printf(f, __VA_ARGS__) : \
                                     FCGX_FPrintF(cgi_state->req.out, f, \
                                                  __VA_ARGS__))
#define cgi_vprintf(f, a) (is_cgi ? vprintf(f, a) : \
                                    FCGX_VFPrintF(cgi_state->req.out, f, a))
```

12c ⟨*(build.nw) C Prototypes* 12c⟩≡ 21e▷

```
int cgi_puts(const char *str);
int cgi_putc(char c);
int cgi_write(const char *buf, int length);
int cgi_printf(const char *format, ...);
int cgi_vprintf(const char *fmt, va_list parms);
void cgi_status(int stat);
int cgi_raw_gets(char *buf, int length);
int cgi_raw_read(char *buf, int length);
const char *cgi_getenv(const char *name);
```

12d ⟨*CGI Support Variables* 10c⟩+≡ (8e) ◁11e 61b▷

```
gboolean is_cgi;
```

12e ⟨*Perform global CGI support initialization* 10b⟩+≡ (9b) ◁10f 61c▷

```
is_cgi = FCGX_IsCGI();
if(debug && is_cgi)
  setbuf(stdout, NULL);
```

12f ⟨*Spawn CGI thread* 10g⟩+≡ (10a) ◁11b

```
if(is_cgi)
  break;
```

Unfortunately, not all operations are possible in threads. In particular, if there is a need to change the process identity via `setuid`(2) or similar functions, POSIX threads will change the identity for all threads in the process at the same time. Older Linux versions were buggy in that respect and in fact provided separate identities per thread, and in fact may still do so via `setfsuid`(2), but programs should not depend on buggy behavior or Linux-specific functions that may not even be exposed in the C library.

Rather than maintain a separate process pool, it is expected that the callback will do a fork and wait for the subprocess to finish. A simple wrapper function is provided to make implementation easier. The wrapper will return `TRUE` if the child code is to be executed. It will not call `fork` for single-run CGI programs.

13a ⟨*(build.nw) Common C Includes* 8b⟩+≡ ◁12a 101b▷

```
#include <sys/wait.h>
```

13b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁10d 13c▷

```
gboolean cgi_fork(cgi_state_t *cgi_state)
{
  pid_t child_pid = is_cgi ? 0 : fork();

  if(child_pid < 0) {
    cgi_neo_error(cgi_state->cgi, nerr_raise_msg_errno("fork"));
  } else if(child_pid) {
    while(waitpid(child_pid, NULL, 0) < 0) {
      if(errno != EINTR && errno != EAGAIN) {
        cgi_neo_error(cgi_state->cgi, nerr_raise_msg_errno("fork"));
        break;
      }
    }
  }
  return !child_pid;
}
```

Sometimes it may be useful to print strings escaped. Rather than allocating a string, escaping into that memory, and then printing and freeing that memory, these functions print their parameters escaped. These functions are not in any which way Unicode compatible.

13c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁13b 13d▷

```
NEOERR *cgi_puts_url_escape(cgi_state_t *cgi_state, const char *s)
{
  const char *e;

  while(*s) {
    for(e = s; *e && (isalnum(*e) || *e == '.' || *e == '-' ||
                                      *e == '_' ||*e == '~'); e++);
    if(e != s)
      if(cgi_write(s, (int)(e - s)) <= 0)
        return nerr_raise_errno(NERR_IO, "CGI Write");
    if(*e == ' ') {
      if(cgi_putc('+') == EOF)
        return nerr_raise_errno(NERR_IO, "CGI Write");
    } else
      if(cgi_printf("%%%02X", (int)*e) <= 0)
        return nerr_raise_errno(NERR_IO, "CGI Write");
    s = e + 1;
  }
  return STATUS_OK;
}
```

13d ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁13c 14▷

```
NEOERR *cgi_puts_html_escape(cgi_state_t *cgi_state, const char *s,
                             gboolean is_js)
{
  const char *e;
  int ret;

  while(*s) {
    for(e = s; *e && (isgraph(*e) || *e == ' ' || *e == '\n') &&
                *e != '<' && *e != '>' && *e != '&' && *e != '"' && *e != '\'' &&
                (*e != '\\' || !is_js);
        e++);
    if(e != s)
      if(cgi_write(s, (int)(e - s)) < 0)
        return nerr_raise_errno(NERR_IO, "CGI Write");
    if(!*e)
      break;
    if(*e == '<')
      ret = cgi_puts("&lt;");
    else if(*e == '>')
      ret = cgi_puts("&gt;");
    else if(*e == '&')
      ret = cgi_puts("&amp;");
#if 0  /* &#34; shorter & more portable than &quot; - same for ' and &apos; */
    else if(*e == '"')
      ret = cgi_puts("&quot;");
    else if(*e == '\'')
      ret = cgi_puts("&apos;");
#endif
    else if(*e == '\\')
      ret = cgi_puts("\\\\");
    else
      ret = cgi_printf("&#%02X;", (int)*e) - 1; /* <= 0, not just < 0 -> err */
    if(ret < 0)
      return nerr_raise_errno(NERR_IO, "CGI Write");
    s = e + 1;
  }
  return STATUS_OK;
}
```

For building strings, GLib provides `g_string_append_uri_escaped`, but no equivalent for HTML escaping. Actually, before 2.16, it does not provide either. Since for most uses, the default behavior is fine, no support is provided for `reserved_chars_allowed` or `allow_utf8` in the reimplementation of `g_string_append_uri_escaped`.

14 ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁13d 15a▷

```
GString *g_string_append_html_escaped(GString *buf, const char *s,
                                      gboolean is_js)
{
  const char *e;

  while(*s) {
    for(e = s; *e && isgraph(*e) &&
                *e != '<' && *e != '>' && *e != '&' && *e != '"' && *e != '\'' &&
                (*e != '\\' || !is_js);
        e++);
    if(e != s)
      g_string_append_len(buf, s, (int)(e - s));
    if(!*e)
      break;
    if(*e == '<')
      g_string_append(buf, "&lt;");
    else if(*e == '>')
```

14 ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁13d 15a▷

```
      g_string_append(buf, "&gt;");
    else if(*e == '&')
      g_string_append(buf, "&amp;");
#if 0  /* &#34; shorter & more portable than &quot; - same for ' and &apos; */
    else if(*e == '"')
      g_string_append(buf, "&quot;");
    else if(*e == '\'')
      g_string_append(buf, "&apos;");
#endif
    else if(*e == '\\')
      g_string_append(buf, "\\\\");
    else
      g_string_append_printf(buf, "&#%02X;", (int)*e);
    s = e + 1;
  }
  return buf;
}
```

15a ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁14 15b▷

```
#if !GLIB_CHECK_VERSION(2,16,0)
GString *g_string_append_uri_escaped(GString *buf, const char *s,
                                     const char *reserved,
                                     gboolean utf8)
{
  const char *e;

  while(*s) {
    for(e = s; *e && (isalnum(*e) || *e == '.' || *e == '-' ||
                                     *e == '_' ||*e == '~'); e++);
    if(e != s)
      g_string_append_len(buf, s, (int)(e - s));
    if(*e == ' ')
      g_string_append_c(buf, '+');
    else
      g_string_append_printf(buf, "%%%02X", (int)*e);
    s = e + 1;
  }
  return buf;
}
#endif
```

JSON encoding is provided already for strings, but converting to a string before writing to CGI output may be inefficient for large objects and arrays. For this reason, a function is provided to use the string version for the leaf values, but to output the object and array delimiters directly. This limits the total size of dynamically allocated string buffers. Since reading JSON from a CGI stream will never happen (CGI input is always parsed as CGI parameters), no equivalent input function is provided.

15b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁15a 16b▷

```
NEOERR *cgi_output_json(cgi_state_t *cgi_state, HDF *hdf)
{
  GString *buf;
  NEOERR *nerr = STATUS_OK;

#define io_nerr_op(op) do { \
  if(nerr == STATUS_OK && op < 0) \
    nerr = nerr_raise_msg_errno("JSON output"); \
} while(0)
  if(hdf_obj_child(hdf)) {
    json_var_type vt = hdf_json_var_type(hdf);
```

15b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁15a 16b▷

```
    if (vt == JSON_ARRAY) {
      char c = '[';
      hdf_sort_obj(hdf, comp_hdf_name);
      for (hdf = hdf_obj_child(hdf); hdf; hdf = hdf_obj_next(hdf)) {
        io_nerr_op(cgi_putc(c));
        c = ',';
        nerr_op(cgi_output_json(cgi_state, hdf));
      }
      io_nerr_op(cgi_putc(']'));
    } else {
      buf = g_string_new("");
      char c = '{';
      for (hdf = hdf_obj_child(hdf); hdf; hdf = hdf_obj_next(hdf)) {
        io_nerr_op(cgi_putc(c));
        c = ',';
        g_string_truncate(buf, 0);
        append_unicode_quoted(buf, hdf_obj_name(hdf));
        io_nerr_op(cgi_puts(buf->str));
        io_nerr_op(cgi_putc(':'));
        nerr_op(cgi_output_json(cgi_state, hdf));
      }
      io_nerr_op(cgi_putc('}'));
      g_string_free(buf, TRUE);
    }
    return nerr;
  }
  buf = g_string_new("");
  g_string_append_json(buf, hdf);
  cgi_puts(buf->str);
  g_string_free(buf, TRUE);
  return nerr;
}
```

Any errors that occur should be printed to the user, but not with the standard error reporting mechanism. Instead, they should be reported via the CGI output stream. Unfortunately, that is impossible during initial setup for FastCGI programs, so a dummy `NULL` CGI state indicates that regular output should be used instead. If the CGI state exists, it is assumed that the call is from a NEOERR returning function, so the error is simply returned. This breaks usage in the mainline even though it never gets invoked, so `DIE_IF_ERR` is toggled from the error to a static 1 in the mainline. Once an error is returned from the primary callback, it is displayed to the user.

16a ⟨*CGI Message Overrides* 16a⟩≡ (16b 148b)

```
#undef die_if_err
#define die_if_err(err) do { \
  NEOERR *_err = err; \
 \
  if (_err != STATUS_OK) { \
    if (cgi_state) \
      return DIE_IF_ERR; \
    else if (!is_cgi) \
      nerr_log_error(_err); \
    else \
      cgi_log_error_stdout(_err); \
    exit(1); \
  } \
} while (0)
#define DIE_IF_ERR _err
```

16b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁15b 17c▷

⟨*CGI Message Overrides* 16a⟩

17a ⟨*CGI Mainline Variables* 17a⟩≡ (148b)

```
⟨(cs-glib.nw) Common Mainline Variables (imported)⟩
cgi_state_t *cgi_state unused_attr = NULL;
#undef DIE_IF_ERR
#define DIE_IF_ERR 1
```

17b ⟨*CGI Per-Run Variables* 17b⟩≡ (148b)

⟨*(cs-glib.nw) Common Mainline Variables* (imported)⟩

The error itself is printed as HTML or HTTP text, from a template. The HTTP header can be suppressed by setting the `header_printed` variable to non-blank in the CGI state's HDF. The default status of 500 can also be overridden by setting the `error_status` variable. The text to print can also be overridden by a template by setting the `error_tmpl` variable, which may use the error text in `error_text`. If the HTML tags have already been printed, the XML will become invalid, and if they haven't, the document type will not be printed. In either case, most browsers will print the message anyway, and don't care. Note that in addition to the ability to override the header, this differs from the standard kit's `cgi_neo_error` in that it will HTML-escape the message.

17c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁16b 19a▷

```
void cgi_log_error_stdout(NEOERR *nerr)
{
  is_cgi = TRUE;
  cgi_log_error(nerr, NULL, local_config);
}

void cgi_log_error(NEOERR *nerr, cgi_state_t *cgi_state, HDF *hdf)
{
  cgi_state_t _cgi_state;
  if(!cgi_state) {
    cgi_state = &_cgi_state;
    cgi_state->hdf = hdf;
  }
  if(!*cgi_env_val("header_printed", "")) {
    const char *status = cgi_env_val("error_status", NULL);
    if(!status || !*status)
      status = "500 Internal Server Error";
    cgi_status(atoi(status));
    cgi_puts("Content-type: text/html\nStatus: ");
    cgi_puts(status);
    cgi_puts("\n\n");
  }
  ⟨(cs-glib.nw) Convert nerr to err_str (imported)⟩
  const char *tmpl = cgi_env_val("error_tmpl", NULL);
  if(tmpl && *tmpl) {
    NEOERR *nerr2 = cgi_env_set("error_text", err_str.buf);
    if(nerr2 == STATUS_OK)
      nerr2 = tmpl_to_cgi(tmpl, cgi_state, FALSE);
    if(nerr2 != STATUS_OK) {
      tmpl = NULL;
      nerr_ignore(&nerr2);
    }
  }
  if(!tmpl || !*tmpl) {
    cgi_puts("<html><body>An error occurred:<pre>\n");
    cgi_puts_html_escape(cgi_state, err_str.buf, FALSE);
```

17c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁16b 19a▷

```
    cgi_puts("</pre></body></html>");
  }
  string_clear(&err_str);
}
```

18a ⟨*Display error from CGI callback* 18a⟩≡ (9f)

```
cgi_log_error(nerr, cgi_state, NULL);
```

18b ⟨*CGI Support Configuration* 7e⟩+≡ (149b) ◁10e 61a▷

```
# Set to non-blank to override message to print on errors
# This is evaluated as a ClearSilver template in the CGI environment; the
# text of the error message is in the error_text variable.
# The default message prints:
#    <html><body><h3>An error occurred while processing your request:</h3><pre>
#    <?cs var:error_text ?></pre></body></html>
#error_tmpl = <html><body><h3>An error occurred while processing ...

# Set to non-blank to override the status to return on errors
#error_status = 500 Internal Server Error

# Set to non-blank to disable printing the HTTP header for error messages
# This is usually set by any template which prints an HTTP header.
#header_printed =
```

The only thing that really needs to be prepared for the CGI thread other than allocating the CGI state is to wait for a CGI request and assign the appropriate information to the FastCGI request structure. For plain CGI requests, there is no need to fill in a request at all.

18c ⟨*CGI State Members* 9e⟩+≡ (8g) ◁9e 20b▷

```
FCGX_Request req;
```

18d ⟨*Prepare for CGI thread* 18d⟩≡ (10a)

```
if(!is_cgi) {
  FCGX_InitRequest(&cgi_state->req, 0, 0);
  if(FCGX_Accept_r(&cgi_state->req) < 0) {
    pthread_mutex_lock(&cgi_state_lock);
    --cgi_thread_count;
#if GLIB_MINOR_VERSION < 10 /* GMemChunk is deprecated */
    g_chunk_free(cgi_state, cgi_state_pool);
#else
    g_slice_free(cgi_state_t, cgi_state);
#endif
    pthread_mutex_unlock(&cgi_state_lock);
    break;
  }
}
```

18e ⟨*Free CGI state members* 18e⟩≡ (11g) 21b▷

```
if(!is_cgi) {
  FCGX_Finish_r(&cgi_state->req);
  /* FCGX_Free(&cgi_state->req); */ /* freeing supposedly done by Finish */
}
```

Support for macros in FastCGI programs is partially provided by the HDF forms of the template processing routines. However, the ones providing just file output are inappropriate. Instead, CGI output versions are provided.

19a ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁17c 19b▷

```
NEOERR *cs_to_cgi(void *user, char *str)
{
  cgi_state_t *cgi_state = user;

  return cgi_puts(str) < 0 ? nerr_raise_msg_errno("Writing") : STATUS_OK;
}
```

19b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁19a 19c▷

```
NEOERR *tmpl_to_cgi_hdf(const char *tmpl, HDF *parms, cgi_state_t *cgi_state,
                        gboolean raw)
{
  return nerr_pass(tmpl_string_to(tmpl, parms, cs_to_cgi, cgi_state, raw));
}

NEOERR *tmpl_to_cgi(const char *tmpl, cgi_state_t *cgi_state, gboolean raw)
{
  return nerr_pass(tmpl_to_cgi_hdf(tmpl, cgi_state->hdf, cgi_state, raw));
}
```

19c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁19b 20a▷

```
NEOERR *tmpl_file_to_cgi_hdf(const char *tmpl, HDF *parms,
                             cgi_state_t *cgi_state, gboolean raw)
{
  return nerr_pass(tmpl_file_to(tmpl, parms, cs_to_cgi, cgi_state, raw));
}

NEOERR *tmpl_file_to_cgi(const char *tmpl, cgi_state_t *cgi_state, gboolean raw)
{
  return nerr_pass(tmpl_file_to_cgi_hdf(tmpl, cgi_state->hdf, cgi_state, raw));
}
```

Chapter 4

CGI Parameter Parsing

Once in the thread, though, the real work starts. A function is provided to do most of the initialization work. First, the CGI kit environment must be initialized. This includes setting the function overrides that will enable FastCGI. The `putenv` and `iterenv` functions are not provided; this means that the CGI kit's `debug_init` for reading parameters from configuration files will not work, and the `cgi_output` and `cgi_display` functions will not work with debugging turned on. The CGI parameters will be merged with the global program-wide parameters, so a copy of those global parameters must be made to pass into the kit.

20a ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁19c 21a▷

```
static int read_fcgi(void *parm, char *data, int len)
{
  cgi_state_t *cgi_state = parm;
  ⟨Read data from CGI stream 29c⟩
}

static int writef_fcgi(void *parm, const char *fmt, va_list args)
{
  cgi_state_t *cgi_state = parm;
  return cgi_vprintf(fmt, args);
}

static int write_fcgi(void *parm, const char *data, int len)
{
  cgi_state_t *cgi_state = parm;
  return cgi_write(data, len);
}

static char *getenv_fcgi(void *parm, const char *name)
{
  cgi_state_t *cgi_state = parm;
  /* NOTE: return value *must* be malloc'd */
  const char *s = cgi_getenv(name);
  if(s)
    return strdup(s);
  else
    return NULL;
}
```

20b ⟨*CGI State Members* 9e⟩+≡ (8g) ◁18c 21c▷

```
HDF *hdf;
CGI *cgi;
```

21a ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁20a 24b▷

```
NEOERR *init_cgi_run(cgi_state_t *cgi_state)
{
  NEOERR *nerr = STATUS_OK;
  if(!nerr_op_ok(hdf_init(&cgi_state->hdf)))
    return nerr;
  if(!nerr_op_ok(hdf_copy(cgi_state->hdf, NULL, local_config)))
    return nerr;
  cgiwrap_init_emu(cgi_state, read_fcgi, writef_fcgi, write_fcgi, getenv_fcgi,
                   NULL, NULL);
  if(!nerr_op_ok(cgi_init(&cgi_state->cgi, cgi_state->hdf)))
    return nerr;
  cgi_state->hdf = cgi_state->cgi->hdf; /* in case it changed */
  ⟨Initialize this CGI run 22a⟩
  return nerr;
}
```

21b ⟨*Free CGI state members* 18e⟩+≡ (11g) ◁18e 31a▷

```
⟨Perform cleanups requiring valid CGI parameters 141e⟩
if(cgi_state->cgi)
  cgi_destroy(&cgi_state->cgi);
else if(cgi_state->hdf)
  hdf_destroy(&cgi_state->hdf);
```

For convenience, the CGI parameters and cookies are stored separately in the CGI state structure. If none were created by `cgi_init` from the environment, empty HDF entries will be created. A few macros are provided as well for quick access.

21c ⟨*CGI State Members* 9e⟩+≡ (8g) ◁20b 22e▷

```
HDF *cgi_parms, *cookies;
```

21d ⟨*CGI Support Global Definitions* 8g⟩+≡ (8d) ◁12b 22b▷

```
#define cgi_env_val(n, d) hdf_get_value(cgi_state->hdf, n, d)
#define cgi_env_val_int(n, d) hdf_get_int64_value(cgi_state->hdf, n, d)
#define cgi_env_set(n, v) hdf_set_value(cgi_state->hdf, n, v)
#define cgi_env_set_int(n, v) hdf_set_int64_value(cgi_state->hdf, n, v)
#define cgi_env_obj(n) hdf_get_obj(cgi_state->hdf, n)
#define cgi_parm_val(n, d) hdf_get_value(cgi_state->cgi_parms, n, d)
#define cgi_parm_val_int(n, d) hdf_get_int64_value(cgi_state->cgi_parms, n, d)
#define cgi_parm_obj(n) hdf_get_obj(cgi_state->cgi_parms, n)
#define cgi_parm_set(n, v) hdf_set_value(cgi_state->cgi_parms, n, v)
#define cgi_parm_set_int(n, v) hdf_set_int64_value(cgi_state->cgi_parms, n, v)
#define cgi_cookie_val(n, d) hdf_get_value(cgi_state->cookies, n, d)
#define cgi_cookie_val_int(n, d) hdf_get_int64_value(cgi_state->cookies, n, d)
#define cgi_cookie_obj(n) hdf_get_obj(cgi_state->cookies, n)
#define cgi_cookie_set(n, v) hdf_set_value(cgi_state->cookies, n, v)
#define cgi_cookie_set_int(n, v) hdf_set_int64_value(cgi_state->cookies, n, v)
```

21e ⟨*(build.nw) C Prototypes* 12c⟩+≡ ◁12c 22c▷

```
const char *cgi_env_val(const char *name, const char *def);
gint64 cgi_env_val_int(const char *name, gint64 def);
NEOERR *cgi_env_set(const char *name, const char *value);
NEOERR *cgi_env_set_int(const char *name, gint64 value);
HDF *cgi_env_obj(const char *name);
const char *cgi_parm_val(const char *name, const char *def);
gint64 cgi_parm_val_int(const char *name, gint64 def);
HDF *cgi_parm_obj(const char *name);
```

21e ⟨*(build.nw) C Prototypes* 12c⟩+≡ ◁12c 22c▷

```
NEOERR *cgi_parm_set(const char *name, const char *value);
NEOERR *cgi_parm_set_int(const char *name, gint64 value);
const char *cgi_cookie_val(const char *name, const char *def);
gint64 cgi_cookie_val_int(const char *name, gint64 def);
HDF *cgi_cookie_obj(const char *name);
NEOERR *cgi_cookie_set(const char *name, const char *value);
NEOERR *cgi_cookie_set_int(const char *name, gint64 value);
```

22a ⟨*Initialize this CGI run* 22a⟩≡ (21a) 22f▷

```
if(!nerr_op_ok(hdf_get_node(cgi_state->hdf, "Query", &cgi_state->cgi_parms)))
  return nerr;
if(!nerr_op_ok(hdf_get_node(cgi_state->hdf, "Cookie", &cgi_state->cookies)))
  return nerr;
```

In addition, some looping macros are provided to support the standard method of storing a single-valued CGI parameter as just a value, and a multi-valued parameter as children.

22b ⟨*CGI Support Global Definitions* 8g⟩+≡ (8d) ◁21d 23a▷

```
#define for_each_cgi_parm_obj(obj, f, ...) do { \
  if(hdf_obj_child(obj)) { \
    for(obj = hdf_obj_child(obj); obj && nerr == STATUS_OK; \
                                        obj = hdf_obj_next(obj)) \
      f(obj, __VA_ARGS__); \
  } else \
    f(obj, __VA_ARGS__); \
} while(0)
#define for_each_cgi_parm(p, f, ...) do { \
  HDF *__p = cgi_parm_obj(p); \
  for_each_cgi_parm_obj(__p, f, __VA_ARGS__); \
} while(0)
```

22c ⟨*(build.nw) C Prototypes* 12c⟩+≡ ◁21e 23b▷

```
void (*for_each_cb)(HDF *obj, ...);
void for_each_cgi_parm_obj(HDF *obj, for_each_cb f, ...);
void for_each_cgi_parm(const char *name, for_each_cb f, ...);
```

22d ⟨*(build.nw) Known Data Types* 9a⟩+≡ ◁9d 24h▷

```
for_each_cb,%
```

There are a number of HTTP headers which the CGI kit does not turn into HDF parameters, but which are relevant. Instead of picking headers to add, all HTTP headers are added as HDF parameters. Case is converted the same way as the standard ones, and all are added under the HTTP hierarchy. The only one that will end up duplicated is the differently named `HTTP_SOAPACTION` translation to Soap.Action. For convenience, the HTTP headers are also stored in the CGI state directly.

22e ⟨*CGI State Members* 9e⟩+≡ (8g) ◁21c 23c▷

```
HDF *headers;
```

22f ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁22a 23d▷

```
GString *header_env = g_string_new("");
/* const */ char **envp;
/* NOTE: POSIX says environ should be in unistd.h, but GNU considers it */
/* an extension.  It could also come from 3rd main() parm, but adding */
/* that to CGI apps would cause cproto to break apps that don't use it */
extern char **environ;
for(envp = is_cgi ? environ : cgi_state->req.envp; *envp; envp++)
  if(!strncmp(*envp, "HTTP_", 5)) {
    const char *s;
    char *d;
    gboolean firstc = TRUE;
    g_string_assign(header_env, *envp);
    header_env->str[4] = '.';
    for(s = d = header_env->str + 5; *s && *s != '='; s++, d++) {
      if(*s == '_') {
        firstc = TRUE;
        d--;
      } else if(firstc) {
        firstc = FALSE;
        *d = toupper(*s);
      } else
        *d = tolower(*s);
    }
    if(*s == '=') {
      *d = 0;
      if(!cgi_env_obj(header_env->str))
        nerr_op(cgi_env_set(header_env->str, s + 1));
    }
  }
g_string_free(header_env, TRUE);
nerr_op(hdf_get_node(cgi_state->hdf, "HTTP", &cgi_state->headers));
if(nerr != STATUS_OK)
  return nerr;
```

23a ⟨*CGI Support Global Definitions* 8g⟩+≡ (8d) ◁22b 28c▷

```
#define cgi_header_obj(n) hdf_get_obj(cgi_state->headers, n)
#define cgi_header_val(n, d) hdf_get_value(cgi_state->headers, n, d)
#define cgi_header_val_int(n, d) hdf_get_int64_value(cgi_state->headers, n, d)
#define cgi_header_set(n, v) hdf_set_value(cgi_state->headers, n, v)
#define cgi_header_set_int(n, v) hdf_set_int64_value(cgi_state->headers, n, v)
```

23b ⟨*(build.nw) C Prototypes* 12c⟩+≡ ◁22c 29a▷

```
HDF *cgi_header_obj(const char *name);
const char *cgi_header_val(const char *name, const char *def_val);
gint64 cgi_header_val_int(const char *name, gint64 def_val);
NEOERR *cgi_header_set(const char *name, const char *val);
NEOERR *cgi_header_set_int(const char *name, gint64 val);
```

A few other shortcuts could be set in the CGI state as well. For now, only the user ID is stored.

23c ⟨*CGI State Members* 9e⟩+≡ (8g) ◁22e 24c▷

```
const char *userid;
```

23d ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁22f 24a▷

```
cgi_state->userid = cgi_env_val("CGI.RemoteUser", NULL);
```

After all of the CGI parameters have been parsed, it might be useful to print them. This is only done if debugging is enabled. Although this prints an HTTP header, the `header_printed` variable is not set, so that the intended headers are displayed in the debugging output as well.

24a ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁23d 24d▷

```
if(debug) {
  cgi_status(200);
  cgi_puts("Status: 200\nContent-type: text/html\n\n"
           "<html><head><title>Debug output</title></head><body>");
  cgi_dump_hdf(cgi_state);
}
```

24b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁21a 24f▷

```
void cgi_dump_hdf(cgi_state_t *cgi_state)
{
  NEOERR *nerr;
  STRING ps;

  cgi_puts("CGI Parameters:<pre>\n");
  string_init(&ps);
  nerr = hdf_dump_str(cgi_state->hdf, "", 2, &ps);
  if(nerr == STATUS_OK && ps.buf)
    cgi_puts_html_escape(cgi_state, ps.buf, FALSE);
  else
    nerr_ignore(&nerr);
  string_clear(&ps);
  cgi_puts("--------</pre>\n");
}
```

Traditional CGI programs only need to deal with a few request methods; usually just GET and POST, and maybe HEAD. However, WebDAV adds a whole bunch of new methods, so this library must support dealing with a large number of methods. In order to avoid repeated string comparisons, the method is converted into an integer code, stored in the CGI state, if it is one of the known methods.

24c ⟨*CGI State Members* 9e⟩+≡ (8g) ◁23c 28a▷

```
http_req_t req_method;
```

24d ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁24a 28b▷

```
const char *method = cgi_env_val("CGI.RequestMethod", "GET");
http_req_t req_method = http_req_id(method, strlen(method));
cgi_state->req_method = req_method;
```

24e ⟨*CGI State Dependencies* 9c⟩+≡ (8g) ◁9c 44a▷

```
#include "cgi.h.gperf"
```

24f ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁24b 25c▷

```
#include "cgi.c.gperf"
```

24g ⟨*cgi-gperf-nc-http_req* 24g⟩≡

```
⟨Known HTTP Requests 25b⟩
```

24h ⟨*(build.nw) Known Data Types* 9a⟩+≡ ◁22d 44b▷

```
http_req_t,%
```

25a ⟨*(build.nw) makefile.rules* 25a⟩≡ 151b▷

```
cgi.h: cgi.h.gperf
cgi.c: cgi.c.gperf
```

Methods known and possibly dealt with here are:

RFC 2616[1] HTTP: OPTIONS GET HEAD POST PUT DELETE TRACE CONNECT

RFC 4918[2] WebDAV: PROPFIND PROPPATCH MKCOL COPY MOVE LOCK UNLOCK

RFC 3253[3] WebDAV Versioning: REPORT VERSION-CONTROL CHECKOUT CHECKIN UNCHECKOUT MKWORKSPACE UPDATE LABEL MERGE BASELINE-CONTROL MKACTIVITY

RFC 3648[4] WebDAV Ordered Collections: ORDERPATCH

RFC 3744[5] WebDAV Access Control: ACL

RFC 4437[6] WebDAV Redirect References: MKREDIRECTREF UPDATEREDIRECTREF

RFC 4791[7] WebDAV Calendaring: MKCALENDAR

RFC 5323[8] WebDAV Search: SEARCH

Any methods not covered here can be added by extending ⟨*Known HTTP Requests* 25b⟩.

25b ⟨*Known HTTP Requests* 25b⟩≡ (24g)

```
OPTIONS GET HEAD POST PUT DELETE TRACE CONNECT
PROPFIND PROPPATCH MKCOL COPY MOVE LOCK UNLOCK
REPORT VERSION-CONTROL CHECKOUT CHECKIN UNCHECKOUT MKWORKSPACE UPDATE LABEL
MERGE BASELINE-CONTROL MKACTIVITY
ORDERPATCH
ACL
MKREDIRECTREF UPDATEREDIRECTREF
MKCALENDAR
SEARCH
```

Most requests take paths as their argument: either from the request URI (actually `PATH_INFO`), from headers, or from CGI parameters. All of these paths must be normalized before real use. One way of normalizing them is to just use the underlying file system to resolve to an absolute path. Another is to normalize just the path itself. A normalization function is provided to transform an HDF object's value into a path similar to `PATH_INFO`: an absolute path with the CGI program information optionally removed. This function takes the path it may be relative to as an argument; setting that argument to NULL means that the value must be either an absolute path or an absolute URL. No URL detection takes place if the path may be relative. Also, if a relative parent is provided, it must be normalized. If it is a directory, it must have a terminating slash. Otherwise, the trailing component is removed for relative references. Any further filesystem-specific normalization, such as home directory and soft link resolution, must be done by the applicaiton program.

25c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁24f 29b▷

```
NEOERR *normalize_path_obj(HDF *obj, cgi_state_t *cgi_state,
                           const char *relative_to, gboolean utf_8,
                           gboolean strip_cgi_script)
{
  NEOERR *nerr = STATUS_OK;
  const char *oval = hdf_obj_value(obj);

  ⟨Normalize path obj/oval UTF-8 26a⟩
  if(*oval != '/') {
    if(relative_to) {
      ⟨Prepend relative_to to obj/oval 26b⟩
      /* relative_to is assumed to be already stripped */
      strip_cgi_script = FALSE;
    } else {
      ⟨Strip URL from obj/oval 26c⟩
    }
  }
  if(strip_cgi_script) {
    ⟨Strip CGI script from obj/oval 27a⟩
  }
  ⟨Strip . and .. from obj/oval 27b⟩
  return nerr;
}
```

26a ⟨*Normalize path* obj/oval *UTF-8* 26a⟩≡ (25c)

```
if(utf_8) {
  char *nv = g_utf8_normalize(oval, -1, G_NORMALIZE_DEFAULT);
  if(!nv)
    return nerr_raise_msg("Invalid UTF-8 path");
  if(!strcmp(nv, oval))
    g_free(nv);
  else {
    hdf_set_buf(obj, NULL, nv);
    oval = nv;
  }
}
```

26b ⟨*Prepend* relative_to *to* obj/oval 26b⟩≡ (25c)

```
GString *full_path = g_string_new(relative_to);
char *rtls = strrchr(full_path->str, '/');
if(rtls)
  g_string_truncate(full_path, (int)(rtls - full_path->str));
g_string_append_c(full_path, '/');
g_string_append(full_path, oval);
hdf_set_buf(obj, NULL, g_string_free(full_path, FALSE));
```

26c ⟨*Strip URL from* obj/oval 26c⟩≡ (25c)

```
gboolean is_https = !!cgi_env_val("CGI.HTTPS", NULL);
const char *cmpstr = cgi_header_val("Host", "localhost");
int cmplen = strlen(cmpstr);
if(strncmp(oval, "http", 4) || (is_https && (++oval)[4] != 's') ||
   strncmp((oval += 4), "://", 3) ||
   strncasecmp((oval += 3), cmpstr, cmplen))
  return nerr_raise_msg("Invalid URL");
oval += cmplen;
if(*oval == ':') {
  cmpstr = cgi_header_val("CGI.ServerPort", is_https ? "443" : "80");
  cmplen = strlen(cmpstr);
  if(strncasecmp(oval + 1, cmpstr, cmplen))
```

26c ⟨*Strip URL from* `obj/oval` 26c⟩≡ (25c)

```
    return nerr_raise_msg("Invalid URL");
  oval += cmplen + 1;
}
⟨Strip CGI script from obj/oval 27a⟩
if(*oval != '/')
  return nerr_raise_msg("Invalid URL");
```

27a ⟨*Strip CGI script from* `obj/oval` 27a⟩≡ (25c 26c)

```
const char *scrn = cgi_header_val("CGI.ScriptName", "");
int scrlen = strlen(scrn);
if(strncmp(oval, scrn, scrlen))
  return nerr_raise_msg("Invalid URL");
oval += scrlen;
if(!nerr_op_ok(hdf_set_value(obj, NULL, oval)))
  return nerr;
oval = hdf_obj_value(obj);
```

27b ⟨*Strip* `.` *and* `..` *from* `obj/oval` 27b⟩≡ (25c)

```
char *s = strchr(oval, '/');

if(s && (s[1] == '/' ||
         (s[1] == '.' && (!s[2] || s[2] == '/' ||
                          (s[2] == '.' && (!s[3] || s[3] == '/')))))) {
  char *d = s;
  do {
    if(s[1] == '/')
      s++;
    else if(s[2] != '.')
      s += 2;
    else if(d == oval)
      return nerr_raise_msg("Invalid relative path (..)");
    else {
      while(--d > oval && *d != '/');
      s += 3;
    }
    while(*s && (*s != '/' || (s[1] != '/' &&
                               (s[1] != '.' ||
                                (s[2] && s[2] != '/' &&
                                 (s[2] != '.' || (s[3] &&
                                                  s[3] != '/')))))))
     *d++ = *s++;
  } while(*s);
}
```

Chapter 5

Content Decoding

Normally, the CGI input stream is read directly, using e.g. `fread`. However, reading data from is not always that simple. For apache, at least, transport encoding is taken care of by the web server[1], but content encoding is not. There may be value in saving the encoded stream directly, if the CGI program will just end up serving the same data encoded the same way, but most applications want their data decoded. There are several things that can be done about this. First of all, the content encoding and content length are stored in the CGI state for quick reference. Since the content length might not be accurate due to apache's losing it with chunked transfer-encoding, execution from the command line, or other reasons, a zero length is not trusted. If the content length is zero, a single character is read from the input stream; if it wasnt end-of-file, then content length is set to -1 to indicate data is present, but of unknown length.

28a ⟨*CGI State Members* 9e⟩+≡ (8g) ◁24c 30b▷

```
const char *body_enc;
gint64 body_len;
```

28b ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁24d 34c▷

```
cgi_state->body_enc = cgi_header_val("ContentEncoding", NULL);
if(cgi_state->body_enc && (!*cgi_state->body_enc ||
                           !strcmp(cgi_state->body_enc, "identity")))
  cgi_state->body_enc = NULL;
cgi_state->body_len = cgi_env_val_int("CGI.ContentLength", 0);
if(!cgi_state->body_len) {
  int c = is_cgi ? getchar() : FCGX_GetChar(cgi_state->req.in);
  if(c != EOF) {
    cgi_env_set_int("CGI.ContentLength", (cgi_state->body_len = -1));
    if(is_cgi)
      ungetc(c, stdin);
    else
      FCGX_UnGetChar(c, cgi_state->req.in);
  }
}
```

Next, the standard encoding methods should be supported. A new set of input functions and macros is used to select the decoded stream. The first call initializes the decoder, and the final cleanup cleans it up.

28c ⟨*CGI Support Global Definitions* 8g⟩+≡ (8d) ◁23a 64a▷

```
#define cgi_gets(s, l) cgi_decode_gets(cgi_state, s, l)
#define cgi_read(s, l, rl) cgi_decode_read(cgi_state, s, l, rl)
```

[1] However, at least in apache 2.2.8, using chunk encoding causes the content length and any headers in the last chunk to be lost.

29a ⟨*(build.nw) C Prototypes* 12c⟩+≡ ◁23b 107b▷

```
NEOERR *cgi_gets(char *buf, int length);
NEOERR *cgi_read(char *buf, int length, int *retlength);
```

29b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁25c 31c▷

```
NEOERR *cgi_decode_gets(cgi_state_t *cgi_state, char *buf, int length)
{
  NEOERR *nerr;
  if(!cgi_state->body_enc) {
    if(!cgi_raw_gets(buf, length))
      return nerr_raise_msg_errno("Error reading body");
    else
      return STATUS_OK;
  }
  ⟨Initialize body decoder 30c⟩
  ⟨Decode a string into buf/length 33⟩
}

NEOERR *cgi_decode_read(cgi_state_t *cgi_state, char *buf, int length,
                        int *retlength)
{
  int ret;
  NEOERR *nerr;
  if(!cgi_state->body_enc) {
    *retlength = ret = cgi_raw_read(buf, length);
    if(ret < 0)
      return nerr_raise_msg_errno("Error reading body");
    else
      return STATUS_OK;
  }
  ⟨Initialize body decoder 30c⟩
  ⟨Decode data into buf/length 32c⟩
}
```

29c ⟨*Read data from CGI stream* 29c⟩≡ (20a)

```
NEOERR *nerr = cgi_read(data, len, &len);
if(nerr == STATUS_OK)
  return len;
nerr_ignore(&nerr);
return -1;
```

The gzip, x-gzip, and deflate methods can be handled by zlib[2]. There is no simple library which supports the compress method, but beginning with version 2.8, libarchive[3] can be used for this. This was originally tested using 2.7.902a; thus the version barrier uses this number rather than 2.8. The undefined bzip2 methods are handled by libarchive as well. Both of these methods require a separate buffer for I/O; zlib needs one for input and one for output, and libarchive allocates output buffers internally, so it only needs one for input.

29d ⟨*(build.nw) makefile.vars* 7g⟩+≡ ◁8f 61d▷

```
EXTRA_LDFLAGS += -lz $(LIBAR_LDFLAGS) -larchive
EXTRA_CFLAGS += $(LIBAR_CFLAGS)
```

[2] http://www.zlib.net
[3] http://people.freebsd.org/~kientzle/libarchive

29e ⟨*(build.nw) makefile.config* 7f⟩+≡ ◁7f 113d▷

```
# Extra flags needed to find libarchive 2.8
LIBAR_CFLAGS = -I/usr/local/include
LIBAR_LDFLAGS = -L/usr/local/lib
```

30a ⟨*CGI Prerequisite Headers* 8c⟩+≡ (8b) ◁8c 61e▷

```
#include <zlib.h>
#include <archive.h>
#include <archive_entry.h>
```

30b ⟨*CGI State Members* 9e⟩+≡ (8g) ◁28a 31b▷

```
z_stream *dec_zs;
struct archive *dec_ar;
char *dec_buf;
#define DEC_BUF_LEN 65536
```

30c ⟨*Initialize body decoder* 30c⟩≡ (29b)

```
if(!cgi_state->dec_buf) {
  /* decoding invalidates body length */
  if(cgi_state->body_enc && cgi_state->body_len >= 0)
    cgi_env_set_int("CGI.ContentLength", (cgi_state->body_len = -1));
  if(!strcasecmp(cgi_state->body_enc, "gzip") ||
     !strcasecmp(cgi_state->body_enc, "x-gzip") ||
     !strcasecmp(cgi_state->body_enc, "deflate")) {
    /* type is auto-detected by zlib, although we probably ought */
    /* to verify that advertised method matches detected method */
    ⟨Initialize decoding for zlib 30d⟩
#if ARCHIVE_VERSION_NUMBER >= 2007902
  } else if(!strcasecmp(cgi_state->body_enc, "compress") ||
            !strcasecmp(cgi_state->body_enc, "x-compress") ||
            !strcasecmp(cgi_state->body_enc, "bzip2") ||
            !strcasecmp(cgi_state->body_enc, "x-bzip2")) {
    ⟨Initialize decoding for libarchive 30e⟩
#endif
  } else {
    cgi_env_set("error_status", "415 Unsupported Media Type");
    return nerr_raise(GENERIC_ERR, "Encoding type %s not supported",
                      cgi_state->body_enc);
  }
}
```

30d ⟨*Initialize decoding for zlib* 30d⟩≡ (30c)

```
cgi_state->dec_buf = malloc(2 * DEC_BUF_LEN);
if(cgi_state->dec_buf)
  cgi_state->dec_zs = calloc(sizeof(*cgi_state->dec_zs), 1);
if(!cgi_state->dec_zs)
  return nerr_raise_msg_errno("Error decoding body");
```

30e ⟨*Initialize decoding for libarchive* 30e⟩≡ (30c)

```
struct archive_entry *ae; /* dummy */
cgi_state->dec_buf = malloc(DEC_BUF_LEN);
if(cgi_state->dec_buf)
  cgi_state->dec_ar = archive_read_new();
ae = archive_entry_new();
if(!cgi_state->dec_ar || !ae ||
   archive_read_support_compression_compress(cgi_state->dec_ar) ||
   archive_read_support_compression_bzip2(cgi_state->dec_ar) ||
```

30e ⟨*Initialize decoding for libarchive* 30e⟩≡ (30c)

```
    archive_read_support_format_raw(cgi_state->dec_ar) ||
    archive_read_open(cgi_state->dec_ar, cgi_state, NULL, read_cgi_for_ar, NULL) ||
    archive_read_next_header2(cgi_state->dec_ar, ae)) {
  if(ae)
    archive_entry_free(ae);
  return nerr_raise_msg_errno("Error decoding body");
}
```

31a ⟨*Free CGI state members* 18e⟩+≡ (11g) ◁21b 62b▷

```
if(cgi_state->dec_zs) {
  inflateEnd(cgi_state->dec_zs);
  free(cgi_state->dec_zs);
}
#if ARCHIVE_VERSION_NUMBER >= 2007902
if(cgi_state->dec_ar)
  archive_read_finish(cgi_state->dec_ar);
#endif
if(cgi_state->dec_buf)
  free(cgi_state->dec_buf);
```

The decoders should be run independently of the data requests, filling an output buffer as needed. This requires that an output buffer pointer and length be stored in the CGI state. Since the existence of the decoding buffer, rather than the existence of either decoder's state, is used to indicate decoding is needed, the decoder's state can be closed and freed on end of file, while still allowing the last output buffer to be used. This is really only possible with zlib, because libarchive allocates output buffers internally. Instead, libarchive decoding is cleared after the last read.

31b ⟨*CGI State Members* 9e⟩+≡ (8g) ◁30b 34a▷

```
const char *dec_out;
int dec_out_len;
```

31c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁29b 32a▷

```
NEOERR *fill_decode_buffer(cgi_state_t *cgi_state)
{
  if(cgi_state->dec_zs) {
    int ret = Z_OK;
    z_stream *zs = cgi_state->dec_zs;

    while(ret == Z_OK) {
      if(!zs->avail_in) {
        zs->next_in = (Bytef *)cgi_state->dec_buf;
        zs->avail_in = cgi_raw_read(cgi_state->dec_buf, DEC_BUF_LEN);
        if(zs->avail_in < 0)
          return nerr_raise_msg_errno("Error decoding body");
      }
      zs->next_out = (Bytef *)cgi_state->dec_buf + DEC_BUF_LEN;
      zs->avail_out = DEC_BUF_LEN;
      ret = inflate(zs, Z_SYNC_FLUSH);
      if(zs->avail_out < DEC_BUF_LEN || ret == Z_STREAM_END) {
        cgi_state->dec_out = cgi_state->dec_buf + DEC_BUF_LEN;
        cgi_state->dec_out_len = DEC_BUF_LEN - zs->avail_out;
        break;
      }
    }
    if(ret == Z_STREAM_END) {
      inflateEnd(zs);
      free(zs);
      cgi_state->dec_zs = NULL;
```

31c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁29b 32a▷

```
    } else if (ret != Z_OK)
      return nerr_raise_msg("Error decoding body");
#if ARCHIVE_VERSION_NUMBER >= 2007902
  } else if (cgi_state->dec_ar) {
    const void *outbuf;
    off_t outoff;
    size_t outlen;
    int ret = archive_read_data_block(cgi_state->dec_ar, &outbuf, &outlen,
                                      &outoff);
    if (ret != ARCHIVE_OK && ret != ARCHIVE_EOF)
      return nerr_raise_msg(archive_error_string(cgi_state->dec_ar));
    cgi_state->dec_out = outbuf;
    cgi_state->dec_out_len = outlen;
    /* raw format has no gaps, so outoff can be ignored */
    if (ret == ARCHIVE_EOF) {
      archive_read_finish(cgi_state->dec_ar);
      cgi_state->dec_ar = NULL;
    }
#endif
  }
  return STATUS_OK;
}
```

32a ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁31c 34b▷

```
#if ARCHIVE_VERSION_NUMBER >= 2007902
ssize_t read_cgi_for_ar(struct archive *ar, void *cbdata, const void **buf)
{
  cgi_state_t *cgi_state = cbdata;
  int ret;
  *buf = cgi_state->dec_buf;
  ret = cgi_raw_read(cgi_state->dec_buf, DEC_BUF_LEN);
  if (ret < 0)
    archive_set_error(ar, errno, "Error decoding body");
  return ret;
}
#endif
```

The specific action for the read function is to just return as much available data as possible, and to update the output buffer pointers.

32b ⟨*Get more decoded data* 32b⟩≡ (32c 33)

```
if (!cgi_state->dec_out_len) {
  nerr = fill_decode_buffer(cgi_state);
  if (nerr != STATUS_OK || !cgi_state->dec_out_len) {
    if (cgi_state->dec_buf) {
      free(cgi_state->dec_buf);
      cgi_state->dec_buf = NULL;
    }
    return nerr;
  }
}
```

32c ⟨*Decode data into* `buf`/`length` 32c⟩≡ (29b)

```
*retlength = 0;
if (length <= 0)
  return STATUS_OK;
⟨Get more decoded data 32b⟩
if (length > cgi_state->dec_out_len)
  length = cgi_state->dec_out_len;
```

32c ⟨*Decode data into* `buf/length` 32c⟩≡ (29b)

```
*retlength = length;
memcpy(buf, cgi_state->dec_out, length);
cgi_state->dec_out_len -= length;
cgi_state->dec_out += length;
return STATUS_OK;
```

For getting a line, the decode buffer needs to be scanned for newlines.

33 ⟨*Decode a string into* `buf/length` 33⟩≡ (29b)

```
if (length <= 0)
  return STATUS_OK;
*buf = 0;
if (!--length)
  return STATUS_OK;
while (1) {
  int cp_len = length;
  ⟨Get more decoded data 32b⟩
  if (cp_len > cgi_state->dec_out_len)
    cp_len = cgi_state->dec_out_len;
  length -= cp_len;
  cgi_state->dec_out_len -= cp_len;
  while (cp_len--)
    if ((*buf++ = *cgi_state->dec_out++) == '\n')
      break;
  *buf = 0;
  cgi_state->dec_out_len += cp_len + 1;
  if (cp_len >= 0 || buf[-1] == '\n')
    return STATUS_OK;
}
```

Chapter 6

POST and PUT Uploads

In order to handle POST parameters correctly, a parsing handler could be provided to the CGI kit. There are two problems with the CGI kit's handler mechanism, though. First, the handler can be selected by content type, but only by exact matches, so POST uploads which tack on an arbitrary boundary cannot be easily intercepted. Second, the handler is run in addition to the built-in handler, rather than instead of the built-in handler.

The first problem could be worked around by handling all content types and filtering in the function, or by only adding the handler if the content type is multi-part, using the exact content type string that was passed. The second problem could be worked around by returning a dummy error, which will abort the built-in parser.

Instead of these workarounds, the `cgi_parse` function is avoided entirely for multi-part POST parameters. Also, in order to be more forgiving to browsers, any unrecognized content type for POST parameters is converted to `application/x-www-form-urlencoded` to force reading the query string from standard input[1]. In fact, since the `cgi_parse` function doesn't do anything but upload and POST parameter parsing, it is bypassed for anything but simple POST parsing. Since the request method and content type are used to decide what to do, the content type is placed directly in the CGI state for quick access.

34a ⟨*CGI State Members* 9e⟩+≡ (8g) ◁31b 44c▷

```
const char *body_type;
```

34b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁32a 35b▷

```
static NEOERR *cgi_parse_multpart(cgi_state_t *cgi_state, const char *ctype)
{
  NEOERR *nerr = STATUS_OK;
  ⟨Parse multi-part CGI parameters 35a⟩
  return nerr;
}
```

34c ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁28b 47d▷

```
cgi_state->body_type = cgi_env_val("CGI.ContentType", "");
if(req_method == HTTP_REQ_POST) {
  if(!strncasecmp(cgi_state->body_type, "multipart/form-data", 19)) {
    if(!nerr_op_ok(cgi_parse_multpart(cgi_state, cgi_state->body_type)))
      return nerr;
  } else if(strcmp(cgi_state->body_type, "application/soap+xml")) {
    if(strcmp(cgi_state->body_type, "application/x-www-form-urlencoded"))
      hdf_set_value(cgi_state->cgi->hdf, "CGI.ContentType",
                    "application/x-www-form-urlencoded");
```

[1] This is probably unsafe; e.g., a previous version of this code did this to SOAP requests.

34c ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁28b 47d▷

```
    if(!nerr_op_ok(cgi_parse(cgi_state->cgi)))
      return nerr;
  }
}
cgi_state->hdf = cgi_state->cgi->hdf; /* in case it changed */
```

Parsing multi-part POST parameters per RFC 2388[2] is straightforward:

35a ⟨*Parse multi-part CGI parameters* 35a⟩≡ (34b)

```
⟨Multi-part Parse Variables 37a⟩

⟨Find multi-part boundary 35c⟩
⟨Create buffer at least large enough for multi-part boundary 36a⟩
while(1) {
  ⟨Find next multi-part boundary and break if last 36d⟩
  ⟨Parse multi-part headers until blank line 37b⟩
  if(!(⟨Found multi-part file 39⟩)) {
    ⟨Parse multi-part body as parameter 40c⟩
  }
  ⟨Set multi-part parameters 41b⟩
  if(⟨Found multi-part file 39⟩) {
    ⟨Parse multi-part body as file 44d⟩
  }
}
⟨Clean up multi-part parse 36b⟩
```

There are two ways to read lines: read everything in a line, like `g_string_fgets`, or read just enough to fill a buffer, like `fgets`. Both have disadvantages. Reading everything has the potential to use too much memory. Reading just the amount there is room for has problems with long lines. The HTTP and MIME specifications try to keep line lengths reasonable, but there is never any guarantee. For this reason, only file reads will use a limited-length buffer. The rest will use the CGI equivalent of `g_string_fgets`, but with ASCII NUL support:

35b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁34b 40d▷

```
static char *cgi_gets_cb(char *buf, int len, void *str)
{
  cgi_state_t *cgi_state = str;
  NEOERR *nerr = cgi_gets(buf, len);
  if(nerr == STATUS_OK)
    return buf;
  nerr_ignore(&nerr);
  return NULL;
}

char *g_string_cgi_gets(GString **_buf, guint offset, cgi_state_t *cgi_state)
{
  return g_string_gets_generic(_buf, offset, cgi_gets_cb, cgi_state, TRUE);
}
```

Most of the time, the boundary is passed in with the content type. However, the perl CGI module allows for the boundary to be missing, and instead uses the first line read from input as the boundary. This code will do the same. However, since the first line may be part of a file, the buffer will be limited to 128 characters. 78 is the most a boundary should contain, anyway.

[2] http://www.ietf.org/rfc/rfc2388.txt

35c ⟨*Find multi-part boundary* 35c⟩≡ (35a)

```
char boundary_buf[128];
const char *boundary = strstr(cgi_state->body_type + 19, "boundary=");
if(boundary)
  boundary += 9;
else {
  if(!cgi_gets(boundary_buf, sizeof(boundary_buf)))
    return STATUS_OK;
  /* in use, each boundary is preceeded by -- */
  if(!memcmp(boundary_buf, "--", 2))
    boundary = boundary_buf + 2;
  else
    return nerr_raise(NERR_PARSE, "Malformed input; no boundary");
}
int blen = strlen(boundary);
```

As is usual with this code, a `GString` is used to buffer the input. To make I/O more efficient, the buffer is at least 8192 bytes long.

36a ⟨*Create buffer at least large enough for multi-part boundary* 36a⟩≡ (35a) 36c▷

```
GString *buf = g_string_sized_new(blen > 8192 - 7 ? blen + 7 : 8192);
                                  /* 7 extra for --<b>--\r\n\0 */
```

36b ⟨*Clean up multi-part parse* 36b⟩≡ (35a) 37c▷

```
g_string_free(buf, TRUE);
```

The first thing to do in the loop is to read the next line. However, if the boundary line needed to be read, one line has already been read. Also, searching for the end of the body will usually result with the next boundary line having been read. For this reason, a line is always read before entring the loop.

36c ⟨*Create buffer at least large enough for multi-part boundary* 36a⟩+≡ (35a) ◁36a

```
if(boundary == boundary_buf + 2)
  g_string_assign(buf, boundary_buf);
else
  if(!cgi_gets(buf->str, buf->allocated_len)) {
    g_string_free(buf, TRUE);
    return STATUS_OK;
  }
```

The loop should be entered with the boundary, prefixed by a double dash, in the read buffer. However, there may be some sort of corruption in the headers. In that case, the next line is read using a limited buffer read and the loop is started over. If the boundary is valid, then the final boundary is indicated by an additional trailing double dash.

36d ⟨*Find next multi-part boundary and* `break` *if last* 36d⟩≡ (35a)

```
if(memcmp(buf->str, "--", 2) ||
   memcmp(buf->str + 2, boundary, blen)) {
  if(!cgi_gets(buf->str, buf->allocated_len))
    break; /* end of input */
  continue; /* invalid boundary; find next */
}
if(!strncmp(buf->str + 2 + blen, "--", 2) &&
   (buf->str[2 + blen + 2] == '\r' ||
    buf->str[2 + blen + 2] == '\n' ||
    !buf->str[2 + blen + 2]))
  break; /* valid end boundary */
if(buf->str[2 + blen] != '\r' &&
```

36d ⟨*Find next multi-part boundary and* `break` *if last* 36d⟩≡ (35a)

```
    buf->str[2 + blen] != '\n') {
  /* invalid boundary */
  if(!cgi_gets(buf->str, buf->allocated_len))
    break; /* end of input */
  continue; /* invalid boundary; find next */
}
```

The header consists of keywords, followed by a colon, followed by values. The values may be semicolon-separated keyword-value pairs. The only headers of interest are `content-dispostion`, which has keyword-value pairs as its value that include `name` and `filename`, `content-transfer-encoding`, and `content-length`. The `content-type` is returned by the CGI kit, so it needs to be saved as well. This will allow post-processors to deal with multipart files; otherwise such files would have to be rejected.

37a ⟨*Multi-part Parse Variables* 37a⟩≡ (35a) 37d▷

```
gint64 flen = -1;
char *content_type = NULL;
```

37b ⟨*Parse multi-part headers until blank line* 37b⟩≡ (35a) 38a▷

```
while(1) {
  g_string_cgi_gets(&buf, 0, cgi_state);
  if(!buf->str[0] || buf->str[0] == '\r' || buf->str[0] == '\n')
    break; /* blank line @ end of header */
  char *c = strchr(buf->str, ':');
  if(!c) /* invalid line; just ignore */
    continue;
  *c = 0;
  if(!strcasecmp(buf->str, "content-disposition")) {
    ⟨Parse content-disposition 38d⟩
  } else if(!strcasecmp(buf->str, "content-transfer-encoding")) {
    ⟨Parse content-transfer-encoding 37e⟩
  } else if(!strcasecmp(buf->str, "content-length")) {
    while(isspace(*++c));
    if(isdigit(*c))
      flen = strtoull(c, NULL, 10);
  } else if(!strcasecmp(buf->str, "content-type")) {
    while(isspace(*++c));
    if(!content_type)
      content_type = strdup(c);
  }
  /* ignore other header lines */
}
```

37c ⟨*Clean up multi-part parse* 36b⟩+≡ (35a) ◁36b 38c▷

```
if(content_type)
  free(content_type);
```

The content-transfer-encoding field indicates encodings other than “7bit”, which indicates plain text, sans the final newline. Any unknown encoding will be ignored. Note that this is another improvement over ClearSilver: ClearSilver only supports the plain encoding types.

37d ⟨*Multi-part Parse Variables* 37a⟩+≡ (35a) ◁37a 38b▷

```
enum { CT_ENC_PLAIN, CT_ENC_Q, CT_ENC_B, CT_ENC_INV } enct = CT_ENC_PLAIN;
```

37e ⟨*Parse content-transfer-encoding* 37e⟩≡ (37b)

```
while(isspace(*++c));
char *t = c;
while(*t && !isspace(*t))
  t++;
*t = 0;
if(!strcasecmp(c, "7bit") || !strcasecmp(c, "8bit") || !strcasecmp(c, "binary"))
  enct = CT_ENC_PLAIN;
else if(!strcasecmp(c, "quoted-printable"))
  enct = CT_ENC_Q;
else if(!strcasecmp(c, "base64"))
  enct = CT_ENC_B;
else if(*c)
  enct = CT_ENC_INV;
```

38a ⟨*Parse multi-part headers until blank line* 37b⟩+≡ (35a) ◁37b

```
if(enct == CT_ENC_INV)
  continue; /* scan for next header; enctype not supported */
```

Parsing the multi-valued content-disposition field[3] is the hardest. First it must locate semicolon-separated fields, ignoring whitespace. The first field is ignored entirely, because it is the content disposition itself. Spurious semicolons are just ignored. The field name is always literal text, so it is fairly easy to parse out. The specfic field name is checked after the full pair has been parsed. After the field name comes an equals sign, surrounded by optional whitespace. Then comes the value, which can be either a token or a quoted string. Quotes embedded in values are not always properly escaped, so an ending quote is only accepted if followed immediately by a semicolon or end of line. Since the values need to be saved, the entire header line is saved as well.

38b ⟨*Multi-part Parse Variables* 37a⟩+≡ (35a) ◁37d 40a▷

```
GString *content_disposition = NULL;
const char *pname = NULL, *fname = NULL;
const char *mdate = NULL, *cdate = NULL, *rdate = NULL;
```

38c ⟨*Clean up multi-part parse* 36b⟩+≡ (35a) ◁37c 40b▷

```
if(content_disposition)
  g_string_free(content_disposition, TRUE);
```

38d ⟨*Parse content-disposition* 38d⟩≡ (37b)

```
char *s, *n, *ne, *v;
gboolean restore_semi = FALSE;
if(!content_disposition)
  content_disposition = g_string_new(buf->str);
else
  continue; /* multiple disposition lines not supported */
for(s = strchr(content_disposition->str + 20, ';'); s && *s; ) {
  while(*s == ';' || isspace(*s))
    s++;
  if(!*s)
    break;
  n = s;
  /* RFC 2183 would have me filter more chars, but this is good enough */
```

[3] http://www.ietf.org/rfc/rfc2183.txt

38d ⟨*Parse content-disposition* 38d⟩≡ (37b)

```
  while (isgraph(*s) && *s != ';' && *s != '=')
    s++;
  ne = s;
  while (isspace(*s))
    s++;
  if (!*s)
    break;
  if (*s != '=')
    continue;
  *ne = 0;
  while (isspace(*++s));
  v = s;
  if (*v != '"') {
    while (isgraph(*++s));
    ne = s;
    if (*s && !isspace(*s) && *s != ';')
      /* invalid token; ignore */
      continue;
    while (isspace(*s))
      s++;
    if (*s && *s != ';')
      /* invalid token again; ignore */
      continue;
    restore_semi = *ne == ';';
    *ne = 0;
  } else {
    ne = ++v;
    for (s = v; *s; s++) {
      if (*s == '\\' && s[1] && s[1] != '\r' && s[1] != '\n')
        *ne = *++s;
      else if (*s == '"') {
        char *ss;
        for (ss = s + 1; isspace(*ss); ss++);
        if (!*ss || *ss == ';')
          break;
        *ne = *s;
      } else
        *ne = *s;
    }
    *ne = 0;
  }
  if (!strcasecmp(n, "name"))
    pname = v;
  else if (!strcasecmp(n, "filename"))
    fname = v;
  else if (!strcasecmp(n, "creation-date"))
    cdate = v;
  else if (!strcasecmp(n, "modification-date"))
    mdate = v;
  else if (!strcasecmp(n, "read-date"))
    rdate = v;
  else if (!strcasecmp(n, "size"))
    if (isdigit(*v))
      flen = strtoull(v, NULL, 10);
  /* else ignore */
  if (restore_semi) {
    *ne = ';';
    restore_semi = FALSE;
  }
}
```

39 ⟨*Found multi-part file* 39⟩≡ (35a)

```
fname
```

Finally, the body is parsed. This must take content encoding into consideration, but otherwise includes all text up to the last end-of-line followed by a boundary line. For CGI values, the value is assumed to be small enough to fit in memory, so it is sucked into a buffer and assigned to the parameter.

40a ⟨*Multi-part Parse Variables* 37a⟩+≡ (35a) ◁38b 41a▷

```
GString *val = NULL;
```

40b ⟨*Clean up multi-part parse* 36b⟩+≡ (35a) ◁38c

```
if(val)
  g_string_free(val, TRUE);
```

40c ⟨*Parse multi-part body as parameter* 40c⟩≡ (35a)

```
int off = 0;
while(g_string_cgi_gets(&val, off, cgi_state)) {
  if(val->str[off] == '-' && val->str[off + 1] == '-' &&
     !strncmp(val->str + off + 2, boundary, blen))
    break;
    off += val->len;
}
if(val->len > off)
  g_string_assign(buf, val->str + off);
else
  g_string_assign(buf, "");
if(off > 0 && val->str[off - 1] == '\n')
  off--;
if(off > 0 && val->str[off - 1] == '\r')
  off--;
val->len = off;
val->str[off] = 0;
if(enct == CT_ENC_Q) {
  val->len = decode_quoted_printable(val->str);
  val->str[val->len] = 0;
} else if(enct == CT_ENC_B) {
  gint b64_state = 0;
  guint b64_save = 0;

  val->len = g_base64_decode_step(val->str, val->len, (guchar *)val->str,
                                  &b64_state, &b64_save);
  val->str[val->len] = 0;
}
```

40d ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁35b 43▷

```
int decode_quoted_printable(char *buf)
{
  char *s, *d, *l;

  for(d = s = buf; *s; s++) {
    /* note: RFC says upper-case only, but allow lower-case since */
    /* otherwise lower-case just makes it invalid */
    if(*s == '=' && isxdigit(s[1]) && isxdigit(s[2])) {
      *d++ = *++s & 0xf;
      if(*s > '9')
        *d += 10 - ('a' & 0xf);
      *d <<= 4;
      *d += *++s & 0xf;
```

40d ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁35b 43▷

```
      if(*s > '9')
        *d += 10 - ('a' & 0xf);
      d++;
    } else if(*s == '=' && (!s[1] || isspace(s[1]))) {
      for(l = s + 1; isspace(*l) && *l != '\r' && *l != '\n'; l++);
      if(!*l || *l == '\r' || *l == '\n') {
        s = l;
        if(s[1] == '\r' || s[1] == '\n')
          s++;
      } else {
        /* otherwise invalid, but go ahead and retain = */
        *d++ = *s;
      }
    } else if(*s == '\r' || *s == '\n') {
      *d++ = *s;
      if(s[1] == '\n' || s[1] == '\r') {
        *d++ = *++s;
    } else if(!isspace(*s))
      /* invalid if =, but go ahead and retain */
      *d++ = *s;
    } else {
      /* for every space, see if it's at end of line, in which case remove */
      for(l = s + 1; isspace(*l) && *l != '\r' && *l != '\n'; l++);
      if(*l == '\r' || *l == '\n')
        s = l;
      else {
        memmove(d, s, (int)(l - s));
        d += (int)(l - s);
        s += (int)(l - s) - 1;
      }
    }
  }
  return (int)(d - buf);
}
```

The algorithm for setting the correct HDF values was lifted from the ClearSilver source with minor modification. The file-related attributes are added as a hierarchy under the parameter. The value of the file parameter itself is also the file name. Since the primary value is a string, NUL characters in the value and all subsequent characters are lost. If lost NUL characters are detected, the raw buffer is duplicated and assigned to the value, and a "Length" subparameter is set to the parameter length. It is up to the user to check for the presence of this subparameter if binary values are expected. In most cases, though, the user will likely silently ignore that as an error.

41a ⟨*Multi-part Parse Variables* 37a⟩+≡ (35a) ◁40a

```
int unnamed_parms = 0;
char nbuf[20];
```

41b ⟨*Set multi-part parameters* 41b⟩≡ (35a)

```
if(!pname) {
  snprintf(nbuf, sizeof(nbuf), "_%d", unnamed_parms++);
  pname = nbuf;
}
HDF *obj = cgi_parm_obj(pname);
if(obj) {
  /* multiple values get assigned to .0, .1, ... */
  /* but first one remains on named node, and is copied to .0 */
  int i = 0;
  HDF *child;
  /* don't count FileName & other non-numeric parms */
```

41b ⟨*Set multi-part parameters* 41b⟩≡ (35a)

```
  for(child = hdf_obj_child(obj); child; child = hdf_obj_next(child))
    if(isdigit(hdf_obj_name(child)[0]))
      i++;
  if(!i) {
    nerr_op(hdf_set_value(obj, "0", hdf_obj_value(obj)));
    if(nerr != STATUS_OK)
      return nerr;
    /* note that doing an hdf_copy() would cause an infinite recursion */
    /* same for soft link */
    HDF *zero = hdf_get_obj(obj, "0");
    for(child = hdf_obj_child(obj); child; child = hdf_obj_next(child))
      if(!isdigit(hdf_obj_name(child)[0]))
        if(!nerr_op_ok(hdf_set_value(zero, hdf_obj_name(child),
                                     hdf_obj_value(child))))
          return nerr;
    i = 1;
  }
  sprintf(nbuf, "%d", i);
  pname = nbuf;
} else
  /* single value gets assigned to named node */
  obj = cgi_state->cgi_parms;
/* pname is now parm to set, and obj is parent HDF */
if(fname)
  nerr_op(hdf_set_value(obj, pname, fname));
else if(strlen(val->str) == val->len)
  nerr_op(hdf_set_value(obj, pname, val->str));
else {
  char *vbuf = malloc(val->len + 1);
  if(!vbuf)
    nerr = nerr_raise_errno(NERR_NOMEM, "parameter too long");
  else {
    memcpy(vbuf, val->str, val->len);
    if(nerr_op_ok(hdf_set_buf(obj, pname, vbuf))) {
      obj = hdf_get_obj(obj, pname);
      /* obj is ready for setting children of newly added parm */
      nerr_op(hdf_set_int_value(obj, "Length", val->len));
    }
  }
}
if(nerr == STATUS_OK && fname) {
  obj = hdf_get_obj(obj, pname);
  /* obj is ready for setting children of newly added parm */
  if(mdate)
    nerr_op((hdf_set_int64_value(obj, "ModTime", parse_http_date(mdate,
                                                                 FALSE))));
  if(nerr == STATUS_OK && cdate)
    nerr_op(hdf_set_int64_value(obj, "CreationTime", parse_http_date(cdate,
                                                                     FALSE)));
  if(nerr == STATUS_OK && rdate)
    nerr_op(hdf_set_int64_value(obj, "AccessTime", parse_http_date(rdate,
                                                                   FALSE)));
  if(nerr == STATUS_OK && content_type)
    nerr_op(hdf_set_value(obj, "Type", content_type));
  if(nerr == STATUS_OK && flen >= 0)
    hdf_set_int64_value(obj, "Length", flen);
}
if(nerr != STATUS_OK)
  return nerr;
```

The date fields are converted to a standard UNIX GMT timestamp as per RFC 822 or RFC 2616, which references RFC 1123/822 and RFC 850. HTTP dates are a bit stricter than the MIME dates, so a flag chooses which. The timezone format requires glibc or similar extensions. This function proceeds similarly to the

more generic `parse_date` function, but limits the acceptable formats more.

43 ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁40d 49b▷

```
static const char const * http_date_fmts[] = {
  /* RFC 1123, RFC 2616 rfc1123-date */
  "%a, %d %b %Y %T %z",
  /* RFC 1123 */
  "%d %b %Y %T %z",
  "%a, %d %b %Y %R %z",
  "%d %b %Y %R %z",
  /* RFC 822 */
  "%a, %d %b %y %T %z",
  "%d %b %y %T %z",
  "%a, %d %b %y %R %z",
  "%d %b %y %R %z",
  /* RFC 2616 rfc850-date */
  "%a, %d-%b-%y %T %z",
  /* RFC 2616 asctime-date */
  "TZ=GMT",
  "%a %b %d %T",
  NULL
};

time_t parse_http_date(const char *date, gboolean http_only)
{
  const char *e, *fmt, **fmtp;
  struct tm tm_parsed;
  time_t t;
  const char *old_tz = NULL; /* init to shut gcc up */
  gboolean set_tz = FALSE;

  if(!date || !*date)
    return -1;
  for(fmtp = http_date_fmts, e = NULL; (!e || *e) && (fmt = *fmtp); fmtp++) {
    if(!strncmp(fmt, "TZ=", 3)) {
      if(!set_tz) {
        pthread_mutex_lock(&tz_lock);
        old_tz = getenv("TZ");
        set_tz = TRUE;
      }
      setenv("TZ", fmt + 3, 1);
      tzset();
    } else {
      memset(&tm_parsed, 0, sizeof(tm_parsed));
      e = strptime(date, fmt, &tm_parsed);
      /* HTTP requires GMT time zone */
      /* guaranteed more than 3 chars; time alone is 5 chars */
      if(http_only && e && !*e && strstr(fmt, "%z") &&
         strcmp(e - 4, " GMT"))
        e = NULL;
    }
  }
  if(e && !*e)
    t = mktime(&tm_parsed);
  else
    t = -1;
  if(set_tz) {
    if(old_tz)
      setenv("TZ", old_tz, 1);
    else
      unsetenv("TZ");
    tzset();
    pthread_mutex_unlock(&tz_lock);
  }
```

43 ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁40d 49b▷

```
  return t;
}
```

As a last step for files, the contents are read into either a temporary file in the same way that the CGI kit does it, or, if provided, callbacks are called. The callbacks are defined in the CGI state structure rather than passed as arguments. They are placed in a substructure, though, to make storage of existing callbacks easier when chaining multiple callbacks together.

44a ⟨*CGI State Dependencies* 9c⟩+≡ (8g) ◁24e 98a▷

```
typedef NEOERR *(*cgi_open_cb)(HDF *obj, cgi_state_t *cgi_state);
typedef NEOERR *(*cgi_write_cb)(HDF *obj, cgi_state_t *cgi_state,
                                guint64 off, const void *buf, int len);
typedef NEOERR *(*cgi_progress_cb)(HDF *obj, cgi_state_t *cgi_state,
                                   guint64 off, int len);
typedef NEOERR *(*cgi_close_cb)(HDF *obj, cgi_state_t *cgi_state,
                                NEOERR *write_errors);
typedef struct {
  cgi_open_cb open;
  cgi_write_cb write;
  cgi_progress_cb progress;
  cgi_close_cb close;
  void *data;
} cgi_upload_cb;
```

44b ⟨*(build.nw) Known Data Types* 9a⟩+≡ ◁24h 63c▷

```
cgi_open_cb,cgi_write_cb,cgi_progress_cb,cgi_close_cb,cgi_upload_cb,%
```

44c ⟨*CGI State Members* 9e⟩+≡ (8g) ◁34a 61f▷

```
cgi_upload_cb upload_cb;
```

44d ⟨*Parse multi-part body as file* 44d⟩≡ (35a)

```
⟨Open multi-part body file 45a⟩
if(nerr == STATUS_OK) {
  guint64 off = 0;
  gboolean eof = FALSE;
  ⟨Prepare to read multi-part body file lines 45b⟩
  while(nerr == STATUS_OK && ⟨Read a multi-part body line 45c⟩) {
    ⟨Break multi-part file read and set eof if boundary found 45d⟩
    ⟨Decode multi-part file line 46b⟩
    ⟨Write multi-part body line to file 46d⟩
    if(nerr != STATUS_OK)
      break;
    ⟨Prepare for next multi-part body file line 46a⟩
  }
  if(!eof)
    while(⟨Read a multi-part body line 45c⟩) {
      ⟨Break multi-part file read and set eof if boundary found 45d⟩
      ⟨Prepare for next multi-part body file line 46a⟩
    }
  ⟨Close multi-part body file 47a⟩
}
```

Opening the file the CGI kit way is made a little easier via the "internal use only" `open_upload` function. This will likely disappear in a future revision of ClearSilver, but it works for now. Just as the original code which calls this function does, the unlinking parameter (which should probably always be on anyway) must be read manually and passed into the function, and the "file descriptor" must be read as the

number of files opened so far. Errors writing the file should result in an unlink anyway, so the path name is saved for later if unlinking is disabled.

45a ⟨*Open multi-part body file* 45a⟩≡ (44d 47d) 47b▷

```
FILE *f = NULL;
char *path = NULL;

if(cgi_state->upload_cb.open)
  nerr_op((*cgi_state->upload_cb.open)(obj, cgi_state));
else {
  int unlink_it = hdf_get_int_value(cgi_state->cgi->hdf,
                                    "Config.Upload.Unlink", 1);
  nerr = nerr_pass_ctx(open_upload(cgi_state->cgi, unlink_it, &f),
                       "Can't open file for %s upload",
                        hdf_obj_name(obj));
  int entryno = uListLength(cgi_state->cgi->files);
  hdf_set_int_value(obj, "FileHandle", entryno);
  /* rfc2388.c checks errors on uListGet, but there should never be any */
  uListGet(cgi_state->cgi->files, entryno - 1, (void *)&f);
  if(!unlink_it) {
    uListGet(cgi_state->cgi->filenames, entryno - 1, (void *)&path);
    nerr_op(hdf_set_value(obj, "FileName", path));
  }
}
```

Reading lines requires stripping off the trailing carriage return and line feed for the last line. Since the last line is only known to be the last line after the boundary is read (i.e., this code does not use content-length), the trailing characters (either a newline alone or a carriage returned followed by a newline, even though the RFC only supports the latter) are always stripped off, and then re-added to the beginning of the next line's buffer. Since the buffer length is limited, lines may span multiple buffers. The boundary line is only sensed at the start of a new line, so a flag is used to indicate that the last read was a full line. It would be nice to use the indication of chracters to prepend as a flag that the buffer starts a new line (and may therefore be a boundary), but that would skip the first line.

Just like the infinite line reader, this must find the end of the read by means other than the return value. For this reason, the buffer is always pre-filled, and the last terminator is found to determine length.

45b ⟨*Prepare to read multi-part body file lines* 45b⟩≡ (44d) 46c▷

```
gboolean was_line = TRUE;
int eol_len = 0;
memset(buf->str, 255, buf->allocated_len);
```

45c ⟨*Read a multi-part body line* 45c⟩≡ (44d)

```
cgi_gets(buf->str + eol_len, buf->allocated_len - eol_len)
```

45d ⟨*Break multi-part file read and set* `eof` *if boundary found* 45d⟩≡ (44d)

```
/* no need to detect binary data in boundary */
if(was_line && buf->str[eol_len] == '-' && buf->str[eol_len + 1] == '-' &&
  !strncmp(buf->str + 2, boundary, blen))
  break;
char *ep;
for(ep = buf->str + buf->allocated_len - 1; *ep; --ep); /* memrchr is GNU */
buf->len = (int)(ep - buf->str);
if(buf->len == eol_len)
  break; /* ignore EOL chars at EOF as well */
was_line = buf->str[buf->len - 1] == '\n';
if(was_line) {
  if(buf->len > 1 && buf->str[buf->len - 1] == '\r')
```

45d ⟨*Break multi-part file read and set* `eof` *if boundary found* 45d⟩≡ (44d)

```
    eol_len = 2;
  else
    eol_len = 1;
  buf->len -= eol_len;
  if(!buf->len)
    continue; /* 1st 1-2 chars are already [\r]\n */
}
```

46a ⟨*Prepare for next multi-part body file line* 46a⟩≡ (44d)

```
if(eol_len) {
  buf->str[0] = '\r';
  buf->str[eol_len - 1] = '\n';
  buf->str[eol_len] = 0;
  buf->len = eol_len;
}
```

The CGI kit only supports plain data; there is no decoding required for that. This library also supports the standard text encoding methods. These are slightly more complicated than the decoding done for parameter values, because only one line at a time is available for decoding.

46b ⟨*Decode multi-part file line* 46b⟩≡ (44d)

```
if(enct == CT_ENC_Q) {
  /* RFC states that no line shall exceed 79 chars, so assume EOL present */
  /* handle line continuation here */
  while(buf->len > 0 && isspace(buf->str[buf->len - 1]))
    buf->len--;
  if(buf->len > 0 && buf->str[buf->len - 1] == '=') {
    eol_len = 0;
    buf->len--;
  }
  buf->str[buf->len] = 0;
  int init_eol = 0;
  if(buf->str[0] == '\r')
    init_eol = 1;
  if(buf->str[init_eol] == '\n')
    init_eol++;
  buf->len = decode_quoted_printable(buf->str + init_eol) + init_eol;
  buf->str[buf->len] = 0;
} else if(enct == CT_ENC_B) {
  buf->len = g_base64_decode_step(buf->str, buf->len, (guchar *)buf->str,
                                  &b64_state, &b64_save);
  buf->str[buf->len] = 0;
}
```

46c ⟨*Prepare to read multi-part body file lines* 45b⟩+≡ (44d) ◁45b

```
gint b64_state = 0;
guint b64_save = 0;
```

Writing the result is straightforward. Unfortunately, there is no way to control how much data has been read; accumulating a large amount into a buffer would make the writes more efficient. A progress callback is also called to replace the standard kit's upload callback; like that callback, it is called before the associated write. If a user callback was provided for opening the file, a user call back must be provided for the write, as well, since no assumption can be made that the callback data is a file descriptor.

46d ⟨*Write multi-part body line to file* 46d⟩≡ (44d 47d)

```
if(cgi_state->upload_cb.progress) {
  if(!nerr_op_ok((*cgi_state->upload_cb.progress)(obj, cgi_state, off,
                                                  buf->len)))
    break;
}
if(cgi_state->upload_cb.write)
  nerr_op((*cgi_state->upload_cb.write)(obj, cgi_state, off, buf->str,
                                        buf->len));
else if(f)
  if(fwrite(buf->str, buf->len, 1, f) != 1)
    nerr = nerr_raise_errno(NERR_IO, "Failed write for %s upload",
                                     hdf_obj_name(obj));
```

Finally, the standard kit does not actually close the file. Instead, it seeks to the beginning and saves the pseudo file descriptor. This code also flushes the data to ensure it has been written. Any write errors cause the resulting file to be removed. This all is done in addition to the close callback, since it is not likely that the close callback will not also override the open callback.

47a ⟨*Close multi-part body file* 47a⟩≡ (44d 47d) 47c▷

```
if(f && nerr == STATUS_OK) {
  if(fflush(f))
    nerr = nerr_raise_errno(NERR_IO, "Failed write for %s upload",
                                     hdf_obj_name(obj));
  else
    fseek(f, 0, SEEK_SET);
}
if(f && nerr != STATUS_OK) {
  fclose(f);
  f = NULL;
  if(path)
    unlink(path);
}
if(cgi_state->upload_cb.close)
  nerr = (*cgi_state->upload_cb.close)(obj, cgi_state, nerr);
```

In case the standard open routine is used, but callbacks want to access the file, the file descriptor is assigned to the callback data if it was `NULL`. On close, the callback data is set back to `NULL` if it matches the file descriptor (very unlikely if it is not actually the file descriptor).

47b ⟨*Open multi-part body file* 45a⟩+≡ (44d 47d) ◁45a

```
if(!cgi_state->upload_cb.data)
    cgi_state->upload_cb.data = f;
```

47c ⟨*Close multi-part body file* 47a⟩+≡ (44d 47d) ◁47a

```
if((void *)f == cgi_state->upload_cb.data)
  cgi_state->upload_cb.data = NULL;
```

The POST method is not the only method with a body. The PUT method is normally handled by `cgi_parse`, but the above code disables that processing. The built-in PUT processing, like the POST processing, has a few deficiencies. There is no callback at all, and the uploaded file is placed in a common temporary directory. It also does not support any content encoding. For this implementation, the upload parameters are saved to the `PUT` variable as with the standard library, and the upload callbacks are used with this parameter. Since the callback has access to the method, that can be used to distinguish the cases.

47d ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁34c 49a▷

```
if(req_method == HTTP_REQ_PUT) {
  guint64 off = 0;
  HDF *obj;
  struct {
    char *str;
    int len;
  } _buf, *buf = &_buf;
  const char *pi = cgi_env_val("CGI.PathInfo", "");

  nerr_op(cgi_env_set("PUT", pi));
  if(*cgi_state->body_type)
    nerr_op(cgi_env_set("PUT.Type", cgi_state->body_type));
  /* this is done by cgi_decode_read as well, but that's too late for */
  /* open callback */
  if(cgi_state->body_enc && cgi_state->body_len >= 0)
    nerr_op(cgi_env_set_int("CGI.ContentLength", (cgi_state->body_len = -1)));
  nerr_op(cgi_env_set_int("PUT.Length", cgi_state->body_len));
  if(nerr != STATUS_OK)
    return nerr;
  obj = cgi_env_obj("PUT");
  if(!(_buf.str = malloc(DEC_BUF_LEN)))
    return nerr_raise_msg_errno("Error reading upload stream");
  ⟨Open multi-part body file 45a⟩
  if(nerr != STATUS_OK) {
    free(_buf.str);
    return nerr;
  }
  while(nerr_op_ok(cgi_read(_buf.str, DEC_BUF_LEN, &_buf.len)) && _buf.len > 0) {
    ⟨Write multi-part body line to file 46d⟩
  }
  ⟨Close multi-part body file 47a⟩
}
```

Chapter 7

Common Headers

Some headers are handled differently depening on the request method. However, for headers that are sufficiently well defined by RFCs, some syntax checking and parsing can be done for any CGI program that needs it.

49a ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁47d 62a▷

```
HDF *hobj; /* the header value under consideration */
⟨Process common HTTP headers 50⟩
if(nerr != STATUS_OK) {
  cgi_env_set("error_status", "401 Bad Request");
  return nerr;
}
```

First are the ones defined by RFC 2616. Many of these require processing tokens and quoted strings.

49b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁43 49c▷

```
char *skip_http_token(const char *s)
{
  while(*s > 32 && !strchr("()<>@,;:\\\"/[]?={}\177", *s))
    s++;
  return (char *)s;
}
```

49c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁49b 49d▷

```
char *skip_http_quoted_string(const char *s)
{
  if(*s != '"')
    return NULL;
  while(*++s != '"' && *s)
    if(*s == '\\' && s[1])
      s++;
  if(*s != '"')
    return NULL;
  return (char *)s + 1;
}
```

49d ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁49c 62c▷

```
char *mangle_and_skip_http_quoted_string(char *s, char **ms)
{
  char *d;
  if(*s != '"')
    return NULL;
  d = s + 1;
  if(ms)
    *ms = d;
  while(*++s != '"' && *s) {
    if(*s == '\\' && s[1])
      s++;
    *d++ = *s;
  }
  if(*s != '"')
    return NULL;
  *d = 0;
  return s + 1;
}
```

The Accept header takes a list of media types, with optional keyword-value pairs describing each type. The `q` keyword has a particular format and interpretation. Errors cause the header to be ignored rather than giving a 401 or 406 error. Each supplied content-type is added as a child to the header, and the keyword-value pairs are stored as attributes for those children. No attempt is made to sort by precedence.

50 ⟨*Process common HTTP headers* 50⟩≡ (49a) 52a▷

```
if((hobj = cgi_header_obj("Accept"))) {
  char nbuf[22];
  int cno = 0;
  char *hv = hdf_obj_value(hobj), *ev, *vp, *vs, ec;
  HDF *cobj;
  while(hv) {
    vs = hv;
    if(*hv == '*')
      ev = hv + 1;
    else
      ev = skip_http_token(hv);
    if(ev != hv) {
      vp = ev;
      while(isspace(*ev))
        ev++;
      if(*ev == '/') {
        *vp++ = '/';
        while(isspace(*++ev));
        if(*ev == '*')
          hv = ev + 1;
        else if(*vs != '*')
          hv = skip_http_token(ev);
        else
          hv = ev;
        if(hv != ev) {
          if(ev != vp)
            memmove(ev, vp, (int)(hv - ev));
          vp += (int)(hv - ev);
          ec = *vp;
          sprintf(nbuf, "%d", cno++);
          die_if_err(hdf_get_node(hobj, nbuf, &cobj));
          *vp = 0;
          hdf_set_value(cobj, NULL, vs);
          *vp = ec;
          while(hv) {
            char *as, *ae, *avs, *ave;
            while(isspace(*hv))
```

50 ⟨*Process common HTTP headers* 50⟩≡ (49a) 52a▷

```
            hv++;
          if(*hv != ';')
            break;
          as = hv;
          ae = hv = skip_http_token(hv);
          if(ae != as) {
            while(isspace(*hv))
              hv++;
            if(*hv == '=') {
              while(isspace(*++hv));
              avs = hv;
              if(*hv == '"')
                ave = mangle_and_skip_http_quoted_string(hv, &avs);
              else {
                ave = skip_http_token(avs);
                if(ave == avs)
                  ave = NULL;
              }
            } else
              avs = ave = ae;
            if(ave) {
              ec = *ave;
              *ave = *ae = 0;
              die_if_err(hdf_set_attr(cobj, NULL, as, avs));
              if(!strcmp(as, "q")) {
                if(*avs != '0' || *avs != '1' ||
                    (avs[1] && (avs[1] != '.' ||
                                (avs[2] && !isdigit(avs[2])) ||
                                (avs[3] && !isdigit(avs[3])) ||
                                (avs[4] && !isdigit(avs[4])) ||
                                avs[5])))
                  hv = NULL;
              }
            } else
              hv = NULL;
          } else
            hv = NULL;
        }
      } else
        hv = NULL;
    } else
      hv = NULL;
    if(hv) {
      while(isspace(*hv))
        hv++;
      if(!*hv)
        break;
      if(*hv != ',') {
        hv = NULL;
        break;
      }
      while(isspace(*++hv));
    }
  }
  if(!hv)
    hdf_remove_tree(cgi_state->headers, "Accept");
}
```

The Accept-Charset header takes a list of character set identifiers, with optional `q` keywords to describe preference order. It is processed the same way as the Accept header, except that the character set format is different from a content type, and only the `q` keyword is allowed. The format for the Accept-Encoding header

is the same. The format for Accept-Language is stricter, but a non-matching language is a non-matching language regardless of the reason, so they are parsed the same way as well.

52a ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁50 53a▷

```
if((hobj = cgi_header_obj("AcceptCharset"))) {
  ⟨Process priority http token list 52b⟩
  if(!hv)
    hdf_remove_tree(cgi_state->headers, "AcceptCharset");
}
if((hobj = cgi_header_obj("AcceptEncoding"))) {
  ⟨Process priority http token list 52b⟩
  if(!hv)
    hdf_remove_tree(cgi_state->headers, "AcceptEncoding");
}
if((hobj = cgi_header_obj("AcceptLanguage"))) {
  ⟨Process priority http token list 52b⟩
  if(!hv)
    hdf_remove_tree(cgi_state->headers, "AcceptLanguage");
}
```

52b ⟨*Process priority http token list* 52b⟩≡ (52a)

```
char nbuf[22];
int cno = 0;
char *hv = hdf_obj_value(hobj), *ev, ec;
HDF *cobj;
while(hv) {
  if(*hv == '*')
    ev = hv + 1;
  else
    ev = skip_http_token(hv);
  if(ev != hv) {
    ec = *ev;
    sprintf(nbuf, "%d", cno++);
    die_if_err(hdf_get_node(hobj, nbuf, &cobj));
    *ev = 0;
    hdf_set_value(cobj, NULL, hv);
    *ev = ec;
    while(isspace(*hv))
      hv++;
    if(*hv == ';') {
      while(isspace(*++hv));
      if(*hv == 'q') {
        while(isspace(*++hv));
        if(*hv == '=') {
          while(isspace(*++hv));
          ev = skip_http_token(hv);
          ec = *ev;
          *ev = 0;
          if(*hv != '0' || *hv != '1' ||
              (hv[1] && (hv[1] != '.' ||
                         (hv[2] && !isdigit(hv[2])) ||
                         (hv[3] && !isdigit(hv[3])) ||
                         (hv[4] && !isdigit(hv[4])) ||
                         hv[5])))
            hv = NULL;
          else
            die_if_err(hdf_set_attr(cobj, NULL, "q", hv));
          *ev = ec;
        } else
          hv = NULL;
      } else
        hv = NULL;
    }
```

52b ⟨*Process priority http token list* 52b⟩≡ (52a)

```
    if (hv) {
      while (isspace(*hv))
        hv++;
      if (!*hv)
        break;
      if (*hv != ',') {
        hv = NULL;
        break;
      }
      while (isspace(*++hv));
    }
  }
}
```

The Allow header's utility is limited, so it is left alone. The Authorization header and proxy, cache, and protocol control headers should be taken care of by the server, so they are also left alone. Entity headers may be useful for PUT requests, but they are ignored in the general case. The From, Host, and Referer fields are free-format. The application must process these headers if required.

The conditional headers are checked and adjusted. The If-Match and If-None-Match take a list of quoted strings, which are stored as children of the header parameter. The list of entity tags is split out into children of the HDF parameter after having the string quoting removed. The existence tag (*) is encoded as no children of the HDF parameter; if there are really no tags, an error 400 is returned. Although the string quoting is removed, the first character is retained as either a quote for strong tags, or a capital W for weak tags.

53a ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁52a 54a▷

```
if ((hobj = cgi_header_obj("IfMatch"))) {
  ⟨Set match children of hobj 53b⟩
}
if ((hobj = cgi_header_obj("IfNoneMatch"))) {
  ⟨Set match children of hobj 53b⟩
}
```

53b ⟨*Set match children of* hobj 53b⟩≡ (53a 54c)

```
char *v = hdf_obj_value(hobj);
if (!*v) {
  cgi_env_set("error_status", "400 Bad Request");
  die_msg("Invalid match format");
}
if (strcmp(v, "*")) {
  char *s, *e;
  int cno = 0;
  char nbuf[22];

  while (1) {
    s = v;
    if (toupper(*v) == 'W' && v[1] == '/')
      v += 2;
    if (*v != '"')
      break;
    if (s != v)
      *v = 'W';
    e = ++v;
    while (*v && *v != '"') {
      if (*v == '\\') {
        if (!v[1])
          break;
        ++v;
      }
```

53b ⟨*Set match children of* `hobj` 53b⟩≡ (53a 54c)

```
      *e++ = *v++;
    }
    if(!*v) {
      --v;
      break;
    }
    sprintf(nbuf, "%d", cno++);
    *e = 0;
    die_if_err(hdf_set_value(hobj, nbuf, s));
    while(isspace(*++v));
    if(!*v || *v != ',')
      break;
    while(isspace(*++v));
    if(!*v) {
      --v;
      break;
    }
  }
  if(*v) {
    cgi_env_set("error_status", "400 Bad Request");
    die_msg("Invalid match format");
  }
}
```

The If-Modified-Since and If-Unmodified-Since take a date parameter; this is converted to its numeric equivalent. The standard says that headers with invalid dates are to be ignored, rather than rejected with a request error.

54a ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁53a 54c▷

```
if((hobj = cgi_header_obj("IfModifiedSince"))) {
  ⟨Parse hobj as HTTP date 54b⟩
}
if((hobj = cgi_header_obj("IfUnmodifiedSince"))) {
  ⟨Parse hobj as HTTP date 54b⟩
}
```

54b ⟨*Parse* `hobj` *as HTTP date* 54b⟩≡ (54)

```
time_t t = parse_http_date(hdf_obj_value(hobj), TRUE);
if(t < 0) {
#if 0
  cgi_env_set("error_status", "400 Bad Request");
  die_msg("Invalid header date format");
#else
  cgi_header_set(hdf_obj_name(hobj), NULL);
#endif
} else
  hdf_set_int64_value(hobj, NULL, t);
```

The If-Range header gets either an entity tag, which is stored as a single child for consistency with the IfMatch header, or a date, which is converted to a numeric date for consistency with the If-Modified-Since header. It is to be ignored if no Range header is present.

54c ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁54a 55▷

```
if((hobj = cgi_header_obj("IfRange"))) {
  const char *cv = hdf_obj_value(hobj);
  if(!cgi_header_obj("Range"))
    cgi_header_set("IfRange", NULL);
  else {
```

54c ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁54a 55▷

```
    if(*cv == '"' || (toupper(*cv) == 'W' && cv[1] == '/')) {
      ⟨Set match children of hobj 53b⟩
      if(hdf_obj_next(hdf_obj_child(hobj))) {
        cgi_env_set("error_status", "400 Bad Request");
        die_msg("Only specify one entity tag for If-Range");
      }
    } else {
      ⟨Parse hobj as HTTP date 54b⟩
    }
  }
}
```

The next supported header is the partial download specifier, Range. This takes an arbitrary named range specifier, but the RFC only defines one, so that is what is supported: the byte range. Each range is converted into a child whose name is the start of the range and whose value is the end of the range. This makes it easier to sort and merge the ranges. If multiple ranges have the same start, the highest end value is used. Note that unknown range types are ignored by not having any children, and that the RFC recommendation of returning ranges in order is violated by merging overlapping ranges. Once the children are sorted, it will become impossible to match the requested ordering.

55 ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁54c 56a▷

```
if((hobj = cgi_header_obj("Range"))) {
  const char *cv = hdf_obj_value(hobj);
  if(!strncasecmp(cv, "bytes", 5)) {
    cv += 4;
    while(isspace(*++cv));
    while(1) {
      guint64 start = 0, end = ~0; /* open range */
      char nbuf[22];
      HDF *ov;
      gboolean got_start = FALSE, got_end = FALSE;
      if(isdigit(*cv)) {
        got_start = TRUE;
        start = strtoull(cv, (char **)&cv, 10);
        while(isspace(*cv))
          cv++;
      }
      if(*cv == '-') {
        while(isspace(*++cv));
        if(isdigit(*cv)) {
          got_end = TRUE;
          end = strtoull(cv, (char **)&cv, 10);
          while(isspace(*cv))
            cv++;
        }
      }
      if((*cv && *cv != ',') || (!got_start && !got_end) || end < start) {
        cgi_env_set("error_status", "400 Bad Request");
        die_msg("Invalid range specification");
      }
      sprintf(nbuf, "%llu", (ullong)start);
      if((ov = hdf_get_obj(hobj, nbuf))) {
        start = hdf_get_int64_value(hobj, nbuf, 0);
        if(start < end)
          die_if_err(hdf_set_int64_value(hobj, nbuf, end));
      } else
        die_if_err(hdf_set_int64_value(hobj, nbuf, end));
    }
  }
}
```

Next are the headers described by the WebDAV RFCs. RFC 4918 describes the Depth header, which should be ignored if not needed, but otherwise must consist of one of three values. Rather than ignore it as required, the header causes an error 400 if it is not one of the three known values.

56a ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁55 56b▷

```
if((hobj = cgi_header_obj("Depth"))) {
  const char *v = hdf_obj_value(hobj);
  if(v && strcmp(v, "0") && strcmp(v, "1") && strcasecmp(v, "infinity")) {
    cgi_env_set("error_status", "400 Bad Request"); /* 422? */
    die_msg("Depth is invalid");
  }
}
```

The next header is the Destination header, which must be a so-called *simple reference*. This is either an absolute URI or an absolute path with no relative references (`.` or `..`) which also normally matches the request URI as much as possible. For the Destination header, the last requirement is relaxed. For this library, URIs are expected to be relative to the same CGI program. To use this with a program that supports alternate targets, such as for thrid party copies, simply use the environment variable directly rather than the stored version, and delete it from the environment to avoid errors generated here.

56b ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁56a 56c▷

```
if((hobj = cgi_header_obj("Destination"))) {
  nerr = normalize_path_obj(hobj, cgi_state, NULL, TRUE, TRUE);
  if(nerr != STATUS_OK) {
    cgi_env_set("error_status", "400 Bad Request");
    return nerr_pass_ctx(nerr, "Destination");
  }
}
```

The RFC 4918 conditional header, If, is a bit more complex than the plain HTTP conditionals. It can specify conditions on several resources, via resource tags (the request URI by default). Multiple lists are then specified which apply to that resource. For each such list, a child is created whose value is the resource to which it applies, and whose children are the conditions which must all apply for the child to evaluate to true. The overall header evaluates to true if any of its immediate children evaluate to true, which implies that all of that child's children evaluate to true. The conditions themselves are an entity tag or a state token. The entity tags are encoded as above, except that inversion is specified by using a lower-case w for weak tags and a single quote for strong tags. State tokens are enclosed in angle brackets, so the less-than sign is retained as a prefix to distinguish them, or a greater-than sign for inverted state tokens. The only validation done on state tokens and entity tags is to verify that state tokens have a protocol separator (`:`).

56c ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁56b 58b▷

```
if((hobj = cgi_header_obj("If"))) {
  char *cv = hdf_obj_value(hobj);
  gboolean rtag_allowed = *cv == '<';
  const char *path_info = cgi_header_val("CGI.PathInfo", "");
  const char *res = path_info;
  int orn = 0;
  char nbuf[22];
  while(1) {
    if(rtag_allowed && *cv == '<') {
      res = ++cv;
      while(*cv && *cv != '>')
        cv++;
      if(!*cv) {
        cv = NULL;
        break;
      }
```

56c ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁56b 58b▷

```
      *cv = 0;
      while (isspace(*++cv));
      ⟨Check and adjust resource URI res (imported)⟩
    }
    if (*cv != '(') {
      cv = NULL;
      break;
    }
    while (isspace(*++cv));
    int andn = 0;
    HDF *res_obj;
    sprintf(nbuf, "%d", orn++);
    die_if_err(hdf_get_node(hobj, nbuf, &res_obj));
    die_if_err(hdf_set_value(res_obj, NULL, res));
    if (res != path_info) {
      ⟨Check and adjust resource URI res_obj/res 58a⟩
      path_info = res;
    }
    while (1) {
      gboolean is_not = !strncasecmp(cv, "Not", 3);
      const char *sv;
      if (is_not) {
        cv += 2;
        while (isspace(*++cv));
      }
      if (*cv == '<') {
        sv = cv;
        if (is_not)
          *cv = '>';
        while (*++cv && *cv != '>');
        if (*cv != '>') {
          cv = NULL;
          break;
        }
        *cv = 0;
        if (!strchr(sv, ':')) {
          cv = NULL;
          break;
        }
      } else if (*cv == '[') {
        sv = cv;
        if (toupper(*++cv) == 'W' && cv[1] == '/')
          cv += 2;
        if (*cv != '"') {
          cv = NULL;
          break;
        }
        if (sv != cv) {
          sv = cv;
          *cv = is_not ? 'w' : 'W';
        } else if (is_not)
          *cv = '\'';
        char *e = cv + 1;
        while (*++cv && *cv != '"') {
          if (*cv == '\\') {
            if (!cv[1]) {
              cv = NULL;
              break;
            }
            cv++;
          }
          *e++ = *cv;
        }
        if (*cv != '"' || *++cv != ']') {
```

56c ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁56b 58b▷

```
            cv = NULL;
            break;
          }
          *e = 0;
        } else {
          cv = NULL;
          break;
        }
        sprintf(nbuf, "%d", andn++);
        die_if_err(hdf_set_value(res_obj, nbuf, sv));
        while(isspace(*++cv));
        if(*cv == ')')
          break;
      }
      if(!cv)
        break;
      while(isspace(*++cv));
      if(!*cv)
        break;
    }
    if(!cv) {
      cgi_env_set("error_status", "400 Bad Request");
      die_msg("Bad If syntax");
    }
  }
```

The resource tags actually specify entities which must be checked for the applicable conditions. They have the simple reference conditions described for the Destination header, except that in this case, the restriction of being the same or beneath the request URI is a requirement. Or at least that is what the RFC says; in actuality, it can also match the Destination header. In fact, some samples in the RFCs indicate that the path only needs to match after normalization, including sloppy hostname matching, resolution of home directories, and so forth. For this reason, the path is only subjected to standard normalization of absolute URLs, with no further processing. This at least requires that URLs be on the same host.

58a ⟨*Check and adjust resource URI* `res_obj/res` 58a⟩≡ (56c)

```
cgi_url_unescape(hdf_obj_value(res_obj));
die_if_err(normalize_path_obj(res_obj, cgi_state, NULL, TRUE, TRUE));
res = hdf_obj_value(res_obj);
```

The Lock-Token header specifies a lock token, which is a URL enclosed in angle brackets. The only check is that it contains a colon.

58b ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁56c 58c▷

```
if((hobj = cgi_header_obj("LockToken"))) {
  const char *cv = hdf_obj_value(hobj), *gt;
  if(*cv != '<' || !strchr(cv, ':') || !(gt = strrchr(cv, '>')) || gt[1]) {
    cgi_env_set("error_status", "400 Bad Request");
    die_msg("Invalid state token Lock-Token");
  }
}
```

The Overwrite header must match the value T or F. It is converted to upper-case.

58c ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁58b 59a▷

```
if((hobj = cgi_header_obj("Overwrite"))) {
  char *cv = hdf_obj_value(hobj);
```

58c ⟨*Process common HTTP headers 50*⟩+≡ (49a) ◁58b 59a▷

```
  if(cv[1])
    cv = NULL;
  else if(*cv == 't')
    *cv = 'T';
  else if(*cv == 'f')
    *cv = 'F';
  else if(*cv != 'T' && *cv != 'F')
    cv = NULL;
  if(!cv) {
    cgi_env_set("error_status", "400 Bad Request");
    die_msg("Overwrite must be T or F");
  }
}
```

The Timeout header consists of one or more comma-separated timeout values, which are a number of seconds or the word Infinite. If there is only one, it is simply checked for validity. If there is more than one, children are created with the values.

59a ⟨*Process common HTTP headers 50*⟩+≡ (49a) ◁58c 59b▷

```
if((hobj = cgi_header_obj("Timeout"))) {
  char *v = hdf_obj_value(hobj), *vs, *ve;
  int cno = 0;
  char nbuf[22];
  while(1) {
    vs = v;
    if(!strncasecmp(v, "Infinite", 8)) {
      v += 8;
    } else if(!strncasecmp(v, "Second-", 7) && isdigit(v[7])) {
      v += 7;
      while(isdigit(*++v));
    } else
      v= NULL;
    if(!v)
      break;
    ve = v;
    while(isspace(*v))
      v++;
    if(*v && *v != ',') {
      v = NULL;
      break;
    }
    if(cno || *v == ',') {
      char c = *ve;
      *ve = 0;
      sprintf(nbuf, "%d", cno++);
      die_if_err(hdf_set_value(hobj, nbuf, vs));
      *ve = c;
    }
    if(*v == ',')
      while(isspace(*++v));
    else
      break;
  }
  if(!v) {
    cgi_env_set("error_status", "400 Bad Request"); /* 422? */
    die_msg("Timeout is invalid");
  }
}
```

RFC 3253 defines the Label header, which is an arbitray URL-encoded string.

59b ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁59a 60▷

```
if((hobj = cgi_header_obj("Label"))) {
  char *cv = hdf_obj_value(hobj);

  cgi_url_unescape(cv);
}
```

Finally, RFC 4437 defines the Apply-To-Redirect-Ref header, which also takes either T or F. Since it applies universally, it is removed if it matches its default value of F.

60 ⟨*Process common HTTP headers* 50⟩+≡ (49a) ◁59b

```
if((hobj = cgi_header_obj("ApplyToRedirectRef"))) {
  char *cv = hdf_obj_value(hobj);

  if(cv[1])
    cv = NULL;
  else if(*cv == 't')
    *cv = 'T';
  else if(*cv == 'f')
    *cv = 'F';
  else if(*cv != 'T' && *cv != 'F')
    cv = NULL;
  if(!cv) {
    cgi_env_set("error_status", "400 Bad Request");
    die_msg("Apply-To-Redirect-Ref must be T or F");
  } else if(*cv == 'F')
    cgi_header_set("AplyToRedirectRef", NULL);
}
```

Chapter 8

XML Parameter Bodies

Another group of requests that take bodies is the set of WebDAV requests. These take XML bodies and special headers instead of plain CGI request parameters. What methods take what parameters, and how they are interpreted, is determined by which RFCs the underlying CGI supports. For this reason, if the method is anything but PUT, POST, HEAD, or GET, and the content type is XML, the XML is just sucked verbatim into an XML parse tree stored in the CGI state. This means that POST parameters with SOAP requests must be parsed manually after CGI initialization. The parsing is done using libxml2[1] using the document-at-once parser[2]. While incremental (xmlreader) mode would probably minimize the extra storage required for the XML tree, the size of the trees should be fairly small, anyway, and HDF is not much better for excess storage minimization (60-112 bytes per node vs. 60-120 bytes per node). A separate upload limit parameter is provided to limit incoming XML requests.

61a ⟨*CGI Support Configuration* 7e⟩+≡ (149b) ◁18b

```
# If set, limit XML request bodies (not PUT or POST) to this size, in bytes
#max_xml_body = 100000
```

61b ⟨*CGI Support Variables* 10c⟩+≡ (8e) ◁12d

```
guint64 max_xml_body;
```

61c ⟨*Perform global CGI support initialization* 10b⟩+≡ (9b) ◁12e

```
max_xml_body = getconf_int("max_xml_body", 100000);
xmlInitParser();
```

61d ⟨*(build.nw) makefile.vars* 7g⟩+≡ ◁29d 113e▷

```
EXTRA_CFLAGS += $(shell xml2-config --cflags)
EXTRA_LDFLAGS += $(shell xml2-config --libs)
```

61e ⟨*CGI Prerequisite Headers* 8c⟩+≡ (8b) ◁30a

```
#include <libxml/parser.h>
#include <libxml/xmlerror.h>
```

[1] http://xmlsoft.org/
[2] http://xmlsoft.org/html/libxml-parser.html

61f ⟨*CGI State Members* 9e⟩+≡ (8g) ◁44c 64f▷

```
xmlDocPtr xmlbody;
guint64 xmllen;
```

62a ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁49a 62d▷

```
/* actually, content-type should be text/xml or application/xml */
/* but we are not required to be that picky */
if(cgi_state->body_len && strstr(cgi_state->body_type, "/xml") &&
   req_method != HTTP_REQ_GET &&
   req_method != HTTP_REQ_POST &&
   req_method != HTTP_REQ_PUT &&
   req_method != HTTP_REQ_HEAD) {
  xmlDocPtr xml = xmlReadIO(xml_read_cgi, NULL, cgi_state, NULL, NULL,
                            XML_PARSE_NOENT | XML_PARSE_NONET |
                            XML_PARSE_NOWARNING | XML_PARSE_NOERROR |
                            XML_PARSE_NOCDATA | XML_PARSE_COMPACT);
  if(!xml) {
    xmlErrorPtr err = xmlGetLastError();
    if(err && err->code != XML_ERR_OK && err->code != XML_ERR_NO_MEMORY &&
       err->code != XML_ERR_INTERNAL_ERROR &&
       !cgi_env_val("error_status", NULL))
      cgi_env_set("error_status", "400 Bad Request");
    if(err && err->message) {
      nerr = nerr_raise_msg_errno(err->message);
      return nerr_pass_ctx(nerr, "Unable to parse body");
    } else
      return nerr_raise_msg_errno("Unable to parse body");
  }
  cgi_state->xmlbody = xml;
}
```

62b ⟨*Free CGI state members* 18e⟩+≡ (11g) ◁31a 97e▷

```
if(cgi_state->xmlbody)
  xmlFreeDoc(cgi_state->xmlbody);
```

62c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁49d 64b▷

```
int xml_read_cgi(void *context, char *buf, int len)
{
  cgi_state_t *cgi_state = context;
  NEOERR *nerr;

  if(cgi_state->xmllen + len > max_xml_body) {
    cgi_env_set("error_status", "413 Request Entity Too Large");
    errno = ENOMEM;
    return -1;
  }
  nerr = cgi_read(buf, len, &len);
  if(nerr != STATUS_OK) {
    nerr_ignore(&nerr);
    return -1;
  }
  return len;
}
```

Now that the XML parameters have been processed, it might be nice to print them if debugging is enabled.

62d ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁62a 63a▷

```
if(debug && cgi_state->xmlbody) {
  xmlChar *mem;
  int size;
  cgi_puts("<pre>XML:\n");
  xmlIndentTreeOutput = TRUE;
  xmlDocDumpFormatMemory(cgi_state->xmlbody, &mem, &size, TRUE);
  if(mem) {
    cgi_puts_html_escape(cgi_state, (const char *)mem, FALSE);
    xmlFree(mem);
  }
  cgi_puts("--------\n</pre>\n");
}
```

The WebDAV RFCs specify what constitutes valid XML bodies for the WebDAV requests, so they are validated here. That does mean this code will need to be modified if the RFCs change. Validating the XML with a DTD is impossible. Although a DTD-like specification is given in the RFCs, there are requirements which cannot be specified in the DTD language. For example, when specifying multiple elements as children, the DTD imposes an order, but WebDAV does not impose any ordering. Therefore, all checking is just done manually. Empty XML should probably be considered invalid, but instead it is treated as a missing body.

63a ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁62d 64g▷

```
gboolean valid;
xmlNodePtr node = NULL; /* init to shut gcc up */
xmlNsPtr dav_ns = NULL; /* cache for DAV: namespace */
if(!cgi_state->xmlbody || !(node = cgi_state->xmlbody->children))
  valid = TRUE ⟨unless method requires XML body 67b⟩;
else if(!cgi_state->xmlbody && cgi_state->body_len)
  valid = TRUE ⟨unless method requires no body or XML body 66b⟩;
else
  valid = TRUE;
if(valid && cgi_state->body_len)
  valid = TRUE ⟨unless method requires no body 96g⟩;
if(!valid) {
  cgi_env_set("error_status", "415 Unsupported Media Type");
  return nerr_raise_msg("Body not supported for this method");
} else if(cgi_state->xmlbody)
  switch(req_method) {
    ⟨Check WebDAV XML body 66c⟩
    default:
      break; /* ignore unknown methods */
  }
if(!valid) {
  cgi_env_set("error_status", "422 Unprocessable Entity"); /* RFC 4918 */
  return nerr_raise_msg("XML Invalid");
}
```

XML tag names are converted to integer constants to be stored in the `_private` field of the XML nodes. Name comparisons other than that cannot be cached.

63b ⟨*cgi-gperf-xml_name* 63b⟩≡

```
⟨XML validator names 64d⟩
```

63c ⟨*(build.nw) Known Data Types* 9a⟩+≡ ◁44b 98d▷

```
xml_name_t,%
```

64a ⟨*CGI Support Global Definitions* 8g⟩+≡ (8d) ◁28c 64e▷

```
#define xml_str_id(s) xml_name_id((const char *)s, strlen((const char *)s))
```

64b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁62c 64c▷

```
xml_name_t xml_tag(xmlNodePtr node)
{
  if(!node->_private)
    node->_private = (void *)(guint64)xml_str_id(node->name);
  return (xml_name_t)(guint64)node->_private;
}
```

Also, since most comparisons will be done in a single namespace, the namespace comparison can be done by comparing namespace pointers rather than the name itself. If the identifier being used changes from one tag to the next, though, only a direct name comparison can be used. Since there may be further checks against the `DAV:` namespace, it is stored in the CGI state as well. Although unset namespaces are technically not allowed, attributes should inherit the namespace from the tag, but do not in libxml2. Instead, if a NULL namespace is detected, it is always set to the cache.

64c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁64b 64h▷

```
gboolean check_ns(xmlNodePtr node, xml_name_t ns, xmlNsPtr *nscache)
{
  if(nscache && *nscache && node->ns == *nscache)
    return TRUE;
  if(!node->ns || !node->ns->href)
    return nscache && (node->ns = *nscache);
  if(xml_str_id(node->ns->href) != ns)
    return FALSE;
  if(nscache)
    *nscache = node->ns;
  return TRUE;
}
```

64d ⟨*XML validator names* 64d⟩≡ (63b) 66a▷

```
DAV:
```

64e ⟨*CGI Support Global Definitions* 8g⟩+≡ (8d) ◁64a 65a▷

```
#define DAV_NS XML_NAME_DAV_
```

64f ⟨*CGI State Members* 9e⟩+≡ (8g) ◁61f 97d▷

```
xmlNsPtr dav_ns;
```

64g ⟨*Initialize this CGI run* 22a⟩+≡ (21a) ◁63a

```
cgi_state->dav_ns = dav_ns;
```

Blanks and comments are mostly irrelevant, and get in the way. Comments never need to be saved, so they are just dropped. Just in case a processing instruction creeps through, they can be dropped as well. Blanks can be dropped everywhere except in property values: even if the XML declares that whitespace is not relevant, it is always relevant in property values. This means that things cannot be cleaned recursively very easily, but that is probably not necessary, anyway. Instead, a function to return the children with the ignorable elements stripped out is provided. Since comments may split text segments, any consecutive text segments resulting from the deletion are merged.

64h ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁64c 65b▷

```
xmlNodePtr xml_strip_children(xmlNodePtr node, gboolean strip_blanks)
{
  xmlNodePtr first = node->children, next;

  for(node = first; node; node = next) {
    next = node->next;
    switch(node->type) {
      case XML_TEXT_NODE:
        if(!strip_blanks || !xmlIsBlankNode(node))
          break;
        /* fall through */
      case XML_PI_NODE: /* processing instruction */
      case XML_COMMENT_NODE:
        if(first == node)
          first = next;
        xmlUnlinkNode(node);
        xmlFreeNode(node);
        if(next && next->prev &&
           xmlNodeIsText(next) && xmlNodeIsText(next->prev) &&
           (!strip_blanks || !xmlIsBlankNode(next))) {
          xmlTextMerge(next->prev, next);
          node = next;
          next = node->next;
          xmlUnlinkNode(node);
          xmlFreeNode(node);
        }
        break;
      default:
        break;
    }
  }
  return first;
}
```

65a ⟨*CGI Support Global Definitions* 8g⟩+≡ (8d) ◁64e 76c▷

```
#define xml_child(n) xml_strip_children(n, TRUE)
```

For properties only, a recursive function is provided to strip comments and processing instructions.

65b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁64h 66d▷

```
void xml_clean_prop(xmlNodePtr node)
{
  for(node = xml_strip_children(node, FALSE); node; node = node->next)
    xml_clean_prop(node);
}
```

The PROPFIND request takes an optional body containing exactly one propfind element.

65c ⟨*webdav.dtd* 65c⟩≡ 66e▷

```
<!-- PROPFIND method, optional body with exactly one propfind -->
<!ELEMENT propfind ( propname | (allprop, include?) | prop ) >
  <!ELEMENT propname EMPTY >
  <!ELEMENT allprop EMPTY >
  <!ELEMENT include ANY >
    <!-- children must be only empty tags -->
  <!ELEMENT prop ANY >
    <!-- children must be only empty tags -->
```

66a ⟨*XML validator names* 64d⟩+≡ (63b) ◁64d 67a▷

```
propfind propname allprop include prop
```

66b ⟨*unless method requires no body or XML body* 66b⟩≡ (63a) 68b▷

```
&& req_method != HTTP_REQ_PROPFIND
```

66c ⟨*Check WebDAV XML body* 66c⟩≡ (63a) 67c▷

```
case HTTP_REQ_PROPFIND:
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_PROPFIND || !check_ns(node, DAV_NS, &dav_ns) ||
     !(node = xml_child(node)) || node->type != XML_ELEMENT_NODE ||
     !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  if(valid) {
    xml_name_t tn = xml_tag(node);
    if(tn == XML_NAME_ALLPROP) {
      /* allprop + optional include */
      valid = !xml_child(node);
      if(node->next && node->next->type == XML_ELEMENT_NODE &&
         xml_tag(node->next) == XML_NAME_INCLUDE &&
         check_ns(node->next, DAV_NS, &dav_ns)) {
        node = node->next;
        valid = valid && check_prop(node, FALSE);
      }
      valid = valid && !node->next;
    } else if(tn == XML_NAME_INCLUDE)
      /* include + allprop */
      valid = node->next && !node->next->next && !xml_child(node->next) &&
              node->next->type == XML_ELEMENT_NODE &&
              xml_tag(node->next) == XML_NAME_ALLPROP &&
              check_ns(node->next, DAV_NS, &dav_ns) && check_prop(node, FALSE);
    else if(tn == XML_NAME_PROP)
      valid = !node->next && check_prop(node, FALSE);
    else if(tn == XML_NAME_PROPNAME)
      valid = !node->next && !xml_child(node);
  }
  break;
```

66d ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁65b 72a▷

```
static gboolean check_prop(xmlNodePtr node, gboolean has_val)
{
  if(!xml_child(node))
    return FALSE;
  for(node = node->children; node; node = node->next)
    if(node->type != XML_ELEMENT_NODE || (!has_val && xml_child(node)) ||
       !node->ns || !node->ns->href)
      return FALSE;
  return TRUE;
}
```

The PROPPATCH method requires an XML body with exactly one propertyupdate element.

66e ⟨*webdav.dtd* 65c⟩+≡ ◁65c 67d▷

```
<!-- PROPPATCH method, required XML body, exactly one propertyupdate -->
<!ELEMENT propertyupdate (remove | set)+ >
  <!ELEMENT remove (prop) >
      <!ELEMENT prop ANY >
        <!-- children must be only empty tags -->
   <!ELEMENT set (prop) >
      <!ELEMENT prop ANY >
        <!-- children must be tags with valid XML values -->
```

67a ⟨*XML validator names* 64d⟩+≡ (63b) ◁66a 68a▷

```
propertyupdate remove set
```

67b ⟨*unless method requires XML body* 67b⟩≡ (63a) 69c▷

```
&& req_method != HTTP_REQ_PROPPATCH
```

67c ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁66c 68c▷

```
case HTTP_REQ_PROPPATCH:
  /* exactly one child element called propertyupdate, with at least one child */
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_PROPERTYUPDATE || !xml_child(node) ||
     !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  for(node = node->children; node && valid; node = node->next) {
    xmlNodePtr c;
    gboolean no_val = FALSE; /* init to shut gcc up */
    xml_name_t tn;
    /* one or more set/reset nodes */
    if(node->type != XML_ELEMENT_NODE || !check_ns(node, DAV_NS, &dav_ns))
      valid = FALSE;
    else if((tn = xml_tag(node)) == XML_NAME_REMOVE)
      no_val = TRUE;
    else if(tn == XML_NAME_SET)
      no_val = FALSE;
    else
      valid = FALSE;
    /* with a single prop child whose children are property tags */
    if(valid && (!(c = xml_child(node)) || c->next ||
                 c->type != XML_ELEMENT_NODE ||
                 !check_ns(c, DAV_NS, &dav_ns) ||
                 xml_tag(c) != XML_NAME_PROP || !check_prop(c, !no_val)))
      valid = FALSE;
  }
  break;
```

The LOCK method takes an optional body with exactly one lockinfo element.

67d ⟨*webdav.dtd* 65c⟩+≡ ◁66e 68d▷

```
<!-- LOCK method, optional body, exactly one lockinfo (def: refresh) -->
<!ELEMENT lockinfo (lockscope, locktype, owner?)  >
  <!ELEMENT lockscope (exclusive | shared) >
    <!ELEMENT exclusive EMPTY >  <!-- empty tag flag (what a waste) -->
    <!ELEMENT shared EMPTY ><!-- empty tag flag -->
  <!ELEMENT locktype (write) >
    <!ELEMENT write EMPTY ><!-- empty tag flag -->
  <!ELEMENT owner ANY ><!-- preserve value verbatim like a property -->
```

68a ⟨*XML validator names* 64d⟩+≡ (63b) ◁67a 69b▷

```
lockinfo lockscope locktype owner exclusive shared write
```

68b ⟨*unless method requires no body or XML body* 66b⟩+≡ (63a) ◁66b 75c▷

```
&& req_method != HTTP_REQ_LOCK
```

68c ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁67c 69d▷

```
case HTTP_REQ_LOCK:
  /* exactly one child element called lockinfo, with at least one child */
  if (node->next || node->type != XML_ELEMENT_NODE || !xml_child(node) ||
      xml_tag(node) != XML_NAME_LOCKINFO || !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  if (valid) {
    gboolean got_ls = FALSE, got_lt = FALSE, got_o = FALSE;
    for (node = node->children; node && valid; node = node->next) {
      xmlNodePtr c;
      xml_name_t tn;
      if (node->type != XML_ELEMENT_NODE || !(c = xml_child(node)) ||
          !check_ns(node, DAV_NS, &dav_ns))
        valid = FALSE;
      else if ((tn = xml_tag(node)) == XML_NAME_LOCKSCOPE) {
        valid = !got_ls;
        got_ls = TRUE;
        if (c->type != XML_ELEMENT_NODE || c->next || xml_child(c) ||
            !check_ns(c, DAV_NS, &dav_ns) ||
            ((tn = xml_tag(c)) != XML_NAME_EXCLUSIVE && tn != XML_NAME_SHARED))
          valid = FALSE;
      } else if (tn == XML_NAME_LOCKTYPE) {
        valid = !got_lt;
        got_lt = TRUE;
        if (c->type != XML_ELEMENT_NODE || c->next || xml_child(c) ||
            xml_tag(c) != XML_NAME_WRITE || !check_ns(c, DAV_NS, &dav_ns))
          valid = FALSE;
      } else if (tn == XML_NAME_OWNER) {
        valid = !got_o;
        got_o = TRUE;
      } else
        valid = FALSE;
    }
    if (!got_ls || !got_lt)
      valid = FALSE;
  }
  break;
```

The SEARCH method requires a body with exactly one element which is either searchrequest or query-schema-discovery. Each child is an element whose name is that of a supported search method; the only search method which is checked for validity is the basicsearch method. It is up to the application to validate and throw errors for unsupported search methods. The specification is unclear on how to handle multiple search methods in a single request, but the implication is that only one should be supported, so that is what is enforced here.

68d ⟨*webdav.dtd* 65c⟩+≡ ◁67d 73b▷

```
<!-- SEARCH method, required body with searchrequest or query-schema-discovery -->
<!ELEMENT searchrequest ANY ><!-- one child whose name determines grammar -->
  ⟨basicsearch DTD 69a⟩
<!ELEMENT query-schema-discovery ANY>
  <!-- child is query grammar and scope -->
  <!ELEMENT basicsearch   (from) >
```

69a ⟨*basicsearch DTD* 69a⟩≡ (68d)

```
<!ELEMENT basicsearch   (select, from, where?, orderby?, limit?) >
  <!ELEMENT select        (allprop | prop) >
    <!-- props are just tags, and allprop is a flag -->
  <!ELEMENT from          (scope+) >
    <!ELEMENT scope         (href, depth, include-versions?) >
    <!ELEMENT include-versions EMPTY >
  <!-- where -->
    <!ENTITY % all_ops      "%comp_ops; | %log_ops; | %special_ops; |
                             %string_ops; | %content_ops;">
    <!ENTITY % comp_ops     "eq | lt | gt| lte | gte">
    <!ENTITY % log_ops      "and | or | not">
    <!ENTITY % special_ops  "is-collection | is-defined |
                             language-defined | language-matches">
    <!ENTITY % string_ops   "like">
    <!ENTITY % content_ops  "contains">
  <!ELEMENT where         ( %all_ops; ) >
      <!ELEMENT and           ( %all_ops; )+ >
      <!ELEMENT or            ( %all_ops; )+ >
      <!ELEMENT not           ( %all_ops; ) >
      <!ELEMENT lt            (prop, (literal|typed-literal)) >
        <!ATTLIST lt            caseless   (yes|no) #IMPLIED>
      <!ELEMENT lte           (prop, (literal|typed-literal)) >
        <!ATTLIST lte           caseless   (yes|no) #IMPLIED>
      <!ELEMENT gt           (prop, (literal|typed-literal)) >
        <!ATTLIST gt           caseless   (yes|no) #IMPLIED>
      <!ELEMENT gte           (prop, (literal|typed-literal)) >
        <!ATTLIST gte           caseless   (yes|no) #IMPLIED>
      <!ELEMENT eq           (prop, (literal|typed-literal)) >
        <!ATTLIST eq           caseless   (yes|no) #IMPLIED>
      <!ELEMENT literal       (#PCDATA)>
      <!ELEMENT typed-literal (#PCDATA)>
        <!ATTLIST typed-literal xsi:type CDATA #IMPLIED>
      <!ELEMENT is-collection EMPTY >
      <!ELEMENT is-defined    (prop) >
      <!ELEMENT language-defined    (prop) >
      <!ELEMENT language-matches    (prop, literal) >
      <!ELEMENT like          (prop, literal) >
        <!ATTLIST like          caseless   (yes|no) #IMPLIED>
      <!ELEMENT contains      (#PCDATA)>
  <!ELEMENT orderby       (order+) >
    <!ELEMENT order         ((prop | score), (ascending | descending)?)>
      <!ATTLIST order         caseless   (yes|no) #IMPLIED>
      <!ELEMENT ascending     EMPTY>
      <!ELEMENT descending    EMPTY>
  <!ELEMENT limit         (nresults) >
    <!ELEMENT nresults      (#PCDATA) >
```

69b ⟨*XML validator names* 64d⟩+≡ (63b) ◁68a 74a▷

```
searchrequest query-schema-discovery basicsearch select from scope href
depth 0 1 infinity include-versions where and or not lt caseless yes no lte
gt gte eq literal typed-literal is-collection is-defined language-defined
language-matches like contains orderby order score ascending descending
limit nresults
```

69c ⟨*unless method requires XML body* 67b⟩+≡ (63a) ◁67b 76b▷

```
&& req_method != HTTP_REQ_SEARCH
```

69d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁68c 74b▷

```
case HTTP_REQ_SEARCH: {
  /* exactly one child element called searchrequest or query-schema-discovery,
       with at least one child */
  xml_name_t tn;
  if(node->next || node->type != XML_ELEMENT_NODE ||
      ((tn = xml_tag(node)) != XML_NAME_SEARCHREQUEST &&
       tn != XML_NAME_QUERY_SCHEMA_DISCOVERY) ||
      !check_ns(node, DAV_NS, &dav_ns) || !xml_child(node) ||
      node->children->type != XML_ELEMENT_NODE)
    valid = FALSE;
  gboolean is_search = valid && tn == XML_NAME_SEARCHREQUEST;
  node = node->children;
  /* searchrequest has exactly one child element specifying search */
  /* not sure if query-schema-discovery should support >1 child element */
  valid = valid && !node->next; /* not sure this is made clear in RFC */
  /* make it a loop in case above condition is removed */
  for( ; node; node = node->next) {
    xmlNodePtr c;
    if(node->type != XML_ELEMENT_NODE || !(c = xml_child(node))) {
      valid = FALSE;
      break;
    }
    /* name check is done after ns check so _private isn't set if it isn't */
    /* DAV:basicsearch; this allows the client to avoid check_ns */
    if(!check_ns(node, DAV_NS, &dav_ns) ||
       xml_tag(node) != XML_NAME_BASICSEARCH)
      continue;
    gboolean got_sel, got_from, got_where, got_oby, got_limit;
    got_sel = got_where = got_oby = got_limit = !is_search;
    got_from = FALSE;
    for(; c && valid; c = c->next) {
      xmlNodePtr subc;
      if(!check_ns(c, DAV_NS, &dav_ns) || !(subc = xml_child(c)) ||
         subc->type != XML_ELEMENT_NODE || !check_ns(subc, DAV_NS, &dav_ns)) {
        valid = FALSE;
        break;
      }
      if((tn = xml_tag(c)) == XML_NAME_SELECT) {
        valid = !got_sel && !subc->next;
        got_sel = TRUE;
        if((tn = xml_tag(subc)) == XML_NAME_ALLPROP)
          valid = valid && !xml_child(subc);
        else if(tn == XML_NAME_PROP)
          valid = check_prop(subc, FALSE);
        else
          valid = FALSE;
      } else if(tn == XML_NAME_FROM) {
        valid = !got_from;
        got_from = TRUE;
        xmlNodePtr scope;
        for(; subc && valid; subc = subc->next) {
          valid = subc->type == XML_ELEMENT_NODE && (scope = xml_child(subc)) &&
                  xml_tag(subc) == XML_NAME_SCOPE &&
                  check_ns(subc, DAV_NS, &dav_ns);
          gboolean got_href = FALSE, got_depth = FALSE, got_incv = FALSE;
          for(; scope && valid; scope = scope->next) {
            if(!check_ns(scope, DAV_NS, &dav_ns))
              valid = FALSE;
            else if((tn = xml_tag(scope)) == XML_NAME_HREF) {
              valid = !got_href && xml_child(scope) && !scope->children->next &&
                      xmlNodeIsText(scope->children);
              got_href = TRUE;
            } else if(tn == XML_NAME_DEPTH) {
              valid = !got_depth;
```

69d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁68c 74b▷

```
            got_depth = TRUE;
            xmlNodePtr val;
            if(valid && (val = xml_child(scope)))
              valid = !val->next && xmlNodeIsText(val) &&
                      ((tn = xml_str_id(val->content)) == XML_NAME_0 ||
                       tn == XML_NAME_1 || tn == XML_NAME_INFINITY);
          } else if(tn == XML_NAME_INCLUDE_VERSIONS) {
            valid = !got_incv && !xml_child(scope);
            got_incv = TRUE;
          } else
            valid = FALSE;
        }
        if(!got_href || !got_depth)
          valid = FALSE;
      }
    } else if(tn == XML_NAME_WHERE) {
      valid = !got_where && check_where_ops(subc, &dav_ns);
      got_where = TRUE;
    } else if(tn == XML_NAME_ORDERBY) {
      valid = !got_oby;
      got_oby = TRUE;
      for(; subc && valid; subc = subc->next) {
        xmlNodePtr order;
        if(!xml_child(subc) || xml_tag(subc) != XML_NAME_ORDER ||
           !check_ns(subc, DAV_NS, &dav_ns))
          valid = FALSE;
        else {
          gboolean got_prop = FALSE, got_asc = FALSE;
          for(order = xml_child(subc); order; order = order->next) {
            if(!check_ns(order, DAV_NS, &dav_ns))
              valid = FALSE;
            else if((tn = xml_tag(order)) == XML_NAME_PROP) {
              valid = !got_prop && check_prop(order, FALSE);
              got_prop = TRUE;
            } else if(tn == XML_NAME_SCORE) {
              valid = !got_prop && xml_child(order) &&
                      !order->children->next &&
                      xmlNodeIsText(order->children);
              got_prop = TRUE;
              if(valid) {
                const char *s = (const char *)order->children->content;

                for(; *s; s++)
                  if(!isdigit(*s))
                    valid = FALSE;
              }
            } else if(tn == XML_NAME_ASCENDING ||
                      tn == XML_NAME_DESCENDING) {
              valid = !got_asc && !xml_child(order);
              got_asc = TRUE;
            } else
              valid = FALSE;
          }
        }
        if(valid)
          valid = check_caseless_attr(subc, &dav_ns);
      }
    } else if(tn == XML_NAME_LIMIT) {
      valid = !got_limit && !subc->next && xml_child(subc) &&
              !subc->children->next && xmlNodeIsText(subc->children) &&
              xml_tag(subc) == XML_NAME_NRESULTS &&
              check_ns(subc, DAV_NS, &dav_ns);
      got_limit = TRUE;
      const char *s;
```

69d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁68c 74b▷

```
        for(s = (const char *)subc->children->content; *s; s++)
          if(!isdigit(*s))
            valid = FALSE;
      } else
        valid = FALSE;
    }
    if(!got_sel || !got_from)
      valid = FALSE;
  }
  break;
}
```

72a ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁66d 72b▷

```
static gboolean check_caseless_attr(xmlNodePtr node, xmlNsPtr *nsc)
{
  xmlAttrPtr attrs;
  gboolean valid = TRUE;
  xml_name_t tn;

  for(attrs = node->properties; attrs && valid; attrs = attrs->next)
    if(xml_tag((xmlNodePtr)attrs) == XML_NAME_CASELESS &&
       check_ns((xmlNodePtr)attrs, DAV_NS, nsc))
      valid = attrs->children && !attrs->children->next &&
              xmlNodeIsText(attrs->children) &&
              ((tn = xml_str_id(attrs->children->content)) == XML_NAME_NO ||
               tn == XML_NAME_YES);
  return valid;
}
```

72b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁72a 78c▷

```
static gboolean check_where_ops(xmlNodePtr node, xmlNsPtr *nsc)
{
  xmlNodePtr c;
  xml_name_t tn;

  if(!node)
    return FALSE;
  for(; node; node = node->next) {
    if(node->type != XML_ELEMENT_NODE || !check_ns(node, DAV_NS, nsc))
      return FALSE;
    c = xml_child(node);
    tn = xml_tag(node);
    /* comp_ops */
    if(tn == XML_NAME_EQ  || tn == XML_NAME_LT || tn == XML_NAME_GT ||
       tn == XML_NAME_LTE || tn == XML_NAME_GTE) {
      if(!c || !c->next || c->next->next || c->type != XML_ELEMENT_NODE ||
         c->next->type != XML_ELEMENT_NODE || !check_ns(c, DAV_NS, nsc) ||
         !check_ns(c->next, DAV_NS, nsc))
        return FALSE;
      xmlNodePtr p;
      if((tn = xml_tag(c)) == XML_NAME_PROP) {
        p = c;
        c = c->next;
        tn = xml_tag(c);
      } else {
        p = c->next;
        if(xml_tag(p) != XML_NAME_PROP)
          return FALSE;
      }
      if(!xml_child(p) || p->children->next || !check_prop(p, FALSE))
        return FALSE;
```

72b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁72a 78c▷

```
      if((xml_child(c) && (c->children->next || !xmlNodeIsText(c->children))) ||
         (tn != XML_NAME_LITERAL && tn != XML_NAME_TYPED_LITERAL) ||
         !check_caseless_attr(node, nsc))
        return FALSE;
    /* log_ops */
    } else if(tn == XML_NAME_AND || tn == XML_NAME_OR) {
      if(!check_where_ops(c, nsc))
        return FALSE;
    } else if(tn == XML_NAME_NOT) {
      if(!c || c->next || !check_where_ops(c, nsc))
        return FALSE;
    /* special_ops */
    } else if(tn == XML_NAME_IS_COLLECTION) {
      if(c)
        return FALSE;
    } else if(tn == XML_NAME_IS_DEFINED || tn == XML_NAME_LANGUAGE_DEFINED) {
      if(!c || c->next || !check_ns(c, DAV_NS, nsc) ||
         !(c = xml_child(c)) || c->type != XML_ELEMENT_NODE ||
         !check_ns(c, DAV_NS, nsc) || xml_tag(c) != XML_NAME_PROP ||
         !check_prop(c, FALSE))
        return FALSE;
    } else if(tn == XML_NAME_LANGUAGE_MATCHES) {
      ⟨Check c for prop and literal 73a⟩
    /* string_ops */
    } else if(tn == XML_NAME_LIKE) {
      ⟨Check c for prop and literal 73a⟩
      if(!check_caseless_attr(node, nsc))
        return FALSE;
    /* content_ops */
    } else if(tn == XML_NAME_CONTAINS) {
      if(c && (c->next || !xmlNodeIsText(c)))
        return FALSE;
    } else
      return FALSE;
  }
  return TRUE;
}
```

73a ⟨*Check* c *for prop and literal* 73a⟩≡ (72b)

```
if(!c || !xml_child(c) || c->children->next || !c->next ||
   !xml_child(c->next) || c->next->children->next || c->next->next ||
   !check_ns(c, DAV_NS, nsc) || !check_ns(c->next->next, DAV_NS, nsc))
  return FALSE;
if((tn = xml_tag(c)) == XML_NAME_LITERAL) {
  if(!xmlNodeIsText(c->children))
    return FALSE;
  c = c->next;
  tn = xml_tag(c);
}
if(tn != XML_NAME_PROP || !check_prop(c, FALSE))
  return FALSE;
if((c = c->next) && (xml_tag(c) != XML_NAME_LITERAL ||
                     !xmlNodeIsText(c->children)))
  return FALSE;
```

The ACL method requires an XML body with exactly one acl element.

73b ⟨*webdav.dtd* 65c⟩+≡ ◁68d 75a▷

```
<!-- ACL method, required body with single acl element -->
<!ELEMENT acl (ace*) >
  <!ELEMENT ace ((principal | invert), (grant|deny))>
    <!ELEMENT principal (href | all | authenticated | unauthenticated |
                         property | self)>
      <!ELEMENT all EMPTY>
      <!ELEMENT authenticated EMPTY>
      <!ELEMENT unauthenticated EMPTY>
      <!ELEMENT property ANY>
        <!-- child is exactly one property name -->
      <!ELEMENT self EMPTY>
    <!ELEMENT invert principal>
    <!ELEMENT grant (privilege+)>
    <!ELEMENT deny (privilege+)>
      <!ELEMENT privilege ANY> <!-- actually, only valid empty priv tags -->
        <!ELEMENT read EMPTY>
        <!ELEMENT write EMPTY>
        <!ELEMENT write-properties EMPTY>
        <!ELEMENT write-content EMPTY>
        <!ELEMENT unlock EMPTY>
        <!ELEMENT read-acl EMPTY>
        <!ELEMENT read-current-user-privilege-set EMPTY>
        <!ELEMENT write-acl EMPTY>
        <!ELEMENT bind EMPTY>
        <!ELEMENT unbind EMPTY>
        <!ELEMENT all EMPTY>
```

74a ⟨*XML validator names* 64d⟩+≡ (63b) ◁69b 75b▷

```
acl ace principal invert grant deny all authenticated unauthenticated
property self privilege read write-properties write-content unlock
read-acl read-current-user-privilege-set write-acl bind unbind
```

74b ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁69d 76a▷

```
case HTTP_REQ_ACL:
  /* exactly one child element called acl, with zero or more ace children */
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_ACL || !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  else
    for(node = xml_child(node); node && valid; node = node->next) {
      xmlNodePtr c;
      /* 2 children: [!]principal, grant\deny */
      if(node->type != XML_ELEMENT_NODE || !(c = xml_child(node)) ||
         !c->next || c->next->next || c->type != XML_ELEMENT_NODE ||
         c->next->type != XML_ELEMENT_NODE ||
         !check_ns(node, DAV_NS, &dav_ns) || xml_tag(node) != XML_NAME_ACE ||
         !check_ns(c, DAV_NS, &dav_ns) || !check_ns(c->next, DAV_NS, &dav_ns))
        valid = FALSE;
      else {
        gboolean got_g = FALSE;
        xmlNodePtr p = NULL;
        xml_name_t tn;
        for(; c && valid; c = c->next) {
          if(!xml_child(c))
            valid = FALSE;
          else if((tn = xml_tag(c)) == XML_NAME_INVERT) {
            if(p || c->children->next || !xml_child(c->children) ||
               c->children->type != XML_ELEMENT_NODE ||
               !check_ns(c->children, DAV_NS, &dav_ns) ||
               xml_tag(c->children) != XML_NAME_PRINCIPAL)
              valid = FALSE;
```

74b ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁69d 76a▷

```
            else
              p = c->children->children;
          } else if(tn == XML_NAME_PRINCIPAL) {
            if(p)
              valid = FALSE;
            else
              p = c->children;
          } else if(tn == XML_NAME_GRANT || tn == XML_NAME_DENY) {
            valid = !got_g;
            got_g = TRUE;
            xmlNodePtr priv;
            for(priv = c->children; priv && valid; priv = priv->next)
              /* each privilege tag has a single empty child tag */
              /* each child tag can be anything, just like a property */
              if(priv->type != XML_ELEMENT_NODE || !xml_child(priv) ||
                 priv->children->next || !check_prop(priv, FALSE) ||
                 xml_tag(priv) != XML_NAME_PRIVILEGE ||
                 !check_ns(priv, DAV_NS, &dav_ns))
                valid = FALSE;
            continue;
          } else
            valid = FALSE;
          if(valid) { /* finish principal processing */
            if(p->next || p->type != XML_ELEMENT_NODE ||
               !check_ns(p, DAV_NS, &dav_ns))
              valid = FALSE;
            else if((tn = xml_tag(p)) == XML_NAME_HREF)
              valid = xml_child(p) && !p->children->next &&
                      xmlNodeIsText(p->children);
            else if(tn == XML_NAME_ALL || tn == XML_NAME_AUTHENTICATED ||
                    tn == XML_NAME_UNAUTHENTICATED || tn == XML_NAME_SELF)
              valid = !p->children;
            else if(tn == XML_NAME_PROPERTY)
              valid = xml_child(p) && !p->children->next && check_prop(p, FALSE);
            else
              valid = FALSE;
          }
        }
        if(!got_g || !p)
          valid = FALSE;
      }
    }
  break;
```

According to RFC 4918, MKCOL takes no body. However, RFC 5689[3] allows a body, which must be a single mkcol element.

75a ⟨*webdav.dtd* 65c⟩+≡ ◁73b 77b▷

```
<!-- MKCOL method, optional XML body, exactly one mkcol -->
<!ELEMENT mkcol (set+)>
   <!ELEMENT set (prop) >
      <!ELEMENT prop ANY >
        <!-- children must be tags with valid XML values -->
```

75b ⟨*XML validator names* 64d⟩+≡ (63b) ◁74a 77a▷

```
mkcol
```

[3] http://www.ietf.org/rfc/rfc5689.txt

75c ⟨*unless method requires no body or XML body* 66b⟩+≡ (63a) ◁68b 87d▷

```
&& req_method != HTTP_REQ_MKCOL
```

76a ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁74b 76d▷

```
case HTTP_REQ_MKCOL:
  /* exactly one child element called propertyupdate, with at least one child */
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_MKCOL || !xml_child(node) ||
     !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  for(node = node->children; node && valid; node = node->next) {
    xmlNodePtr c;
    /* one or more set nodes */
    /* with a single prop child whose children are property tags */
    valid = node->type == XML_ELEMENT_NODE && (c = xml_child(node)) &&
            !c->next && c->type == XML_ELEMENT_NODE &&
            check_ns(node, DAV_NS, &dav_ns) && check_ns(c, DAV_NS, &dav_ns) &&
            xml_tag(node) == XML_NAME_SET && xml_tag(c) == XML_NAME_PROP &&
            check_prop(c, TRUE);
  }
  break;
```

The REPORT method requires an XML body, but allows any single root tag as its body content. Several RFCs define bodies, though. Unfortunately, one of those defines a different namespace (urn:ietf:params:xml:ns:caldav), as well, so this can't be a simple switch after checking for the usual DAV: namespace. Any report types not covered by known RFCs must be dealt with by the application.

76b ⟨*unless method requires XML body* 67b⟩+≡ (63a) ◁69c 90a▷

```
&& req_method != HTTP_REQ_REPORT
```

76c ⟨*CGI Support Global Definitions* 8g⟩+≡ (8d) ◁65a 107a▷

```
#define CALDAV_NS XML_NAME_URN_IETF_PARAMS_XML_NS_CALDAV
```

76d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁76a 87e▷

```
case HTTP_REQ_REPORT:
  if(node->next || node->type != XML_ELEMENT_NODE) {
    valid = FALSE;
    break;
  }
  if(check_ns(node, DAV_NS, &dav_ns))
    switch(xml_tag(node)) {
      ⟨Check known DAV: report types 77d⟩
      default:
        break; /* ignore unknown tags */
    }
  else {
    xmlNsPtr caldav_ns = NULL;
    if(check_ns(node, CALDAV_NS, &caldav_ns))
      switch(xml_tag(node)) {
        ⟨Check known CALDAV: report types 81c⟩
        default:
          break; /* ignore unkown tags */
      }
  }
  break;
```

77a ⟨*XML validator names* 64d⟩+≡ (63b) ◁75b 77c▷

```
urn:ietf:params:xml:ns:caldav
```

RFC 3253 defines a number of version control-related report types.

77b ⟨*webdav.dtd* 65c⟩+≡ ◁75a 79a▷

```
<!ELEMENT version-tree ANY>
  <!-- 0 or more elements, at most one DAV:prop -->
  <!-- other elements undefined -->
<!ELEMENT expand-property (property*)>
  <!ELEMENT property (property*)>
  <!ATTLIST property name NMTOKEN #REQUIRED>
  <!ATTLIST property namespace NMTOKEN "DAV:">
<!ELEMENT locate-by-history (version-history-set, prop)>
  <!ELEMENT version-history-set (href+)>
<!ELEMENT merge-preview (source)>
  <!ELEMENT source (href)>
<!ELEMENT compare-baseline (href)>
<!ELEMENT latest-activity-version (href)>
```

77c ⟨*XML validator names* 64d⟩+≡ (63b) ◁77a 79b▷

```
version-tree expand-property locate-by-history merge-preview
compare-baseline latest-activity-version
name namespace version-history-set
```

77d ⟨*Check known DAV: report types* 77d⟩≡ (76d) 77e▷

```
case XML_NAME_VERSION_TREE:
  if((node = xml_child(node))) {
    gboolean got_prop = FALSE;
    for(; valid && node; node = node->next) {
      valid = node->type == XML_ELEMENT_NODE;
      if(valid && check_ns(node, DAV_NS, &dav_ns) &&
         xml_tag(node) == XML_NAME_PROP) {
        valid = !got_prop && check_prop(node, FALSE);
        got_prop = TRUE;
      }
    }
  }
  break;
```

77e ⟨*Check known DAV: report types* 77d⟩+≡ (76d) ◁77d 77f▷

```
case XML_NAME_EXPAND_PROPERTY:
  valid = check_property(xml_child(node), &dav_ns);
  break;
```

77f ⟨*Check known DAV: report types* 77d⟩+≡ (76d) ◁77e 78a▷

```
case XML_NAME_LOCATE_BY_HISTORY:
  valid = (node = xml_child(node)) && node->type == XML_ELEMENT_NODE &&
          node->next && node->next->type == XML_ELEMENT_NODE &&
          !node->next->next && check_ns(node, DAV_NS, &dav_ns) &&
          check_ns(node->next, DAV_NS, &dav_ns);
  if(valid) {
    xml_name_t tn = xml_tag(node);
    if(tn == XML_NAME_PROP) {
      valid = check_prop(node, FALSE);
      node = node->next;
```

77f ⟨*Check known DAV: report types* 77d⟩+≡ (76d) ◁77e 78a▷

```
      tn = xml_tag(node);
    } else
      valid = xml_tag(node->next) == XML_NAME_PROP &&
              check_ns(node->next, DAV_NS, &dav_ns) &&
              check_prop(node->next, FALSE);
    if(valid)
      valid = xml_tag(node) == XML_NAME_VERSION_HISTORY_SET &&
              xml_child(node);
    for(node = xml_child(node); node && valid; node = node->next)
      valid = node->type == XML_ELEMENT_NODE &&
              xml_tag(node) == XML_NAME_HREF &&
              check_ns(node, DAV_NS, &dav_ns) && xml_child(node) &&
              xmlNodeIsText(node->children);
  }
  break;
```

78a ⟨*Check known DAV: report types* 77d⟩+≡ (76d) ◁77f 78b▷

```
case XML_NAME_MERGE_PREVIEW:
  valid = (node = xml_child(node)) && node->type == XML_ELEMENT_NODE &&
          xml_tag(node) == XML_NAME_SOURCE &&
          check_ns(node, DAV_NS, &dav_ns) && (node = xml_child(node)) &&
          !node->next && node->type == XML_ELEMENT_NODE &&
          xml_tag(node) == XML_NAME_HREF && xml_child(node) &&
          !node->children->next && xmlNodeIsText(node->children) &&
          check_ns(node, DAV_NS, &dav_ns);
  break;
```

78b ⟨*Check known DAV: report types* 77d⟩+≡ (76d) ◁78a 79c▷

```
case XML_NAME_COMPARE_BASELINE:
case XML_NAME_LATEST_ACTIVITY_VERSION:
  valid = (node = xml_child(node)) && node->type == XML_ELEMENT_NODE &&
          xml_tag(node) == XML_NAME_HREF && xml_child(node) &&
          !node->children->next && xmlNodeIsText(node->children) &&
          check_ns(node, DAV_NS, &dav_ns);
  break;
```

78c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁72b 82▷

```
static gboolean check_property(xmlNodePtr node, xmlNsPtr *nsc)
{
  for(; node; node = node->next) {
    xmlAttrPtr attrs;
    gboolean got_name = FALSE;
    xml_name_t an;

    if(node->type != XML_ELEMENT_NODE || xml_tag(node) != XML_NAME_PROPERTY ||
       !check_ns(node, DAV_NS, nsc))
      return FALSE;
    for(attrs = node->properties; attrs; attrs = attrs->next)
      if(check_ns((xmlNodePtr)attrs, DAV_NS, nsc) &&
         ((an = xml_tag((xmlNodePtr)attrs)) == XML_NAME_NAME ||
          an == XML_NAME_NAMESPACE)) {
        /* no attempt to validate the actual text value as a name token */
        if(!attrs->children || attrs->children->next ||
           !xmlNodeIsText(attrs->children))
          return FALSE;
        if(an == XML_NAME_NAME)
          got_name = TRUE;
      }
    if(!got_name)
      return FALSE;
```

78c ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁72b 82▷

```
    if(!check_property(xml_child(node), nsc))
      return FALSE;
  }
  return TRUE;
}
```

RFC 3744 defines a few ACL-related report types.

79a ⟨*webdav.dtd* 65c⟩+≡ ◁77b 81a▷

```
<!ELEMENT acl-principal-prop-set ANY>
  <!-- one or more children, with at most one prop element -->
  <!-- all others to be ignored.  Note that lack of prop makes no sense -->
<!ELEMENT principal-match ((principal-property | self), prop?)>
  <!ELEMENT principal-property ANY>
  <!-- ANY: an element whose value identifies a property -->
  <!ELEMENT self EMPTY>
<!ELEMENT principal-property-search
    ((property-search+), prop?, apply-to-principal-collection-set?) >
  <!ELEMENT property-search (prop, match) >
    <!ELEMENT match #PCDATA >
  <!-- apply-to-principal-collection-set unspecified, but presumed EMPTY -->
<!ELEMENT principal-search-property-set EMPTY >
```

79b ⟨*XML validator names* 64d⟩+≡ (63b) ◁77c 81b▷

```
acl-principal-prop-set principal-match principal-property-search
principal-search-property-set
principal-property property-search match apply-to-principal-collection-set
```

79c ⟨*Check known DAV: report types* 77d⟩+≡ (76d) ◁78b 79d▷

```
case XML_NAME_ACL_PRINCIPAL_PROP_SET:
  if(!xml_child(node))
    valid = FALSE;
  else {
    gboolean got_prop = FALSE;
    for(node = node->children; node && valid; node = node->next) {
      valid = node->type == XML_ELEMENT_NODE;
      if(valid && xml_tag(node) == XML_NAME_PROP &&
         check_ns(node, DAV_NS, &dav_ns)) {
        valid = !got_prop && check_prop(node, FALSE);
        got_prop = TRUE;
      }
    }
  }
  break;
```

79d ⟨*Check known DAV: report types* 77d⟩+≡ (76d) ◁79c 80a▷

```
case XML_NAME_PRINCIPAL_MATCH:
  valid = (node = xml_child(node)) && (!node->next || !node->next->next) &&
          node->type == XML_ELEMENT_NODE && check_ns(node, DAV_NS, &dav_ns) &&
          (!node->next || (node->next->type == XML_ELEMENT_NODE &&
                           check_ns(node->next, DAV_NS, &dav_ns)));
  if(valid) {
    xml_name_t tn = xml_tag(node);
    if(tn == XML_NAME_PROP) {
      valid = check_prop(node, FALSE) && node->next;
      if(valid) {
        node = node->next;
        tn = xml_tag(node);
```

79d ⟨*Check known DAV: report types* 77d⟩+≡ (76d) ◁79c 80a▷

```
      }
    }
    if(tn == XML_NAME_SELF)
      valid = !xml_child(node);
    else if(tn == XML_NAME_PRINCIPAL_PROPERTY)
      valid = xml_child(node) && !node->children->next &&
              check_prop(node, FALSE);
    else
      valid = FALSE;
    if(valid && node->next)
      valid = xml_tag(node->next) == XML_NAME_PROP &&
              check_prop(node->next, FALSE);
  }
  break;
```

80a ⟨*Check known DAV: report types* 77d⟩+≡ (76d) ◁79d 80b▷

```
case XML_NAME_PRINCIPAL_PROPERTY_SEARCH:
  if(!xml_child(node))
    valid = FALSE;
  else {
    gboolean got_ps = FALSE, got_prop = FALSE, got_atpcs = FALSE;
    for(node = node->children; node && valid; node = node->next) {
      xml_name_t tn = xml_tag(node);
      if(tn == XML_NAME_PROPERTY_SEARCH) {
        got_ps = TRUE;
        valid = xml_child(node) && node->children->next && !node->children->next->next &&
                node->children->type == XML_ELEMENT_NODE &&
                node->children->next->type == XML_ELEMENT_NODE &&
                check_ns(node->children, DAV_NS, &dav_ns) &&
                check_ns(node->children->next, DAV_NS, &dav_ns);
        if(valid) {
          xmlNodePtr c = node->children;
          tn = xml_tag(c);
          if(tn == XML_NAME_PROP) {
            valid = check_prop(c, FALSE);
            c = c->next;
            tn = xml_tag(c);
          } else
            valid = xml_tag(c->next) == XML_NAME_PROP &&
                    check_prop(c->next, FALSE);
          if(valid)
            valid = tn == XML_NAME_MATCH && xml_child(c) &&
                    xmlNodeIsText(c->children) &&
                    !c->children->next;
        }
      } else if(tn == XML_NAME_PROP) {
        valid = !got_prop && check_prop(node, FALSE);
        got_prop = TRUE;
      } else if(tn == XML_NAME_APPLY_TO_PRINCIPAL_COLLECTION_SET) {
        valid = !got_atpcs && !xml_child(node);
        got_atpcs = TRUE;
      }
    }
    valid = valid && got_ps;
  }
  break;
```

80b ⟨*Check known DAV: report types* 77d⟩+≡ (76d) ◁80a

```
case XML_NAME_PRINCIPAL_SEARCH_PROPERTY_SET:
  valid = !xml_child(node);
  break;
```

Finally, RFC 4791 defines a few calendaring report types in the urn:ietf:params:xml:ns:caldav namespace. Note that unlike the other methods, calendaring reports specify properties using children of calendar-data rather than using properties directly under prop.

81a ⟨*webdav.dtd* 65c⟩+≡ ◁79a 87b▷

```
<!ELEMENT calendar-query ((DAV:allprop |
                           DAV:propname |
                           DAV:prop)?, filter, timezone?)>
  <!ELEMENT filter (comp-filter)>
    <!ELEMENT comp-filter (is-not-defined | (time-range?,
                          prop-filter*, comp-filter*))>
      <!ATTLIST comp-filter name CDATA #REQUIRED>
      <!ELEMENT is-not-defined EMPTY>
      <!ELEMENT time-range EMPTY>
        <!ATTLIST time-range start CDATA #IMPLIED
                          end   CDATA #IMPLIED>
      <!ELEMENT prop-filter (is-not-defined |
                          ((time-range | text-match)?,
                           param-filter*))>
        <!ATTLIST prop-filter name CDATA #REQUIRED>
        <!ELEMENT text-match (#PCDATA)>
          <!ATTLIST text-match collation        CDATA "i;ascii-casemap"
                           negate-condition (yes | no) "no">
        <!ELEMENT param-filter (is-not-defined | text-match?)>
          <!ATTLIST param-filter name CDATA #REQUIRED>
  <!ELEMENT timezone (#PCDATA)>
<!ELEMENT calendar-multiget ((DAV:allprop |
                              DAV:propname |
                              DAV:prop)?, DAV:href+)>
<!ELEMENT free-busy-query (time-range)>

<!ELEMENT calendar-data (comp?, (expand | limit-recurrence-set)?,
                         limit-freebusy-set?)>
  <!ATTLIST calendar-data content-type CDATA "text/calendar"
                          version CDATA "2.0">
  <!-- actual samples seem to indicate there should ?s after all{prop,comp} -->
  <!ELEMENT comp ((allprop | prop*), (allcomp | comp*))>
    <!ATTLIST comp name CDATA #REQUIRED>
    <!ELEMENT allcomp EMPTY>
    <!ELEMENT allprop EMPTY>
    <!ELEMENT prop EMPTY>
      <!ATTLIST prop name CDATA #REQUIRED
                  novalue (yes | no) "no">
  <!ELEMENT expand EMPTY>
    <!ATTLIST expand start CDATA #REQUIRED
                     end   CDATA #REQUIRED>
  <!ELEMENT limit-recurrence-set EMPTY>
    <!ATTLIST limit-recurrence-set start CDATA #REQUIRED
                                   end   CDATA #REQUIRED>
  <!ELEMENT limit-freebusy-set EMPTY>
    <!ATTLIST limit-freebusy-set start CDATA #REQUIRED
                                 end   CDATA #REQUIRED>
```

81b ⟨*XML validator names* 64d⟩+≡ (63b) ◁79b 87c▷

```
calendar-query calendar-multiget free-busy-query
filter comp-filter is-not-defined time-range prop-filter start end text-match
collation negate-condition timezone param-filter
calendar-data comp expand limit-recurrence-set limit-freebusy-set
content-type version allcomp novalue
```

81c ⟨*Check known CALDAV: report types* 81c⟩≡ (76d) 86b▷

```
case XML_NAME_CALENDAR_QUERY: {
  gboolean got_prop = FALSE, got_filt = FALSE, got_tz = FALSE;
  xml_name_t tn;
  for(node = xml_child(node); node && valid; node = node->next) {
    if(node->type != XML_ELEMENT_NODE) {
      valid = FALSE;
      break;
    }
    tn = xml_tag(node);
    if(check_ns(node, DAV_NS, &dav_ns)) {
      valid = (tn == XML_NAME_ALLPROP || tn == XML_NAME_PROP ||
               tn == XML_NAME_PROPNAME) && !got_prop;
      if(valid) {
        got_prop = TRUE;
        if(tn == XML_NAME_PROP)
          valid = check_caldav_prop(node, &caldav_ns);
        else
          valid = !xml_child(node);
      }
    } else if(check_ns(node, CALDAV_NS, &caldav_ns)) {
      if(tn == XML_NAME_FILTER) {
        valid = !got_filt && xml_child(node) &&
                node->children->type == XML_ELEMENT_NODE &&
                !node->children->next &&
                check_ns(node->children, CALDAV_NS, &caldav_ns) &&
                xml_tag(node->children) == XML_NAME_COMP_FILTER;
        got_filt = TRUE;
        if(valid)
          valid = check_comp_filter(node->children, &caldav_ns);
      } else if(tn == XML_NAME_TIMEZONE) {
        /* no attempt to validate tz format */
        valid = xml_child(node) && xmlNodeIsText(node->children) &&
                !node->children->next && !got_tz;
        got_tz = TRUE;
      } else
        valid = FALSE;
    } else
      valid = FALSE;
  }
  valid = valid && got_filt;
  break;
}
```

82 ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁78c 83▷

```
static gboolean check_caldav_prop(xmlNodePtr node, xmlNsPtr *nsc)
{
  if(!xml_child(node))
    return FALSE;
  for(node = node->children; node; node = node->next) {
    if(node->type != XML_ELEMENT_NODE || !node->ns || !node->ns->href)
      return FALSE;
    if(xml_child(node)) {
      if(!check_ns(node, CALDAV_NS, nsc) ||
         xml_tag(node) != XML_NAME_CALENDAR_DATA)
        return FALSE;
      xmlAttrPtr attrs;
      xml_name_t tn;
      for(attrs = node->properties; attrs; attrs = attrs->next)
        if(check_ns((xmlNodePtr)attrs, CALDAV_NS, nsc) &&
           ((tn = xml_tag((xmlNodePtr)attrs)) == XML_NAME_CONTENT_TYPE ||
            tn == XML_NAME_VERSION) &&
           attrs->children && (attrs->children->next ||
                               !xmlNodeIsText(attrs->children)))
```

82 ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁78c 83▷

```
            return FALSE;
        gboolean got_comp = FALSE, got_explrs = FALSE, got_lfs = FALSE;
        xmlNodePtr c;
        for(c = xml_child(node); c; c = c->next) {
          if(c->type != XML_ELEMENT_NODE || !check_ns(c, CALDAV_NS, nsc))
            return FALSE;
          tn = xml_tag(c);
          if(tn == XML_NAME_COMP) {
            if(got_comp || !check_caldav_comp(c, nsc))
              return FALSE;
            got_comp = TRUE;
          } else if(tn == XML_NAME_EXPAND || tn == XML_NAME_LIMIT_RECURRENCE_SET) {
            if(got_explrs || xml_child(c) ||
                !check_timerange_attr(c, FALSE, nsc))
              return FALSE;
            got_explrs = TRUE;
          } else if(tn == XML_NAME_LIMIT_FREEBUSY_SET) {
            if(got_lfs || xml_child(c) ||
                !check_timerange_attr(c, FALSE, nsc))
              return FALSE;
            got_lfs = TRUE;
          } else
            return FALSE;
        }
      }
    }
    return TRUE;
}
```

83 ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁82 84▷

```
static gboolean check_comp_filter(xmlNodePtr node, xmlNsPtr *nsc)
{
  xml_name_t tn;
  if(!check_name_attr(node, nsc))
    return FALSE;
  /* no explicit text to guarantee children, so x | (? * *)    means optional */
  if(!xml_child(node))
    return TRUE;
  gboolean got_ind = FALSE, got_tr = FALSE;
  for(node = node->children; node; node = node->next) {
    if(node->type != XML_ELEMENT_NODE || !check_ns(node, CALDAV_NS, nsc))
      return FALSE;
    if((tn = xml_tag(node)) == XML_NAME_IS_NOT_DEFINED && got_ind)
      return FALSE;
    got_ind = TRUE; /* prevent ind later even if tag wasn't ind */
    if(tn == XML_NAME_IS_NOT_DEFINED) {
      if(xml_child(node))
        return FALSE;
    } else if(tn == XML_NAME_TIME_RANGE) {
      if(!check_timerange_attr(node, TRUE, nsc) || xml_child(node) || got_tr)
        return FALSE;
      got_tr = TRUE;
    } else if(tn == XML_NAME_COMP_FILTER) {
      if(!check_comp_filter(node, nsc))
        return FALSE;
    } else if(tn == XML_NAME_PROP_FILTER) {
      xmlNodePtr c;
      gboolean got_trm = FALSE, got_ind2 = FALSE;
      if(!check_name_attr(node, nsc))
        return FALSE;
      for(c = xml_child(node); c; c = c->next) {
        if(c->type != XML_ELEMENT_NODE || !check_ns(c, CALDAV_NS, nsc))
          return FALSE;
```

83 ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁82 84▷

```
        if((tn = xml_tag(c)) == XML_NAME_IS_NOT_DEFINED && got_ind2)
          return FALSE;
        got_ind2 = TRUE;
        if(tn == XML_NAME_IS_NOT_DEFINED) {
          if(xml_child(c))
            return FALSE;
        } else if(tn == XML_NAME_TIME_RANGE) {
          if(!check_timerange_attr(c, TRUE, nsc) || xml_child(c) || got_trm)
            return FALSE;
          got_trm = TRUE;
        } else if(tn == XML_NAME_TEXT_MATCH) {
          if(got_trm || !check_text_match(c, nsc))
            return FALSE;
          got_trm = TRUE;
        } else if(tn == XML_NAME_PARAM_FILTER) {
          if(!check_name_attr(c, nsc))
            return FALSE;
          if(xml_child(c)) {
            if(c->children->next || c->type != XML_ELEMENT_NODE ||
               !check_ns(c, CALDAV_NS, nsc))
              return FALSE;
            if((tn = xml_tag(c->children)) == XML_NAME_IS_NOT_DEFINED) {
              if(xml_child(c->children))
                return FALSE;
            } else if(tn == XML_NAME_TEXT_MATCH) {
              if(!check_text_match(c->children, nsc))
                return FALSE;
            } else
              return FALSE;
          }
        } else
          return FALSE;
      }
    } else
      return FALSE;
  }
  return TRUE;
}
```

84 ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁83 85a▷

```
static gboolean check_timerange_attr(xmlNodePtr node, gboolean open_range,
                                     xmlNsPtr *nsc)
{
  xmlAttrPtr attrs;
  gboolean got_start = FALSE, got_end = FALSE;
  xml_name_t tn;

  for(attrs = node->properties; attrs; attrs = attrs->next)
    if(check_ns((xmlNodePtr)attrs, CALDAV_NS, nsc)) {
      if((tn = xml_tag((xmlNodePtr)attrs)) == XML_NAME_START)
        got_start = TRUE;
      else if(tn == XML_NAME_END)
        got_end = TRUE;
      else
        continue;
      /* no attempt to verify date is valid */
      if(!attrs->children || attrs->children->next ||
         !xmlNodeIsText(attrs->children))
        return FALSE;
    }
  return open_range ? got_start || got_end : got_start && got_end;
}
```

85a ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁84 85b▷

```
static gboolean check_name_attr(xmlNodePtr node, xmlNsPtr *nsc)
{
  xmlAttrPtr attrs;

  for(attrs = node->properties; attrs; attrs = attrs->next)
    if(xml_tag((xmlNodePtr)attrs) == XML_NAME_NAME &&
       check_ns((xmlNodePtr)attrs, CALDAV_NS, nsc))
      /* no attempt to verify name is valid */
      return attrs->children && !attrs->children->next &&
             xmlNodeIsText(attrs->children);
  return FALSE;
}
```

85b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁85a 86a▷

```
static gboolean check_caldav_comp(xmlNodePtr node, xmlNsPtr *nsc)
{
  /* actual samples indicate children not really necessary */
#if 0
  if(!xml_child(node) || !check_name_attr(node, nsc))
#else
  if(!check_name_attr(node, nsc))
#endif
    return FALSE;
  gboolean got_prop = FALSE, got_allprop = FALSE, got_comp = FALSE,
           got_allcomp = FALSE;
  xml_name_t tn;
  for(node = xml_child(node); node; node = node->next) {
    if(node->type != XML_ELEMENT_NODE || !check_ns(node, CALDAV_NS, nsc))
      return FALSE;
    tn = xml_tag(node);
    if(tn == XML_NAME_ALLCOMP) {
      if(got_comp || got_allcomp || xml_child(node))
        return FALSE;
      got_allcomp = TRUE;
    } else if(tn == XML_NAME_ALLPROP) {
      if(got_prop || got_allprop || xml_child(node))
        return FALSE;
      got_allprop = TRUE;
    } else if(tn == XML_NAME_COMP) {
      if(got_allcomp || !check_caldav_comp(node, nsc))
        return FALSE;
      got_comp = TRUE;
    } else if(tn == XML_NAME_PROP) {
      if(got_allprop)
        return FALSE;
      got_prop = TRUE;
      xmlAttrPtr attrs;
      gboolean got_name = FALSE;
      for(attrs = node->properties; attrs; attrs = attrs->next) {
        if(!check_ns((xmlNodePtr)attrs, CALDAV_NS, nsc))
          continue;
        if((tn = xml_tag((xmlNodePtr)attrs)) == XML_NAME_NAME)
          got_name = attrs->children && !attrs->children->next &&
                     xmlNodeIsText(attrs->children);
        else if(tn == XML_NAME_NOVALUE) {
          if((tn = xml_str_id(attrs->children)) != XML_NAME_YES &&
             tn != XML_NAME_NO)
            return FALSE;
        } else
          return FALSE;
      }
```

85b ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁85a 86a▷

```
      if(!got_name)
        return FALSE;
    } else
      return FALSE;
  }
#if 0
  /* actual samples seem to indicate allprop/allcomp is implicit */
  return (got_prop || got_allprop) && (got_comp || got_allcomp);
#else
  return TRUE;
#endif
}
```

86a ⟨*CGI Support Functions* 9b⟩+≡ (8e) ◁85b

```
static gboolean check_text_match(xmlNodePtr node, xmlNsPtr *nsc)
{
  xmlAttrPtr attrs;
  xml_name_t tn;

  if(!xml_child(node) || node->children->next || !xmlNodeIsText(node->children))
    return FALSE;
  for(attrs = node->properties; attrs; attrs = attrs->next) {
    if(!check_ns((xmlNodePtr)attrs, CALDAV_NS, nsc))
      continue;
    if((tn = xml_tag((xmlNodePtr)attrs)) == XML_NAME_COLLATION) {
      if(!attrs->children || attrs->children->next ||
         !xmlNodeIsText(attrs->children))
        return FALSE;
     } else if(tn == XML_NAME_NEGATE_CONDITION) {
      if(!attrs->children || attrs->children->next ||
         !xmlNodeIsText(attrs->children))
        return FALSE;
      if((tn = xml_str_id(attrs->children->content)) != XML_NAME_YES &&
         tn != XML_NAME_NO)
        return FALSE;
    }
  }
  return TRUE;
}
```

86b ⟨*Check known CALDAV: report types* 81c⟩+≡ (76d) ◁81c 87a▷

```
case XML_NAME_CALENDAR_MULTIGET: {
  gboolean got_prop = FALSE, got_hr = FALSE;
  for(node = xml_child(node); node; node = node->next) {
    if(node->type != XML_ELEMENT_NODE || !check_ns(node, DAV_NS, &dav_ns)) {
      valid = FALSE;
      break;
    }
    xml_name_t tn = xml_tag(node);
    if(tn == XML_NAME_ALLPROP || tn == XML_NAME_PROP)
      valid = !xml_child(node) && !got_prop;
    else if(tn == XML_NAME_PROP)
      valid = !got_prop && check_prop(node, FALSE);
    else if(tn == XML_NAME_HREF)
      valid  = xml_child(node) && !node->children->next &&
               xmlNodeIsText(node->children);
    else
      valid = FALSE;
    if(tn == XML_NAME_HREF)
      got_hr = TRUE;
    else
```

86b ⟨*Check known CALDAV: report types* 81c⟩+≡ (76d) ◁81c 87a▷

```
      got_prop = TRUE;
  }
  valid = valid && got_hr;
  break;
}
```

87a ⟨*Check known CALDAV: report types* 81c⟩+≡ (76d) ◁86b

```
case XML_NAME_FREE_BUSY_QUERY:
  valid = (node = xml_child(node)) && !node->next &&
          node->type == XML_ELEMENT_NODE &&
          check_ns(node, CALDAV_NS, &caldav_ns) &&
          xml_tag(node) == XML_NAME_TIME_RANGE &&
          check_timerange_attr(node, TRUE, &caldav_ns);
  break;
```

The VERSION-CONTROL method takes an optional body with a version-control element.

87b ⟨*webdav.dtd* 65c⟩+≡ ◁81a 87f▷

```
<!ELEMENT version-control ANY>
```

87c ⟨*XML validator names* 64d⟩+≡ (63b) ◁81b 87g▷

```
version-control
```

87d ⟨*unless method requires no body or XML body* 66b⟩+≡ (63a) ◁75c 87h▷

```
&& req_method != HTTP_REQ_VERSION_CONTROL
```

87e ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁76d 87i▷

```
case HTTP_REQ_VERSION_CONTROL:
  valid = !node->next && node->type == XML_ELEMENT_NODE &&
          xml_tag(node) == XML_NAME_VERSION_CONTROL &&
          check_ns(node, DAV_NS, &dav_ns);
  break;
```

The CHECKOUT method takes an optional body with a checkout element.

87f ⟨*webdav.dtd* 65c⟩+≡ ◁87b 88a▷

```
<!ELEMENT checkout ANY>
  <!-- children may have at most one fork-ok, but otherwise elements -->
  <!ELEMENT fork-ok EMPTY>
  <!-- or at most one apply-to-version -->
  <!ELEMENT apply-to-version EMPTY>
```

87g ⟨*XML validator names* 64d⟩+≡ (63b) ◁87c 88b▷

```
checkout fork-ok apply-to-version
```

87h ⟨*unless method requires no body or XML body* 66b⟩+≡ (63a) ◁87d 88c▷

```
&& req_method != HTTP_REQ_CHECKOUT
```

87i ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁87e 88d▷

```
case HTTP_REQ_CHECKOUT:
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_CHECKOUT || !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  else {
    gboolean got_fok = FALSE, got_atv = FALSE;
    for(node = xml_child(node); node && valid; node = node->next) {
      if(node->type != XML_ELEMENT_NODE)
        valid = FALSE;
      else if(check_ns(node, DAV_NS, &dav_ns)) {
        xml_name_t tn = xml_tag(node);
        if(tn == XML_NAME_FORK_OK) {
          valid = !got_fok && !xml_child(node);
          got_fok = TRUE;
        } else if(tn == XML_NAME_APPLY_TO_VERSION) {
          valid = !got_atv && !xml_child(node);
          got_atv = TRUE;
        }
      }
    }
  }
  break;
```

The CHECKIN method takes an optional body with a checkin element.

88a ⟨*webdav.dtd* 65c⟩+≡ ◁87f 89a▷

```
<!ELEMENT checkin ANY>
  <!-- children may have at most one fork-ok and keep-checked-out + other el -->
  <!ELEMENT keep-checked-out EMPTY>
  <!ELEMENT fork-ok EMPTY>
```

88b ⟨*XML validator names* 64d⟩+≡ (63b) ◁87g 89b▷

```
checkin keep-checked-out
```

88c ⟨*unless method requires no body or XML body* 66b⟩+≡ (63a) ◁87h 89c▷

```
&& req_method != HTTP_REQ_CHECKIN
```

88d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁87i 89d▷

```
case HTTP_REQ_CHECKIN:
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_CHECKOUT || !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  else {
    gboolean got_fok = FALSE, got_kco = FALSE;
    for(node = xml_child(node); node && valid; node = node->next) {
      if(node->type != XML_ELEMENT_NODE)
        valid = FALSE;
      else if(check_ns(node, DAV_NS, &dav_ns)) {
        xml_name_t tn = xml_tag(node);
        if(tn == XML_NAME_FORK_OK) {
          valid = !got_fok && !xml_child(node);
          got_fok = TRUE;
        } else if(tn == XML_NAME_KEEP_CHECKED_OUT) {
          valid = !got_kco && !xml_child(node);
          got_kco = TRUE;
        }
      }
```

88d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁87i 89d▷

```
    }
  }
  break;
```

The UNCHECKOUT method takes an optional body with an uncheckout element.

89a ⟨*webdav.dtd* 65c⟩+≡ ◁88a 89e▷

```
<!ELEMENT uncheckout ANY>
```

89b ⟨*XML validator names* 64d⟩+≡ (63b) ◁88b 89f▷

```
uncheckout
```

89c ⟨*unless method requires no body or XML body* 66b⟩+≡ (63a) ◁88c 89g▷

```
&& req_method != HTTP_REQ_UNCHECKOUT
```

89d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁88d 89h▷

```
case HTTP_REQ_UNCHECKOUT:
  valid = !node->next && node->type == XML_ELEMENT_NODE &&
          xml_tag(node) == XML_NAME_UNCHECKOUT &&
          check_ns(node, DAV_NS, &dav_ns);
  break;
```

The MKWORKSPACE method takes an optional body with a mkworkspace element.

89e ⟨*webdav.dtd* 65c⟩+≡ ◁89a 89i▷

```
<!ELEMENT mkworkspace ANY>
```

89f ⟨*XML validator names* 64d⟩+≡ (63b) ◁89b 89j▷

```
mkworkspace
```

89g ⟨*unless method requires no body or XML body* 66b⟩+≡ (63a) ◁89c 92c▷

```
&& req_method != HTTP_REQ_MKWORKSPACE
```

89h ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁89d 90b▷

```
case HTTP_REQ_MKWORKSPACE:
  valid = !node->next && node->type == XML_ELEMENT_NODE &&
          xml_tag(node) == XML_NAME_MKWORKSPACE &&
          check_ns(node, DAV_NS, &dav_ns);
  break;
```

The UPDATE method requires an XML body with a an update element.

89i ⟨*webdav.dtd* 65c⟩+≡ ◁89e 90c▷

```
<!ELEMENT update ANY>
  <!-- ANY: elements with at most one version & one prop -->
  <!ELEMENT version (href)>
```

89j ⟨*XML validator names* 64d⟩+≡ (63b) ◁89f 90d▷

```
update
```

90a ⟨*unless method requires XML body* 67b⟩+≡ (63a) ◁76b 90e▷

```
&& req_method != HTTP_REQ_UPDATE
```

90b ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁89h 90f▷

```
case HTTP_REQ_UPDATE:
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_UPDATE || !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  else {
    gboolean got_vers = FALSE, got_prop = FALSE;
    for(node = xml_child(node); node && valid; node = node->next) {
      if(node->type != XML_ELEMENT_NODE)
        valid = FALSE;
      else if(check_ns(node, DAV_NS, &dav_ns)) {
        xml_name_t tn = xml_tag(node);
        if(tn == XML_NAME_VERSION) {
          valid = !got_vers && xml_child(node) && !node->children->next &&
                  node->children->type == XML_ELEMENT_NODE &&
                  xml_tag(node->children) == XML_NAME_HREF &&
                  check_ns(node->children, DAV_NS, &dav_ns) &&
                  xml_child(node->children) && !node->children->children->next &&
                  xmlNodeIsText(node->children->children);
          got_vers = TRUE;
        } else if(tn == XML_NAME_PROP) {
          /* no help in RFC on what format is, so be inclusive */
          valid = !got_prop && check_prop(node, TRUE);
          got_prop = TRUE;
        }
      }
    }
  }
  break;
```

The LABEL method requires an XML body with a label element.

90c ⟨*webdav.dtd* 65c⟩+≡ ◁89i 91a▷

```
<!ELEMENT label ANY>
  <!-- any elements, but at most one add, set, or remove -->
  <!ELEMENT add (label-name)>
    <!ELEMENT label-name (#PCDATA)>
  <!ELEMENT set (label-name)>
  <!ELEMENT remove (label-name)>
```

90d ⟨*XML validator names* 64d⟩+≡ (63b) ◁89j 91b▷

```
label add label-name
```

90e ⟨*unless method requires XML body* 67b⟩+≡ (63a) ◁90a 91c▷

```
&& req_method != HTTP_REQ_LABEL
```

90f ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁90b 91d▷

```
case HTTP_REQ_LABEL:
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_LABEL || !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  else {
    gboolean got_one = FALSE;
    for(node = xml_child(node); node && valid; node = node->next) {
      if(node->type != XML_ELEMENT_NODE)
        valid = FALSE;
      else if(check_ns(node, DAV_NS, &dav_ns)) {
        xml_name_t tn = xml_tag(node);
        if(tn == XML_NAME_SET || tn == XML_NAME_ADD || tn == XML_NAME_REMOVE) {
          valid = !got_one && xml_child(node) && !node->children->next &&
                  node->children->type == XML_ELEMENT_NODE &&
                  xml_tag(node->children) == XML_NAME_LABEL_NAME &&
                  check_ns(node->children, DAV_NS, &dav_ns) &&
                  xml_child(node->children) && !node->children->children->next &&
                  xmlNodeIsText(node->children->children);
          got_one = TRUE;
        }
      }
    }
  }
  break;
```

The MERGE method requires an XML body with a merge element.

91a ⟨*webdav.dtd* 65c⟩+≡ ◁90c 92a▷

```
<!ELEMENT merge ANY>
  <!-- ANY is elements with one source, at most one no-auto-merge, no-checkout,
       prop, update parms (fork-ok, apply-to-version) -->
  <!ELEMENT source (href+)>
  <!ELEMENT no-auto-merge EMPTY>
  <!ELEMENT no-checkout EMPTY>
  <!ELEMENT fork-ok EMPTY>
  <!ELEMENT apply-to-version EMPTY>
```

91b ⟨*XML validator names* 64d⟩+≡ (63b) ◁90d 92b▷

```
merge source no-auto-merge no-checkout
```

91c ⟨*unless method requires XML body* 67b⟩+≡ (63a) ◁90e 93g▷

```
&& req_method != HTTP_REQ_MERGE
```

91d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁90f 92d▷

```
case HTTP_REQ_MERGE:
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_MERGE || !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  else {
    gboolean got_src = FALSE, got_noam = FALSE, got_nc = FALSE,
             got_fok = FALSE, got_atv = FALSE;
    for(node = xml_child(node); node; node = node->next) {
      if(node->type != XML_ELEMENT_NODE)
        valid = FALSE;
      else if(check_ns(node, DAV_NS, &dav_ns)) {
        xml_name_t tn = xml_tag(node);
        if(tn == XML_NAME_NO_AUTO_MERGE) {
```

91d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁90f 92d▷

```
          valid = !xml_child(node) && !got_noam;
          got_noam = TRUE;
        } else if (tn == XML_NAME_NO_CHECKOUT) {
          valid = !xml_child(node) && !got_nc;
          got_nc = TRUE;
        } else if (tn == XML_NAME_FORK_OK) {
          valid = !xml_child(node) && !got_fok;
          got_fok = TRUE;
        } else if (tn == XML_NAME_APPLY_TO_VERSION) {
          valid = !xml_child(node) && !got_atv;
          got_atv = TRUE;
        } else if (tn == XML_NAME_SOURCE) {
          valid = !got_src && xml_child(node);
          got_src = TRUE;
          xmlNodePtr c;
          for (c = xml_child(node); c && valid; c = c->next)
            valid = c->type == XML_ELEMENT_NODE &&
                    check_ns(c, DAV_NS, &dav_ns) &&
                    xml_tag(c) == XML_NAME_HREF &&
                    xml_child(c) && !c->children->next &&
                    xmlNodeIsText(c->children);
        }
      }
    }
    valid = valid && got_src;
  }
  break;
```

The BASELINE-CONTROL method takes an optional body with a baseline-control element.

92a ⟨*webdav.dtd* 65c⟩+≡ ◁91a 93a▷

```
<!ELEMENT baseline-control ANY>
  <!-- ANY is elements w/ at most one baseline -->
  <!ELEMENT baseline (href)>
```

92b ⟨*XML validator names* 64d⟩+≡ (63b) ◁91b 93b▷

```
baseline-control baseline
```

92c ⟨*unless method requires no body or XML body* 66b⟩+≡ (63a) ◁89g 93c▷

```
&& req_method != HTTP_REQ_BASELINE_CONTROL
```

92d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁91d 93d▷

```
case HTTP_REQ_BASELINE_CONTROL:
  if (node->next || node->type != XML_ELEMENT_NODE ||
      xml_tag(node) != XML_NAME_BASELINE_CONTROL ||
      !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  else {
    gboolean got_bl = FALSE;
    for (node = xml_child(node); node && valid; node = node->next) {
      if (node->type != XML_ELEMENT_NODE)
        valid = FALSE;
      else if (check_ns(node, DAV_NS, &dav_ns)) {
        xml_name_t tn = xml_tag(node);
        if (tn == XML_NAME_BASELINE) {
          valid = !got_bl && xml_child(node) && !node->children->next &&
                  node->children->type == XML_ELEMENT_NODE &&
                  xml_tag(node->children) == XML_NAME_HREF &&
```

92d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁91d 93d▷

```
                    check_ns(node->children, DAV_NS, &dav_ns) &&
                    xml_child(node->children) && !node->children->children->next &&
                    xmlNodeIsText(node->children->children);
            got_bl = TRUE;
          }
        }
      }
    }
    break;
```

The MKACTIVITY method takes an optional body with a mkactivity element.

93a ⟨*webdav.dtd* 65c⟩+≡ ◁92a 93e▷

```
<!ELEMENT mkactivity ANY>
```

93b ⟨*XML validator names* 64d⟩+≡ (63b) ◁92b 93f▷

```
mkactivity
```

93c ⟨*unless method requires no body or XML body* 66b⟩+≡ (63a) ◁92c 96e▷

```
&& req_method != HTTP_REQ_MKACTIVITY
```

93d ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁92d 93h▷

```
case HTTP_REQ_MKACTIVITY:
  valid = !node->next && node->type == XML_ELEMENT_NODE &&
          xml_tag(node) == XML_NAME_MKACTIVITY &&
          check_ns(node, DAV_NS, &dav_ns);
  break;
```

The ORDERPATCH requires an XML body with an orderpatch element.

93e ⟨*webdav.dtd* 65c⟩+≡ ◁93a 94▷

```
<!ELEMENT orderpatch (ordering-type?, order-member*) >
  <!ELEMENT ordering-type (href) >
  <!ELEMENT order-member (segment, position) >
    <!ELEMENT segment (#PCDATA)>
    <!ELEMENT position (first | last | before | after)>
      <!ELEMENT first EMPTY >
      <!ELEMENT last EMPTY >
      <!ELEMENT before segment >
      <!ELEMENT after segment >
```

93f ⟨*XML validator names* 64d⟩+≡ (63b) ◁93b 95a▷

```
orderpatch ordering-type order-member segment position first last before after
```

93g ⟨*unless method requires XML body* 67b⟩+≡ (63a) ◁91c 95b▷

```
&& req_method != HTTP_REQ_ORDERPATCH
```

93h ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁93d 95c▷

```
case HTTP_REQ_ORDERPATCH:
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_ORDERPATCH || !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  else {
    gboolean got_ot = FALSE;
    xml_name_t tn;
    xmlNodePtr c;
    for(node = xml_child(node); node && valid; node = node->next) {
      if(node->type != XML_ELEMENT_NODE || !(c = xml_child(node)) ||
         c->type != XML_ELEMENT_NODE || !check_ns(node, DAV_NS, &dav_ns) ||
         !check_ns(c, DAV_NS, &dav_ns))
        valid = FALSE;
      else if((tn = xml_tag(node)) == XML_NAME_ORDERING_TYPE) {
        valid = !got_ot && !c->next && xml_tag(c) == XML_NAME_HREF &&
                xml_child(c) && !c->children->next &&
                xmlNodeIsText(c->children);
        got_ot = TRUE;
      } else if(tn == XML_NAME_ORDER_MEMBER) {
        valid = c->next && !c->next->next &&
                c->next->type == XML_ELEMENT_NODE &&
                check_ns(c->next, DAV_NS, &dav_ns) &&
                xml_child(c) && !c->children->next &&
                xml_child(c->next) && !c->next->children->next;
        if(valid) {
          tn = xml_tag(c);
          if(tn == XML_NAME_SEGMENT) {
            valid = xmlNodeIsText(c->children);
            c = c->next;
            tn = xml_tag(c);
          }
          if(valid && tn == XML_NAME_POSITION) {
            valid = c->children->type == XML_ELEMENT_NODE &&
                    check_ns(c->children, DAV_NS, &dav_ns);
            if(valid) {
              tn = xml_tag(c->children);
              if(tn == XML_NAME_FIRST || tn == XML_NAME_LAST)
                valid = !c->children->children;
              else if(tn == XML_NAME_BEFORE || tn == XML_NAME_AFTER) {
                xmlNodePtr subc = xml_child(c->children);
                valid = subc && !subc->next &&
                        subc->type == XML_ELEMENT_NODE &&
                        check_ns(subc, DAV_NS, &dav_ns) &&
                        xml_tag(subc) == XML_NAME_SEGMENT &&
                        xml_child(subc) && !subc->children->next &&
                        xmlNodeIsText(subc->children);
              } else
                valid = FALSE;
            }
          } else
            valid = FALSE;
          if(valid && (c = c->next))
            valid = xml_tag(c) == XML_NAME_SEGMENT &&
                    xmlNodeIsText(xml_child(c));
        }
      } else
        valid = FALSE;
    }
  }
  break;
```

The MKREDIRECTREF method requires an XML body with a mkredirectref element.

94 ⟨*webdav.dtd* 65c⟩+≡ ◁93e 95e▷

```
<!ELEMENT mkredirectref (reftarget, redirect-lifetime?)>
  <!ELEMENT reftarget (href)>
  <!ELEMENT redirect-lifetime (permanent | temporary)>
    <!ELEMENT permanent EMPTY>
    <!ELEMENT temporary EMPTY>
```

95a ⟨*XML validator names* 64d⟩+≡ (63b) ◁93f 95f▷

```
mkredirectref reftarget redirect-lifetime permanent temporary
```

95b ⟨*unless method requires XML body* 67b⟩+≡ (63a) ◁93g 96a▷

```
&& req_method != HTTP_REQ_MKREDIRECTREF
```

95c ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁93h 96b▷

```
case HTTP_REQ_MKREDIRECTREF:
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_MKREDIRECTREF ||
     !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  else {
    ⟨Check redirect reference creation XML 95d⟩
    valid = valid && got_rt;
  }
  break;
```

95d ⟨*Check redirect reference creation XML* 95d⟩≡ (95c 96b)

```
gboolean got_rt = FALSE, got_rl = FALSE;
for(node = xml_child(node); node && valid; node = node->next) {
  xml_name_t tn;
  xmlNodePtr c;
  if(node->type != XML_ELEMENT_NODE || !(c = xml_child(node)) ||
     c->type != XML_ELEMENT_NODE || c->next ||
     !check_ns(node, DAV_NS, &dav_ns) || !check_ns(c, DAV_NS, &dav_ns))
    valid = FALSE;
  else if((tn = xml_tag(node)) == XML_NAME_REFTARGET) {
    valid = !got_rt && xml_tag(c) == XML_NAME_HREF &&
            xml_child(c) && xmlNodeIsText(c->children) && !c->children->next;
    got_rt = TRUE;
  } else if((tn = xml_tag(node)) == XML_NAME_REDIRECT_LIFETIME) {
    valid = !got_rl && !xml_child(c) &&
            ((tn = xml_tag(node)) == XML_NAME_PERMANENT ||
             tn == XML_NAME_TEMPORARY);
    got_rl = TRUE;
  } else
    valid = FALSE;
}
```

The UPDATEREDIRECTREF method requires an XML body with an updateredirectref element.

95e ⟨*webdav.dtd* 65c⟩+≡ ◁94 96c▷

```
<!ELEMENT updateredirectref (reftarget?, redirect-lifetime?)>
```

95f ⟨*XML validator names* 64d⟩+≡ (63b) ◁95a 96d▷

```
updateredirectref
```

96a ⟨*unless method requires XML body* 67b⟩+≡ (63a) ◁95b

```
&& req_method != HTTP_REQ_UPDATEREDIRECTREF
```

96b ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁95c 96f▷

```
case HTTP_REQ_UPDATEREDIRECTREF:
  if(node->next || node->type != XML_ELEMENT_NODE ||
     xml_tag(node) != XML_NAME_UPDATEREDIRECTREF ||
     !check_ns(node, DAV_NS, &dav_ns))
    valid = FALSE;
  else {
    ⟨Check redirect reference creation XML 95d⟩
  }
  break;
```

The MKCALENDAR method takes an optional body with a mkcalendar element.

96c ⟨*webdav.dtd* 65c⟩+≡ ◁95e

```
<!ELEMENT mkcalendar (DAV:set)>
```

96d ⟨*XML validator names* 64d⟩+≡ (63b) ◁95f

```
mkcalendar
```

96e ⟨*unless method requires no body or XML body* 66b⟩+≡ (63a) ◁93c

```
&& req_method != HTTP_REQ_MKCALENDAR
```

96f ⟨*Check WebDAV XML body* 66c⟩+≡ (63a) ◁96b

```
case HTTP_REQ_MKCALENDAR:
  valid = !node->next && node->type == XML_ELEMENT_NODE &&
          xml_tag(node) == XML_NAME_MKCALENDAR &&
          check_ns(node, CALDAV_NS, NULL) &&
          (node = xml_child(node)) && !node->next &&
          node->type == XML_ELEMENT_NODE &&
          check_ns(node, DAV_NS, &dav_ns) &&
          xml_tag(node) == XML_NAME_SET && (node = xml_child(node)) &&
          !node->next && node->type == XML_ELEMENT_NODE &&
          check_ns(node, DAV_NS, &dav_ns) &&
          xml_tag(node) == XML_NAME_PROP && check_prop(node, TRUE);
  break;
```

Methods which require that no body be present are the GET, HEAD, DELETE, COPY, MOVE, and UNLOCK requests. There is no defined use for the OPTIONS body, but the standard implies that it should be accepted and ignored. The TRACE and CONNECT methods are assumed to be handled by the server; if not, any checks can be done in the application.

96g ⟨*unless method requires no body* 96g⟩≡ (63a)

```
&& req_method != HTTP_REQ_DELETE && req_method != HTTP_REQ_COPY
&& req_method != HTTP_REQ_MOVE   && req_method != HTTP_REQ_UNLOCK
&& req_method != HTTP_REQ_GET    && req_method != HTTP_REQ_HEAD
```

Chapter 9

Session Support

Persistent sessions require shared state between independent CGI invocations, possibly in different processes. If those independent CGI invocations are also on different machines, the shared state will need to be served off of a central server. With this in mind, the traditional approach is to use a database connection to a central server to store session state. Implementing this in a generic manner for all databases is not practical, so a simple single-host file-based session manager is presented here.

97a ⟨*(cs-glib.nw) Library* `cs-supt` *Members* 8a⟩+≡ ◁8a 113a▷

```
session.o
```

97b ⟨*session.c* 97b⟩≡

```
⟨(build.nw) Common C Header (imported)⟩

⟨CGI Session Support Variables 98f⟩
⟨CGI Session Support Functions 97c⟩
```

Initialization of the session routines is dependent on the method used to store the session. The default `init_cgi_session` is only to be used when the simple file-based sessions are used. Any other such routine should probably call `init_cgi` as well, since it is intended to be a replacement.

97c ⟨*CGI Session Support Functions* 97c⟩≡ (97b) 98c▷

```
void init_cgi_session(void)
{
  init_cgi();
  ⟨Initialize file-based session support 98g⟩
}
```

A session requires a unique identifier, and of course the associated persistent state. The identifier and session ID are stored in the CGI state. The session ID's memory is expected to be allocated from the heap, so it is automatically freed when necessary. The session HDF is likewise a freshly allocated HDF; transferring the state for template users will require an HDF copy.

97d ⟨*CGI State Members* 9e⟩+≡ (8g) ◁64f 98b▷

```
char *sessionid;
HDF *session;
```

97e ⟨*Free CGI state members* 18e⟩+≡ (11g) ◁62b

```
if(cgi_state->sessionid)
  free(cgi_state->sessionid);
if(cgi_state->session)
  hdf_destroy(&cgi_state->session);
```

The unique identifier is generated by a session creation function. It is expected that the state is already initialized with session identification information, so it expects that the `session` is non-empty, and creates the session with those initial values. If there happens to already be a session ID, it is simply ignored.

98a ⟨*CGI State Dependencies* 9c⟩+≡ (8g) ◁44a 100a▷

```
typedef NEOERR *(*session_create_cb)(cgi_state_t *cgi_state);
```

98b ⟨*CGI State Members* 9e⟩+≡ (8g) ◁97d 100b▷

```
session_create_cb create_session;
void *session_cb_data;
```

98c ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁97c 100c▷

```
NEOERR *create_session_state(cgi_state_t *cgi_state)
{
  NEOERR *nerr;
  time_t now;

  if(!cgi_state->session || !hdf_obj_child(cgi_state->session))
    return nerr_raise_msg("Empty session");
  if(cgi_state->sessionid) {
    free(cgi_state->sessionid);
    cgi_state->sessionid = NULL;
  }
  if(cgi_state->create_session)
    return (*cgi_state->create_session)(cgi_state);
  ⟨Create session using local files 99b⟩
  return nerr;
}
```

98d ⟨*(build.nw) Known Data Types* 9a⟩+≡ ◁63c 101a▷

```
session_create_cb,%
```

For files, the session ID is also a unqiue file name, as created by `mkstemp` in a configurable directory.

98e ⟨*CGI Session Support Configuration* 98e⟩≡ (149b)

```
# The default directory where local sessions are stored
# Clearing this parameter disables local file session support
#cgi_session_dir = /var/cache/cgi-sessions
```

98f ⟨*CGI Session Support Variables* 98f⟩≡ (97b)

```
static const char *session_dir;
```

98g ⟨*Initialize file-based session support* 98g⟩≡ (97c)

```
session_dir = getconf("cgi_session_dir", "/var/cache/cgi-sessions");
if(!*session_dir)
  session_dir = NULL;
else
  mkdir(session_dir, 0700); /* in case it doesn't exist yet */
```

99a ⟨*Build session file name* 99a⟩≡ (99b 101c 104)

```
GString *path = g_string_new(session_dir);
if(path->str[path->len - 1] != '/')
  g_string_append_c(path, '/');
g_string_append(path, cgi_state->sessionid);
```

New ticket IDs include the current date, masked to 32 bits. This guarantees that there will never be a repeated ticket, as long as the expiration time is longer than a few seconds, and no session lasts longer than 68 years. It is not necessary to lock the file from concurrent access, as the newly created file's name is not known to anyone but the creation function.

99b ⟨*Create session using local files* 99b⟩≡ (98c)

```
if(!session_dir)
  return STATUS_OK;
cgi_state->sessionid = (char *)"CS";
⟨Build session file name 99a⟩
cgi_state->sessionid = NULL;
time(&now);
g_string_append_printf(path, "%08XXXXXXX", (int)now);
int fd = mkstemp(path->str);
if(fd < 0) {
  g_string_free(path, TRUE);
  return nerr_raise_errno(NERR_IO, "Cannot create session state in %s",
                                   session_dir);
}
cgi_state->sessionid = strdup(strrchr(path->str, '/') + 1);
if(!cgi_state->sessionid) {
  close(fd);
  unlink(path->str);
  g_string_free(path, TRUE);
  return nerr_raise_errno(NERR_NOMEM, "cannot create session id");
}
⟨Write session file 99c⟩
close(fd);
if(nerr != STATUS_OK) {
  free(cgi_state->sessionid);
  cgi_state->sessionid = NULL;
}
g_string_free(path, TRUE);
```

The format of the session file is the standard HDF format. Writing the session file is harder than just `hdf_write_file`, because the session file is already open, and there is no direct access to `hdf_dump_format`, which takes an open file as a parameter. Writing to a memory string is the only simple alternative, so memory will be wasted for this operation.

99c ⟨*Write session file* 99c⟩≡ (99b 104)

```
char *s;

nerr = hdf_write_string(cgi_state->session, &s);
if(nerr == STATUS_OK) {
  if(debug) {
    cgi_puts("<pre>Write session:\n");
    cgi_puts_html_escape(cgi_state, s, FALSE);
    cgi_puts("--------\n</pre>\n");
  }
  int n, todo = strlen(s), off = 0;
```

99c ⟨*Write session file* 99c⟩≡ (99b 104)

```
  while(todo) {
    n = write(fd, s + off, todo);
    if(n < 0 && errno != EAGAIN && errno != EINTR) {
      nerr = nerr_raise_errno(NERR_IO, "writing session");
      unlink(path->str);
      break;
    }
    if(n > 0) {
      off += n;
      todo -= n;
    }
  }
  free(s);
}
```

Reading the session state should be an incremental operation. That way, a large amount of state can be stored, but only the needed state is passed around in the state structure. All elements under the passed-in root are destroyed. If there is no persistent state under the desired root, the root node itself will be created anyway. In order to make it easier to read in a particular part of the state based on a CGI parameter, the name is specified in two parts. Either part may be `NULL`. If both are non-`NULL`, the second is a child of the first.

100a ⟨*CGI State Dependencies* 9c⟩+≡ (8g) ◁98a 103a▷

```
typedef NEOERR *(*session_read_cb)(const char *name, const char *child,
                                   HDF *target, gboolean recursive,
                                   cgi_state_t *cgi_state);
```

100b ⟨*CGI State Members* 9e⟩+≡ (8g) ◁98b 103b▷

```
session_read_cb read_session;
```

100c ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁98c 103c▷

```
NEOERR *read_session_state(const char *root_name, const char *child_name,
                           gboolean recursive, cgi_state_t *cgi_state)
{
  NEOERR *nerr;
  HDF *root, *child;

  if(!cgi_state->sessionid)
    return nerr_raise(NERR_PARSE, "No session ID");
  if(child_name && !*child_name)
    child_name = NULL;
  if(!root_name || !*root_name) {
    root_name = child_name;
    child_name = NULL;
  }
  if(root_name && !*root_name)
    root_name = NULL;
  if(!root_name) {
    if(cgi_state->session)
      hdf_destroy(&cgi_state->session);
    if((nerr = hdf_init(&cgi_state->session)))
      return nerr;
    root = cgi_state->session;
  } else {
    if(!cgi_state->session && (nerr = hdf_init(&cgi_state->session)))
      return nerr;
    if((nerr = hdf_get_node(cgi_state->session, root_name, &root)))
      return nerr;
    if(child_name &&
```

100c ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁98c 103c▷

```
        (nerr = hdf_get_node(root, child_name, &root)))
      return nerr;
    while((child = hdf_obj_child(root)))
      hdf_remove_tree(root, hdf_obj_name(child));
  }
  if(cgi_state->read_session)
    return (*cgi_state->read_session)(root_name, child_name, root, recursive,
                                      cgi_state);
  ⟨Read session from local files 101c⟩
  return nerr;
}
```

101a ⟨*(build.nw) Known Data Types* 9a⟩+≡ ◁98d 103d▷

```
session_read_cb,%
```

However, reading just part of a full HDF file is not that simple. Instead, the entire file is read in, and the selected node is copied to the persistent state. Unlike with creation, no assumption can be made about simultaneous access. For this reason, the file is locked during the read. Also, in order to support expiring old sessions, the state files are touched at every access. This allows an external daemon to be used to flush out sessions with old modification times. Access times are often disabled in modern systems, so that is not sufficient.

101b ⟨*(build.nw) Common C Includes* 8b⟩+≡ ◁13a 102a▷

```
#include <utime.h>
```

101c ⟨*Read session from local files* 101c⟩≡ (100c)

```
int fd;
HDF *hdf;

if(!session_dir)
  return STATUS_OK;
if((nerr = hdf_init(&hdf)))
  return nerr;
⟨Build session file name 99a⟩
utime(path->str, NULL);
⟨Open and lock session file 102b⟩
g_string_free(path, TRUE);
if(nerr != STATUS_OK) {
  hdf_destroy(&hdf);
  return nerr;
}
⟨Read session file into hdf 102c⟩
close(fd);
if(nerr == STATUS_OK) {
  if(!root_name)
    cgi_state->session = root = hdf;
  else {
    child = hdf_get_obj(hdf, root_name);
    if(child && child_name)
      child = hdf_get_obj(child, child_name);
    if(child)
      nerr = hdf_copy(root, "", child);
    hdf_destroy(&hdf);
  }
  if(!recursive) {
    root = hdf_obj_child(root);
    while(root) {
      while((child = hdf_obj_child(root)))
```

101c ⟨*Read session from local files* 101c⟩≡ (100c)

```
        hdf_remove_tree(root, hdf_obj_name(child));
      root = hdf_obj_next(root);
    }
  }
}
```

The locks are done using advisory locks (`lockf`(3)). This means that even when locking others out isn't necessary, a lock must be obtained to obey others' locks on the file. Since these are write locks, the file is always opened in read-write mode.

102a ⟨*(build.nw) Common C Includes* 8b⟩+≡ ◁101b 113c▷

```
#include <sys/stat.h>
#include <fcntl.h>
```

102b ⟨*Open and lock session file* 102b⟩≡ (101c 104)

```
fd = open(path->str, O_RDWR);
if(fd < 0)
  nerr = nerr_raise_errno(NERR_IO, "opening session file %s", path->str);
else {
  while(lockf(fd, F_LOCK, 0)) {
    if(errno == EINTR || errno == EAGAIN)
      continue;
    close(fd);
    nerr = nerr_raise_errno(NERR_IO, "locking session");
    break;
  }
}
```

Once again, the file is already open, so reading the session is harder than just `hdf_read_file`. In this case the underlying function is not much different in that the entire file is sucked into memory before processing.

102c ⟨*Read session file into* `hdf` 102c⟩≡ (101c 104)

```
GString *fcont = g_string_sized_new(64);
int off = 0;
while(nerr == STATUS_OK) {
  int l = read(fd, fcont->str + off, fcont->allocated_len - off);
  if(l > 0) {
    off += l;
    if(off == fcont->allocated_len)
      g_string_set_size(fcont, fcont->allocated_len * 2);
    continue;
  } else if(!l)
    break;
  else if(errno != EAGAIN && errno != EINTR)
    nerr = nerr_raise_errno(NERR_IO, "reading session");
}
fcont->str[off] = 0;
if(debug) {
  cgi_puts("<pre>Read session:\n");
  cgi_puts_html_escape(cgi_state, fcont->str, FALSE);
  cgi_puts("--------\n</pre>\n");
}
if(nerr == STATUS_OK)
  nerr = hdf_read_string(hdf, fcont->str);
g_string_free(fcont, TRUE);
```

Finally, updating is potentially the most complex method. Updates do not write the entire state information out, especially since no flags are provided to indicate how much of the state has been read. Instead, the root of the state to update is provided. It is up to the caller to ensure that the result is stored in the in-memory session, if necessary. If the provided root is empty, the associated named root is cleared, both in the persistent storage and in the in-memory session. If the named root is blank or NULL, the entire session is destroyed.

It may also be necessary to generate a unique ID while updating the session data. In other words, rather than replacing a key, it may be necessary to append a new child to the key. Rather than requiring external locking and making the user do it, a pointer may be passed in which will be filled in with a node whose name is unique under the root, and under which the passed-in values are stored rather than the root itself. When deleting, the unique root parameter may point to a node which names a child of `root_name` to be be deleted instead of `root_name`.

103a ⟨*CGI State Dependencies* 9c⟩+≡ (8g) ◁100a

```
typedef NEOERR *(*session_update_cb)(const char *root_name, HDF *root,
                                     HDF **unique_root, cgi_state_t *cgi_state);
```

103b ⟨*CGI State Members* 9e⟩+≡ (8g) ◁100b

```
session_update_cb update_session;
```

103c ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁100c 107c▷

```
NEOERR *update_session_state(const char *root_name, HDF *root,
                             HDF **unique_root, cgi_state_t *cgi_state)
{
  NEOERR *nerr = STATUS_OK;

  if(!cgi_state->sessionid)
    return nerr_raise(NERR_PARSE, "No session ID");
  if(root && !hdf_obj_child(root) && !hdf_obj_value(root))
    root = NULL;
  if(root_name && !*root_name)
    root_name = NULL;
  if(unique_root && !root && !*unique_root)
    return nerr_raise(NERR_PARSE, "Cannot create unique key while deleting");
  if(unique_root && !root_name)
    return nerr_raise(NERR_PARSE, "Cannot create unique key under root");
  if(!root) {
    if(!root_name)
      hdf_destroy(&cgi_state->session);
    else if(!unique_root)
      hdf_remove_tree(cgi_state->session, root_name);
    else {
      HDF *child = hdf_get_obj(cgi_state->session, root_name);
      if(child)
        hdf_remove_tree(child, hdf_obj_name(*unique_root));
    }
  }
  if(cgi_state->update_session)
    return (*cgi_state->update_session)(root_name, root, unique_root, cgi_state);
  ⟨Update session using local files 104⟩
  return nerr;
}
```

103d ⟨*(build.nw) Known Data Types* 9a⟩+≡ ◁101a 114c▷

```
session_update_cb,%
```

To update the state file, the entire file must be read into memory while locked, and after updating the requested information, it must be written back before unlocking.

104 ⟨*Update session using local files* 104⟩≡ (103c)

```
if(!session_dir)
  return STATUS_OK;
⟨Build session file name 99a⟩
if(!root_name && !root)
  unlink(path->str);
else {
  int fd;
  HDF *hdf;

  if(root_name && (nerr = hdf_init(&hdf))) {
    g_string_free(path, TRUE);
    return nerr;
  }
  ⟨Open and lock session file 102b⟩
  if(nerr != STATUS_OK) {
    if(root_name)
      hdf_destroy(&hdf);
    g_string_free(path, TRUE);
    return nerr;
  }
  if(root_name) {
    ⟨Read session file into hdf 102c⟩
    if(lseek(fd, 0, SEEK_SET) < 0)
      ;  /* there is no reason this should fail */
  }
  if(nerr == STATUS_OK) {
    if(!unique_root) {
      if(root_name) {
        hdf_remove_tree(hdf, root_name);
        if(root)
          nerr = hdf_copy(hdf, root_name, root);
        if(hdf_obj_value(root))
          nerr_op(hdf_set_value(hdf, root_name, hdf_obj_value(root)));
      } else
        hdf = root;
    } else if(!root) {
      HDF *child = hdf_get_obj(hdf, root_name);
      if(child)
        hdf_remove_tree(child, hdf_obj_name(*unique_root));
    } else {
      char nbuf[11];
      HDF *child, *parent;
      int i = 0, j;

      nerr = hdf_get_node(hdf, root_name, &parent);
      if(nerr == STATUS_OK) {
        for(child = hdf_obj_child(parent); child; child = hdf_obj_next(child)) {
          j = atoi(hdf_obj_name(child));
          if(j >= i)
            i = j + 1;
        }
        sprintf(nbuf, "%d", i);
        nerr = hdf_get_node(parent, nbuf, unique_root);
        nerr_op(hdf_copy(*unique_root, "", root));
        if(hdf_obj_value(root))
          nerr_op(hdf_set_value(parent, nbuf, hdf_obj_value(root)));
      }
    }
  }
  if(nerr == STATUS_OK) {
```

104 ⟨*Update session using local files* 104⟩≡ (103c)

```
    if(ftruncate(fd, 0))
        ; /* ignore return value */
    HDF *olds = cgi_state->session;
    cgi_state->session = hdf;
    ⟨Write session file 99c⟩
    cgi_state->session = olds;
  }
  close(fd);
  hdf_destroy(&hdf);
  if(nerr != STATUS_OK) /* prevent corrupt sessions */
    unlink(path->str);
  g_string_free(path, TRUE);
}
```

In this simple session file scheme, session expiration is performed by a separate program. This program examines all files in the session directory, and deletes any older than a specified age. It can be called just once, and remain alive forever, or it can be called from cron on a regular basis; calling it from cron may incur significant overhead. It just uses `ftw` to look for old files. Since the `ftw` callback takes no user data, the expiration time is kept in a static variable.

Another option would be to run this in addition to the CGI threads in the main program. That approach has several advantages. There is no need to rememmber to start up a separate process. Sessions will not linger unintentionally, causing potential security issues. On the other hand, if the CGI is run with normal CGI rather than FastCGI, the directory scan will have to be done on every invocation. It is also so simple that it is unlikely to ever crash, so starting it up at system initialization is something that only needs to be remembered once. For added security, it can be added to inittab for the unlikely event of a crash.

105a ⟨*(build.nw) C Executables* 105a⟩≡ 135a▷

```
delete_old_files \
```

105b ⟨*delete_old_files.c* 105b⟩≡

```
⟨(build.nw) Common C Header (imported)⟩
#include <ftw.h>

void help(void)
{
  fputs("Delete old state files\n"
        "Usage: delete_old_files <dir> <exp_time sec> [<next check sec>]\n"
        "If <next check sec> is given, sleep that amount and check again\n"
        "forever.  If <next check sec> is 0, use <exp time sec> / 2.\n",
        stderr);
  exit(1);
}

static time_t exp_time;

int delete_old_files(const char *fpath, const struct stat *sb, int typeflag)
{
  if(typeflag == FTW_F && sb->st_mtime < exp_time)
    unlink(fpath);
  return 0;
}

int main(int argc, const char **argv)
{
  int next, exps;
```

105b ⟨*delete_old_files.c* 105b⟩≡

```
  if(argc < 3 || argc > 4)
    help();
  if(chdir(argv[1])) {
    perror(argv[1]);
    help();
  }
  exps = atoi(argv[2]);
  if(exps <= 0)
    help();
  if(argc > 3) {
    next = atoi(argv[3]);
    if(next <= 0)
      next = exps / 2;
  } else
    next = 0;
  time(&exp_time);
  exp_time -= exps;
  if(next)
    while(1) {
      ftw(".", delete_old_files, 2);
      sleep(next);
      time(&exp_time);
       exp_time -= exps;
    }
  else
    ftw(".", delete_old_files, 2);
  return 0;
}
```

Although finding an active session's ID is not likely, the session must still be keyed to the client. The client can never be completely reliably identified, but a few attributes tend to be fairly stable:

- REMOTE_ADDR (`CGI.RemoteAddress`) — This is only unstable if the user switches between a caching proxy and direct access, or is behind a firewall that masks this. For security, these cases will be ignored and this parameter is always used.
- HTTP_USER_AGENT (`HTTP.UserAgent`) — Again, this is unstable if the user switches between a caching proxy and direct access, but the proxy should use the same user agent anyway to ensure that the correct compatibility flags are set on remote applications. Again, this parameter will always be used.
- REMOTE_USER (`CGI.RemoteUser`) — Users may log on, or not. The session should definitely terminate if the user ID changes.
- An arbitrary CGI parameter or cookie passed around with the session ID — the previous parameters prevent the cookie from being passed around to other people for authentication. However, if the REMOTE_USER parameter is not set by the web server on every access, it should be passed around with the session ID.

These parameters are saved in the state with the same base name, but `Session` as the parent.

106 ⟨*Get Session Invariants* 106⟩≡ (107c)

```
const char *remote_addr, *user_agent, *remote_user;

remote_addr = cgi_env_val("CGI.RemoteAddress", "127.0.0.1");
user_agent = cgi_header_val("UserAgent", "telnet");
remote_user = cgi_state->userid ? cgi_state->userid : "";
```

107a ⟨*CGI Support Global Definitions* 8g⟩+≡ (8d) ◁76c

```
#define cgi_session_val(n, d) hdf_get_value(cgi_state->session, n, d)
#define cgi_session_val_int(n, d) hdf_get_int64_value(cgi_state->session, n, d)
#define cgi_session_obj(n) hdf_get_obj(cgi_state->session, n)
#define cgi_session_set(n, v) hdf_set_value(cgi_state->session, n, v)
#define cgi_session_set_int(n, v) hdf_set_int64_value(cgi_state->session, n, v)
```

107b ⟨*(build.nw) C Prototypes* 12c⟩+≡ ◁29a 115a▷

```
const char *cgi_session_val(const char *name, const char *def_val);
gint64 cgi_session_val_int(const char *name, gint64 def_val);
HDF *cgi_session_obj(const char *name);
NEOERR *cgi_session_set(const char *name, const char *value);
NEOERR *cgi_session_set_int(const char *name, gint64 value);
```

107c ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁103c 107d▷

```
NEOERR *set_session_validation(cgi_state_t *cgi_state)
{
  ⟨Get Session Invariants 106⟩
  NEOERR *nerr = STATUS_OK;
  if(!cgi_state->session)
    nerr = hdf_init(&cgi_state->session);
  if(nerr == STATUS_OK)
    nerr = cgi_session_set("Session.RemoteAddress", remote_addr);
  if(nerr == STATUS_OK)
    nerr = cgi_session_set("Session.UserAgent", user_agent);
  if(nerr == STATUS_OK)
    nerr = cgi_session_set("Session.RemoteUser", remote_user);
  return nerr;
}

gboolean validate_session(cgi_state_t *cgi_state)
{
  if(!cgi_state->session)
    return FALSE;
  ⟨Get Session Invariants 106⟩
  return
    !strcmp(cgi_session_val("Session.RemoteAddress", ""),
            remote_addr) &&
    !strcmp(cgi_session_val("Session.UserAgent", ""),
            user_agent) &&
    !strcmp(cgi_session_val("Session.RemoteUser", ""),
            remote_user);
}
```

Integration of the session into a CGI program requires use of either CGI parameters or cookies. Some people turn cookies off in the browser, so the best approach is to use both, and remove the CGI parameters if the cookies appear to be working. An additional CGI parameter is set to indicate that cookies should no longer be tried. The parameter names begin with a short prefix to avoid sending long names around all the time (`ses.`). The session ID is sent as `id` and the user ID is sent as `un`. The no-cookie flag is `nc`.

107d ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁107c 108a▷

```
NEOERR *print_session_cgi_parms(cgi_state_t *cgi_state, gboolean force)
{
  NEOERR *nerr;
```

107d ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁107c 108a▷

```
  if(!cgi_state->sessionid)
    return STATUS_OK;
  if(!force && cgi_cookie_val("ses.id", NULL))
    return STATUS_OK;
  if(!cgi_puts("<input type=\"hidden\" name=\"ses.id\" value=\""))
    return nerr_raise_errno(NERR_IO, "cgi write");
  if((nerr = cgi_puts_html_escape(cgi_state, cgi_state->sessionid, FALSE)))
    return nerr;
  if(!cgi_puts("\">\n"))
    return nerr_raise_errno(NERR_IO, "cgi write");
  if(cgi_state->userid && !cgi_getenv("REMOTE_USER")) {
    if(!cgi_puts("<input type=\"hidden\" name=\"ses.un\" value=\""))
      return nerr_raise_errno(NERR_IO, "cgi write");
    if((nerr = cgi_puts_html_escape(cgi_state, cgi_state->userid, FALSE)))
      return nerr;
    if(!cgi_puts("\">\n"))
      return nerr_raise_errno(NERR_IO, "cgi write");
  }
  if(!force && cgi_parm_val("ses.id", NULL)) /* 2nd go around */
    if(!cgi_puts("<input type=\"hidden\" name=\"ses.nc\" value=\"y\">\n"))
      return nerr_raise_errno(NERR_IO, "cgi write");
  return STATUS_OK;
}
```

108a ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁107d 108b▷

```
void g_string_append_session_cgi_parms(cgi_state_t *cgi_state, gboolean first,
                                       GString *buf, gboolean force)
{
  if(!cgi_state->sessionid)
    return;
  if(!force && cgi_cookie_val("ses.id", NULL))
    return;
  if(first)
    g_string_append_c(buf, '?');
  else
    g_string_append_c(buf, '&');
  g_string_append(buf, "ses.id=");
  g_string_append_uri_escaped(buf, cgi_state->sessionid, NULL, FALSE);
  if(cgi_state->userid && !cgi_getenv("REMOTE_USER")) {
    g_string_append(buf, "&ses.un=");
    g_string_append_uri_escaped(buf, cgi_state->userid, NULL, FALSE);
  }
  if(!force && cgi_parm_val("ses.id", NULL)) /* 2nd go around */
    g_string_append(buf, "&ses.nc=y");
}
```

108b ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁108a 109a▷

```
NEOERR *print_session_cookie(cgi_state_t *cgi_state, gboolean force)
{
  NEOERR *nerr = STATUS_OK;
  const char *s;
  if(!cgi_state->sessionid)
    return STATUS_OK;
  s = cgi_cookie_val("ses.id", NULL);
  if(!s && !force && cgi_parm_val("ses.nc", NULL))
    return STATUS_OK;
  if(!s || strcmp(s, cgi_state->sessionid)) {
    if(cgi_printf("Set-Cookie: ses.id=%s; path=/\n", cgi_state->sessionid) < 1)
      nerr = nerr_raise_errno(NERR_IO, "CGI Write");
  }
  if(nerr == STATUS_OK && cgi_state->userid && !cgi_getenv("REMOTE_USER")) {
```

108b ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁108a 109a▷

```
    s = cgi_cookie_val("ses.un", NULL);
    if(!s || strcmp(s, cgi_state->userid)) {
      if(cgi_printf("Set-Cookie: ses.un=%s; path=/\n", cgi_state->userid) < 1)
        nerr = nerr_raise_errno(NERR_IO, "CGI Write");
    }
  }
  return nerr;
}
```

Supporting the same things in templates is a bit tricker. If a template is to print the cookie, the easiest way to convey that is to set a variable. Likewise, a variable is set to request dumping the form variables. Rather than setting the variable to the text to be printed, ClearSilver macros are provided to dump the variable's hierarchy in the appropriate format. This makes adding more session variables easier, and simplifies the memory managment in the C code.

109a ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁108b 110b▷

```
NEOERR *set_session_tmpl_vars(cgi_state_t *cgi_state, gboolean force_cookie,
                              gboolean force_form)
{
  NEOERR *nerr = STATUS_OK;

  if(!cgi_state->sessionid)
    return STATUS_OK;
  if(force_form || !cgi_cookie_val("ses.id", NULL)) {
    nerr_op(cgi_env_set("CGI.SessionParms.ses.id", cgi_state->sessionid));
    if(cgi_state->userid && !cgi_getenv("REMOTE_USER"))
      nerr_op(cgi_env_set("CGI.SessionParms.ses.un", cgi_state->userid));
    if(!force_form && cgi_parm_val("ses.id", NULL)) /* 2nd go around */
      nerr_op(cgi_env_set("CGI.SessionParms.ses.nc", "y"));
  }
  const char *s = cgi_cookie_val("ses.id", NULL);
  if(!s && !force_cookie && cgi_parm_val("ses.nc", NULL))
    return nerr;
  if(!s || strcmp(s, cgi_state->sessionid))
    nerr_op(cgi_env_set("CGI.SessionCookies.ses.id", cgi_state->sessionid));
  if(cgi_state->userid && !cgi_getenv("REMOTE_USER"))
    nerr_op(cgi_env_set("CGI.SessionCookies.ses.un", cgi_state->userid));
  return nerr;
}
```

109b ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁7d 109c▷

```
def:print_session_cookie(prefix, cookie) ?><?cs
  if:subcount(cookie) > 0 ?><?cs
   each:c = cookie ?><?cs
      call:print_session_cookie(prefix + name(cookie) + ".", c) ?><?cs
   /each ?><?cs
 else
   ?>Set-Coookie: <?cs var:prefix + name(cookie) ?>=<?cs var:cookie ?>; path=/
<?cs
 /if ?><?cs
/def
?><?cs
def:print_session_cookies() ?><?cs
  each:parm = CGI.SessionCookies ?><?cs
    call:print_session_cookie("", parm) ?><?cs
  /each ?><?cs
/def ?><?cs
```

109c ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁109b 110a▷

```
def:print_session_parm(prefix, parm) ?><?cs
  if:subcount(parm) > 0 ?><?cs
   each:c = parm ?><?cs
      call:print_session_parm(prefix + name(parm) + ".", c) ?><?cs
   /each ?><?cs
 else
   ?><input type="hidden" name="<?cs var:prefix + name(parm)
            ?> value="<?cs var:parm ?>">
<?cs
 /if ?><?cs
/def
?><?cs
def:print_session_parms() ?><?cs
  each:parm = CGI.SessionParms ?><?cs
    call:print_session_parm("", parm) ?><?cs
  /each ?><?cs
/def ?><?cs
```

110a ⟨*Generic ClearSilver HTML Macros* 4d⟩+≡ (4b) ◁109c

```
def:add_session_parm(prefix, parm, first) ?><?cs
  if:subcount(parm) > 0 ?><?cs
   each:c = parm ?><?cs
      call:add_session_parm(prefix + name(parm) + ".", c, first) ?><?cs
      set:first = 0 ?><?cs
   /each ?><?cs
 else
   ?><?cs if:first ?>?<?cs else ?>&<?cs /if ?><?cs
      var:url_escape(prefix + name(parm)) ?>=<?cs var:url_escape(parm) ?><?cs
      set:first = 0 ?><?cs
 /if ?><?cs
/def
?><?cs
def:add_session_parms(first) ?><?cs
  each:parm = CGI.SessionParms ?><?cs
    call:add_session_parm("", parm, first) ?><?cs
    set:first = 0 ?><?cs
  /each ?><?cs
/def ?><?cs
```

To actually set the session ID, the parameters are checked first, and if the parameters do not indicate a valid session, a new session is created. If this function is called multiple times, the session already in progress is retained with no further action.

110b ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁109a 111d▷

```
NEOERR *update_cgi_session(cgi_state_t *cgi_state)
{
  NEOERR *nerr;

  if(cgi_state->sessionid)
    return STATUS_OK;
  ⟨Check CGI session parameters 111a⟩
  if(cgi_state->sessionid) {
    ⟨Verify CGI session returning STATUS_OK if OK 111b⟩
  }
  if(cgi_state->sessionid) {
    nerr = update_session_state(NULL, NULL, NULL, cgi_state); /* delete session */
    nerr_ignore(&nerr);
  }
  ⟨Create new CGI session 111c⟩
  return nerr;
}
```

The session parameter can come from the cookies or the CGI parameters.

111a ⟨*Check CGI session parameters* 111a⟩≡ (110b)

```
cgi_state->sessionid = cgi_cookie_val("ses.id", NULL);
if(!cgi_state->sessionid)
  cgi_state->sessionid = cgi_parm_val("ses.id", NULL);
if(cgi_state->sessionid) {
  cgi_state->sessionid = strdup(cgi_state->sessionid);
  if(!cgi_state->sessionid)
    return nerr_raise_errno(NERR_NOMEM, "Session");
}
gboolean userid_from_session = FALSE;
if(!cgi_state->userid && cgi_state->sessionid) {
  cgi_state->userid = cgi_cookie_val("ses.un", NULL);
  if(!cgi_state->userid)
    cgi_state->userid = cgi_parm_val("ses.un", NULL);
  userid_from_session = TRUE;
}
```

To verify a session ID, the session parameters are read from the file and the user location is verified. No lock needs to be retained, because the user identification information should never change.

111b ⟨*Verify CGI session returning* `STATUS_OK` *if OK* 111b⟩≡ (110b)

```
nerr = read_session_state("Session", NULL, TRUE, cgi_state);
if(nerr == STATUS_OK && !validate_session(cgi_state))
  nerr = nerr_raise_msg("Invalid session");
if(nerr == STATUS_OK && userid_from_session && cgi_state->userid)
  nerr = cgi_env_set("CGI.RemoteUser", cgi_state->userid);
if(nerr != STATUS_OK) {
  if(debug) {
    ⟨(cs-glib.nw) Convert nerr to err_str (imported)⟩
    cgi_puts("<pre>");
    cgi_puts_html_escape(cgi_state, err_str.buf, FALSE);
    cgi_puts("</pre>");
    string_clear(&err_str);
  } else
    nerr_ignore(&nerr);
  if(cgi_state->session)
    hdf_destroy(&cgi_state->session);
} else
  return STATUS_OK;
```

To create a session ID, the user location becomes the session state, and a new state is created. If the user ID changes after this, the state must be updated.

111c ⟨*Create new CGI session* 111c⟩≡ (110b)

```
if(userid_from_session)
  cgi_state->userid = NULL;
nerr = set_session_validation(cgi_state);
if(nerr == STATUS_OK)
  nerr = create_session_state(cgi_state);
```

To destroy a session, the file is removed. No attempt is made to clear the cookies, because the fact that they are invalid will make them be ignored anyway. On the other hand, if the caller really wants to try to clear the cookies, the appropriate flag can be set.

111d ⟨*CGI Session Support Functions* 97c⟩+≡ (97b) ◁110b

```
void logout_cgi_session(cgi_state_t *cgi_state, gboolean print_cookies)
{
  NEOERR *nerr;
  nerr = update_session_state(NULL, NULL, NULL, cgi_state);
  nerr_ignore(&nerr);
  if(print_cookies) {
    nerr_op(cgi_cookie_clear(cgi_state->cgi, "ses.id", NULL, "/"));
    nerr_op(cgi_cookie_clear(cgi_state->cgi, "ses.un", NULL, "/"));
    nerr_ignore(&nerr);
  }
}
```

Chapter 10

Database Session Support

Since the session support is intended to work with databases as well, a sample implementation using ODBC[1]

113a ⟨*(cs-glib.nw) Library* `cs-supt` *Members* 8a⟩+≡ ◁97a 139a▷

```
db_session.o
```

113b ⟨*db_session.c* 113b⟩≡

```
⟨(build.nw) Common C Header (imported)⟩

⟨CGI Database Session Variables 114e⟩
⟨CGI Database Session Functions 117e⟩
```

113c ⟨*(build.nw) Common C Includes* 8b⟩+≡ ◁102a 114a▷

```
#define _OBJC_OBJC_H_ 1 /* suppress iodbc BOOL */
#include <sql.h>
#include <sqlext.h>
```

113d ⟨*(build.nw) makefile.config* 7f⟩+≡ ◁29e

```
# CFLAGS for ODBC
ODBC_INC=
# ODBC library
ODBC_LIB=-lodbc
```

113e ⟨*(build.nw) makefile.vars* 7g⟩+≡ ◁61d 142c▷

```
EXTRA_CFLAGS += $(ODBC_INC)
EXTRA_LDFLAGS += $(ODBC_LIB)
```

In order to remain as database-neutral as possible, some efficiencies provided by various databases must be avoided. These include auto-incrementing integer fields and foreign key constraints with cascading deletes. The only feature which is required is transaction support, and even that is limited as much as possible. Also, in order to allow non-session access to the database, some database support is provided in a separate header file, and some required functions are also exported.

[1] There is no fixed convenient link to a page describing ODBC. The specific implementation used here is that provided by unixODBC (`http://www.unixodbc.org`), iODBC (`http://www.iodbc.org`), and various drivers. ODBC is used because it is the only method known to me to support multiple databases with input and output binding and dynamic SQL. Embedded SQL does not support as many databases, especially with consistent dynamic SQL, and libdbi does not support input parameter binding.

113f ⟨*dbase.h* 113f⟩≡

```
#ifndef _DBASE_H
#define _DBASE_H
⟨Database Support Definitions 114b⟩
#endif
```

114a ⟨*(build.nw) Common C Includes* 8b⟩+≡ ◁113c 142b▷

```
#include "dbase.h"
```

In particular, the ODBC connection parameters are made available, along with some functions to use those parameters, assuming a local variable named `conn` holds a pointer to them.

114b ⟨*Database Support Definitions* 114b⟩≡ (113f) 114d▷

```
typedef struct db_conn {
  SQLHDBC dbc;
  SQLHSTMT stmt;
  ⟨CGI Database Session Parameters 124b⟩
} db_conn;
```

114c ⟨*(build.nw) Known Data Types* 9a⟩+≡ ◁103d 166▷

```
db_conn,%
```

114d ⟨*Database Support Definitions* 114b⟩+≡ (113f) ◁114b 117d▷

```
#define db_exec(s)  SQLExecDirect(conn->stmt, (SQLCHAR *)s, SQL_NTS)
#define db_prep(s)  SQLPrepare(conn->stmt, (SQLCHAR *)s, SQL_NTS)
#define db_trans(t) SQLEndTran(SQL_HANDLE_DBC, conn->dbc, t)
#define db_commit() do { \
  if(SQL_SUCCEEDED(ret)) { \
    ret = db_trans(SQL_COMMIT); \
    if(!SQL_SUCCEEDED(ret)) \
      dbcerr = TRUE; \
  } \
} while(0)
#define db_commit_keeperr() do { \
  if(SQL_SUCCEEDED(ret)) \
    db_commit(); \
  else \
    db_trans(SQL_COMMIT); \
} while(0)
#define db_next() SQLFreeStmt(conn->stmt, SQL_CLOSE);
#define db_bind64(p, i) \
  SQLBindParameter(conn->stmt, p, SQL_PARAM_INPUT, SQL_C_UBIGINT, SQL_BIGINT, \
                   0, 0, &i, 0, NULL)
#define db_bindsl(p, s, l) \
  SQLBindParameter(conn->stmt, p, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR, \
                   l, 0, (void *)s, 0, &l)
extern SQLLEN db_nullen;
#define db_bindnull(p) \
  SQLBindParameter(conn->stmt, p, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR, \
                   0, 0, NULL, 0, &db_nullen)
#define db_binds(p, s)      (s) == NULL ? db_bindnull(p) : \
  SQLBindParameter(conn->stmt, p, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR, \
                   0, 0, (void *)s, 0, NULL)
```

114e ⟨*CGI Database Session Variables* 114e⟩≡ (113b) 115b▷

```
SQLLEN db_nullen = SQL_NULL_DATA;
```

115a ⟨*(build.nw) C Prototypes* 12c⟩+≡ ◁107b 122c▷

```
SQLRETURN db_exec(const char *sql);
SQLRETURN db_prep(const char *sql);
SQLRETURN db_trans(SQLSMALLINT type);
void db_commit(void);
void db_commit_keeperr(void);
SQLRETURN db_next(void);
SQLRETURN db_bind64(int param, guint64 &val);
SQLRETURN db_bindsl(int param, const char *str, SQLLEN len);
SQLRETURN db_bindnull(int param);
SQLRETURN db_binds(int param, const char *str);
```

In addition, debugging may be enabled. This is all session-specific and should normally not be needed, so it is enabled by compile directive.

115b ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁114e 117c▷

```
#ifdef DEBUG_DBSESS
SQLRETURN debug_sqlendtran(db_conn *conn, SQLSMALLINT c)
{
  SQLRETURN ret = SQLEndTran(SQL_HANDLE_DBC, conn->dbc, c);
  SQLExecDirect(conn->stmt, (SQLCHAR *)"select * from sessval", SQL_NTS);
  puts("commiting; new values:");
  while(SQL_SUCCEEDED(SQLFetch(conn->stmt))) {
    guint64 id, parent;
    char name[33];
    char value[1025];
    SQLLEN vallen;

    SQLGetData(conn->stmt, 1, SQL_C_UBIGINT, &id, 0, NULL);
    SQLGetData(conn->stmt, 2, SQL_C_UBIGINT, &parent, 0, NULL);
    SQLGetData(conn->stmt, 3, SQL_C_CHAR, name, sizeof(name), NULL);
    SQLGetData(conn->stmt, 4, SQL_C_CHAR, value, sizeof(value), &vallen);
    if(vallen == SQL_NULL_DATA)
      strcpy(value, "*NULL*");
    printf("%08d %08d %32s %s\n", (int)id, (int)parent, name, value);
  }
  db_next();
  puts("-----");
  fflush(stdout);
  return ret;
}

static SQLRETURN debug_execdirect(db_conn *conn, const SQLCHAR *s)
{
  printf("SQLR: %s\n", s);
  fflush(stdout);
  return db_exec(s);
}

static SQLRETURN debug_prep(db_conn *conn, const SQLCHAR *s)
{
  printf("SQLP: %s\n", s);
  fflush(stdout);
  return db_prep(s);
}

static SQLRETURN debug_bind(SQLHSTMT st, int n, int t2, int t3, int l,
                            void *v, SQLLEN *lp)
{
  if(t2 == SQL_C_CHAR)
    printf("bind %d: %s\n", n, v ? (char *)v : "*NULL*");
  else
```

115b ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁114e 117c▷

```
    printf("bind %d: %llu\n", n, (ullong)*(guint64 *)v);
  fflush(stdout);
  return SQLBindParameter(st, n, SQL_PARAM_INPUT, t2, t3,
                          l, 0, v, 0, lp);
}

#define SQLEndTran(a,b,c) debug_sqlendtran(conn, c)
#define SQLExecDirect(st, sql, l) debug_execdirect(conn, sql)
#define SQLPrepare(st, sql, l) debug_prep(conn, sql)
#define SQLBindParameter(st, n, t1, t2, t3, l, p, v, ol, lp) \
   debug_bind(st, n, t2, t3, l, v, lp)
#endif
```

Creating tables is not always possible, and may require additional steps (like creating a database to store the table), so a separate script is provided. This is a sample script, and may need to be edited for the specific database to be used.

116a ⟨*(build.nw) Plain Files* 7h⟩+≡ ◁7h 149c▷

```
sessdbcreate.sql \
```

The names for the session variables can be controlled, so leaving them at a relatively short length, such as 32, is not unreasonable. However, the values are supplied by the user, so it is not possible to force an arbitrary limit. For some databaseses, this is not an issue: they can store and manipulate strings of arbitrary length in any field. Others treat long strings as binary objects, though, and disallow searching and other forms of manipulation. For this reason a separate table is used to store extension chains for the values. Searching by value would have to be done in two steps: using the database to search on the primary table's value, and then searching the remainder manually. The database cannot be relied on to append strings of indefinite length, so it cannot be used directly. A pointer is stored to the parent record as well, to make deletes (especially with cascading) easier. This column must be created after the parent table or the reference will not get created.

116b ⟨*sessdbcreate.sql* 116b⟩≡ 116c▷

```
create table sessvalext (
  id      bigint primary key not null unique,
  ext     bigint references sessvalext on delete cascade,
  -- parent bigint;
  -- make this a large text field of any arbitrary type
  -- probably best to use text in postgres & sqlite
  value   varchar(32768) not null
);
```

Everything is managed with one session table, since the API provides no special fields which would make a separate session table more useful. Each value has of course a `name` and an optional `value`. Children are linked in via the `parent` field; rather than using the name as the parent key, a separate integer `id` field is used. Uniqueness in the hierarchy is enforced via a uniqueness constraint on the `name` and `parent` fields, but the constraint is not relied upon.

116c ⟨*sessdbcreate.sql* 116b⟩+≡ ◁116b 117a▷

```
create table sessval (
  id      bigint primary key not null unique,
  parent  bigint not null references sessval on delete cascade,
  name    varchar(32) not null,
  value   varchar(126),
  valext  bigint references sessvalext,
  unique (parent, name)
);
```

116c ⟨*sessdbcreate.sql* 116b⟩+≡ ◁116b 117a▷

```
alter table sessvalext add parent bigint references sessval on delete cascade;
-- create index sessval_fk on sessval (parent);
-- create index sessval_uk on sessval (parent, name);
```

Since auto-incrementing fields are non-portable, new `id` values could either be the largest current value plus one, or they could come from a manually incremented field. The former enables a race condition wherein a record is identified, but before an additional operation is performed, other processess delete the record and create a new one with the same `id` but different values, so the latter is used. By incementing the field within a transaction, a unique identifier can be obtained every time. This does limit the total number of record insertions to the maximum integer value, but this can be alleviated by occasionally dropping and recreating the table during maintenance periods. Any database which does not support transactions of this sort cannot be used.

Rather than creating a new table with a field meant for the automatic increment function, a dummy entry is used. This could either use the `value` field for the counter, which would require integer-to-string conversions more frequently than necessary, or it could use the `id` field itself as the counter, always providing a value and then updating to the next available value. The latter option would probably be more efficient, but using the `value` field is easier to implement. The first value is always the root of all other entries, and its value is always the last inserted `id`.

The SQL requires a type cast, so the underlying data type must be known. For most databases, this will the the SQL standard `bigint` type. For databases which do not support this type, another type can be configured. Also, even though it is possible to query the length of the value field from ODBC, the queried length may be inaccurate, and it is easier to just get the length from configuration.

117a ⟨*sessdbcreate.sql* 116b⟩+≡ ◁116c 119a▷

```
insert into sessval (id, parent, name, value) values(1,1,1,1);
```

117b ⟨*Session Value Support Configuration* 117b⟩≡ (120b)

```
# The SQL data type used for database table keys
#id_field_type = bigint

# The maximum length of a session value before extension
# 0 means there is no maximum
#sessval_length = 126

# The maximum length of an extension before further extension
# 0 means there is no maximum
#strext_length = 32768
```

117c ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁115b 121a▷

```
const char *id_field_type = "bigint";
static guint64 sessval_length = 126;
guint64 strext_length = 32768;
```

117d ⟨*Database Support Definitions* 114b⟩+≡ (113f) ◁114d 122b▷

```
extern const char *id_field_type;
extern guint64 strext_length;
```

117e ⟨*CGI Database Session Functions* 117e⟩≡ (113b) 118▷

```
NEOERR *db_get_next_id(db_conn *conn, const char *table, guint64 *newid)
{
  SQLRETURN ret;
  NEOERR *nerr = STATUS_OK;
  gboolean dbcerr = FALSE;
  GString *query = g_string_new("");

  g_string_printf(query, "update %s set value = cast(value as %s) + 1"
                         " where id = 1", table, id_field_type);
  ret = db_exec(query->str);
  if(SQL_SUCCEEDED(ret)) {
    db_next();
    g_string_truncate(query, 0);
    g_string_printf(query, "select value from %s where id = 1", table);
    ret = db_exec(query->str);
    if(SQL_SUCCEEDED(ret))
      ret = SQLFetch(conn->stmt);
    if(SQL_SUCCEEDED(ret))
      ret = SQLGetData(conn->stmt, 1, SQL_C_UBIGINT, newid, sizeof(*newid), NULL);
  }
  if(!SQL_SUCCEEDED(ret)) {
    ⟨Set nerr for ODBC error 123d⟩
  }
  db_next();
  db_commit();
  g_string_free(query, TRUE);
  return nerr;
}
```

118 ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁117e 119b▷

```
static NEOERR *create_sessval(db_conn *conn, guint64 parent, const char *name,
                              const char *val, guint64 *newid)
{
  SQLRETURN ret;
  NEOERR *nerr = STATUS_OK;
  gboolean dbcerr = FALSE;
  SQLLEN vlen = val ? strlen(val) : 0, vlen_small = vlen;

  if(vlen > sessval_length)
    vlen_small = sessval_length;
  nerr_op(db_get_next_id(conn, "sessval", newid));
  if(nerr != STATUS_OK)
    return nerr;
  char tname[22];
  if(!name) {
    sprintf(tname, ".%llu", (ullong)*newid);
    name = tname;
  }
  ret = db_prep("insert into sessval (id, parent, name, value) "
                "values (?, ?, ?, ?)");
  if(SQL_SUCCEEDED(ret))
    ret = db_bind64(1, *newid);
  if(SQL_SUCCEEDED(ret))
    ret = db_bind64(2, parent);
  if(SQL_SUCCEEDED(ret))
    ret = db_binds(3, name);
  if(SQL_SUCCEEDED(ret))
    ret = vlen ? db_bindsl(4, val, vlen_small) : db_bindnull(4);
  if(SQL_SUCCEEDED(ret))
    ret = SQLExecute(conn->stmt);
  if(!SQL_SUCCEEDED(ret)) {
    ⟨Set nerr for ODBC error 123d⟩
  }
```

118 ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁117e 119b▷

```
  db_next();
  if(nerr == STATUS_OK && vlen > sessval_length) {
    guint64 strext;
    nerr_op(db_store_string_ext(conn, "sessvalext", *newid,
                                val + sessval_length,
                                vlen - sessval_length, &strext));
    if(nerr == STATUS_OK) {
      ret = db_prep("update sessval set valext = ? where id = ?");
      if(SQL_SUCCEEDED(ret))
        ret = db_bind64(1, strext);
      if(SQL_SUCCEEDED(ret))
        ret = db_bind64(2, *newid);
      if(SQL_SUCCEEDED(ret))
        ret = SQLExecute(conn->stmt);
      if(!SQL_SUCCEEDED(ret)) {
        ⟨Set nerr for ODBC error 123d⟩
      }
    }
  }
  db_commit();
  return nerr;
}
```

Similarly, string extensions are done by using a special ID at the start of the table. Since other tables may potentially take advantage of string extentions, the actual table name is passed in as a parameter.

119a ⟨*sessdbcreate.sql* 116b⟩+≡ ◁117a

```
insert into sessvalext (id, value) values(1,1);
```

119b ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁118 120a▷

```
NEOERR *db_store_string_ext(db_conn *conn, const char *table, guint64 parent,
                            const char *ext, SQLLEN len, guint64 *id)
{
  SQLRETURN ret;
  NEOERR *nerr = STATUS_OK;
  gboolean dbcerr = FALSE;
  guint64 strext = 0;

  if(len > strext_length) {
    nerr_op(db_store_string_ext(conn, table, parent, ext + strext_length,
                                len - strext_length, &strext));
    if(nerr != STATUS_OK)
      return nerr;
    len = strext_length;
  }
  nerr_op(db_get_next_id(conn, table, id));
  if(nerr == STATUS_OK) {
    GString *query = g_string_new("");
    g_string_printf(query, "insert into %s (id, ext, parent, value) "
                           "values (?, ?, ?, ?)", table);
    ret = db_prep(query->str);
    if(SQL_SUCCEEDED(ret))
      ret = db_bind64(1, *id);
    if(SQL_SUCCEEDED(ret)) {
      if(strext)
        ret = db_bind64(2, strext);
      else
        ret = db_bindnull(2);
    }
    if(SQL_SUCCEEDED(ret))
      db_bind64(3, parent);
```

119b ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁118 120a▷

```
    if(SQL_SUCCEEDED(ret))
      ret = db_bindsl(4, ext, len);
    if(SQL_SUCCEEDED(ret))
      ret = SQLExecute(conn->stmt);
    if(!SQL_SUCCEEDED(ret)) {
      ⟨Set nerr for ODBC error 123d⟩
    }
    db_next();
    db_commit();
  }
  if(nerr != STATUS_OK && strext)
    db_delete_strext(conn, table, strext);
  return nerr;
}
```

120a ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁119b 121b▷

```
void db_delete_strext(db_conn *conn, const char *table, guint64 strext)
{
  SQLRETURN ret = 0;
  gboolean dbcerr = FALSE;
  guint64 strext_next = 0;
  GString *query1 = g_string_new(""), *query2 = g_string_new("");

  g_string_printf(query1, "select id from %s where ext = ?", table);
  g_string_printf(query2, "delete from %s where id = ?", table);
  while(strext) {
    if(!dbcascade) {
      ret = db_prep(query1->str);
      if(SQL_SUCCEEDED(ret))
        ret = db_bind64(1, strext);
      if(SQL_SUCCEEDED(ret))
        ret = SQLExecute(conn->stmt);
      if(SQL_SUCCEEDED(ret)) {
        ret = SQLFetch(conn->stmt);
        if(ret == SQL_NO_DATA) {
          ret = SQL_SUCCESS;
          strext_next = 0;
        } else if(SQL_SUCCEEDED(ret))
          SQLGetData(conn->stmt, 1, SQL_C_UBIGINT, &strext_next,
                     sizeof(strext_next), NULL);
      }
      db_next();
    }
    ret = db_prep(query2->str);
    if(SQL_SUCCEEDED(ret))
      ret = db_bind64(1, strext);
    if(SQL_SUCCEEDED(ret))
      ret = SQLExecute(conn->stmt);
    db_next();
    strext = strext_next;
  }
  g_string_free(query1, TRUE);
  g_string_free(query2, TRUE);
  db_commit();
}
```

Connecting to the database is then just a matter of providing the correct connection string. This can include user ID and password, so only one configuration parameter is needed.

120b ⟨*CGI Database Session Support Configuration* 120b⟩≡ (135d 149b) 130a▷

```
# ODBC Database connection string; blank to disable
# Format is generally DSN=<dsn>[; UID=<username>; PWD=<password>]
# Or apparently also FILEDSN=<filename>
#dbconnect =

⟨Session Value Support Configuration 117b⟩
```

121a ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁117c 123f▷

```
static SQLHENV henv = SQL_NULL_HENV;
static const char *dbconnect = NULL;
```

121b ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁120a 121d▷

```
void init_db_cgi_session(void)
{
  ⟨Set up database connection 121c⟩
}
```

121c ⟨*Set up database connection* 121c⟩≡ (121b) 130c▷

```
SQLRETURN ret;

dbconnect = getconf("dbconnect", NULL);
if(!dbconnect || !*dbconnect)
  return;
/* use connection pool */
ret = SQLSetEnvAttr(SQL_NULL_HANDLE, SQL_ATTR_CONNECTION_POOLING,
                                     (SQLPOINTER)SQL_CP_ONE_PER_DRIVER, 0);
⟨Ignore ODBC ret 123a⟩
ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
if(SQL_SUCCEEDED(ret))
  ret = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
                            (SQLPOINTER)SQL_OV_ODBC3, 0);
if(!SQL_SUCCEEDED(ret)) {
  ⟨Print ODBC env error 123b⟩
  exit(1);
}
id_field_type = getconf("id_field_type", id_field_type);
sessval_length = getconf_int("sessval_length", sessval_length);
strext_length = getconf_int("strext_length", strext_length);
```

121d ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁121b 122a▷

```
NEOERR *open_db(db_conn *conn)
{
  SQLRETURN ret;

  if(!dbconnect || !*dbconnect)
    return nerr_raise_msg("No database connection string specified");
  ret = SQLAllocHandle(SQL_HANDLE_DBC, henv, &conn->dbc);
  if(SQL_SUCCEEDED(ret))
    ret = SQLDriverConnect(conn->dbc, 0, (SQLCHAR *)dbconnect, SQL_NTS, NULL,
                           0, NULL, SQL_DRIVER_NOPROMPT);
  if(SQL_SUCCEEDED(ret)) {
    /* use explicit commit */
    ret = SQLSetConnectAttr(conn->dbc, SQL_AUTOCOMMIT,
                                       (SQLPOINTER)SQL_AUTOCOMMIT_OFF,
                                       SQL_IS_INTEGER);
    if(SQL_SUCCEEDED(ret))
      ret = SQLAllocHandle(SQL_HANDLE_STMT, conn->dbc, &conn->stmt);
  }
```

121d ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁121b 122a▷

```
  if(SQL_SUCCEEDED(ret))
    return STATUS_OK;
  ⟨Return converted ODBC error 123c⟩
}

void close_db(db_conn *conn)
{
  if(conn->stmt) {
    SQLCancel(conn->stmt);
    db_next();
    SQLFreeHandle(SQL_HANDLE_STMT, conn->stmt);
  }
  if(conn->dbc) {
    SQLEndTran(SQL_HANDLE_DBC, conn->dbc, SQL_ROLLBACK);
    SQLFreeHandle(SQL_HANDLE_DBC, conn->dbc);
  }
  memset(conn, 0, sizeof(*conn));
}
```

ODBC has its own error reporting mechanism. Errors are not accumulated between calls; instead they are stored in the various handles. Fortunately, there is no need to explicitly free them. A function is required to gather all the parts of the message together, though. Some versions of unixODBC crash when retrieving anything but the first message when used with sqliteodbc. Since I can't research what versions may cause problems, the loop is stopped after the first iteration with any unixODBC version.

122a ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁121d 123e▷

```
NEOERR *odbc_msg(SQLSMALLINT htype, SQLHANDLE h)
{
  SQLRETURN ret = SQL_SUCCESS;
  static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
  static SQLCHAR state[7];
  SQLINTEGER rc;
  static SQLCHAR msg[1024];
  SQLSMALLINT len;
  NEOERR *nerr = STATUS_OK;
  int i;

  pthread_mutex_lock(&lock);
  for(i = 1; SQL_SUCCEEDED(ret); i++) {
    ret = SQLGetDiagRec(htype, h, i, state, &rc, msg, sizeof(msg), &len);
    if(SQL_SUCCEEDED(ret))
      nerr = nerr == STATUS_OK ?
          nerr_raise(GENERIC_ERR, "ODBC[%d-%s]: %s", rc, state, msg) :
          nerr_pass_ctx(nerr, "ODBC[%d-%s]: %s", rc, state, msg);
#ifdef SQL_ATTR_UNIXODBC_VERSION
    break;
#endif
  }
  pthread_mutex_unlock(&lock);
  if(nerr == STATUS_OK)
    nerr = nerr_raise(GENERIC_ERR, "Unknown ODBC error");
  return nerr;
}
```

122b ⟨*Database Support Definitions* 114b⟩+≡ (113f) ◁117d

```
#define db_err(dbcerr) nerr_pass(odbc_msg(dbcerr ? SQL_HANDLE_DBC : \
                                                   SQL_HANDLE_STMT, \
                                          dbcerr ? conn->dbc : conn->stmt))
```

122c ⟨*(build.nw) C Prototypes* 12c⟩+≡ ◁115a

```
NEOERR *db_err(gboolean dbcerr);
```

123a ⟨*Ignore ODBC* `ret` 123a⟩≡ (121c)

```
ret = SQL_SUCCESS;
```

123b ⟨*Print ODBC env error* 123b⟩≡ (121c)

```
die_if_err(nerr_pass(odbc_msg(SQL_HANDLE_ENV, henv)));
```

123c ⟨*Return converted ODBC error* 123c⟩≡ (121d)

```
return db_err(!conn->stmt);
```

123d ⟨*Set* `nerr` *for ODBC error* 123d⟩≡ (117–19 124c)

```
nerr = db_err(dbcerr);
```

In addition to calling the initialization functions, the individual session operations must be defined and overridden. If they are already overridden, the session initialization does nothing.

123e ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁122a 123g▷

```
static NEOERR *db_create_session(cgi_state_t *cgi_state)
{
  NEOERR *nerr = STATUS_OK;
  ⟨Create session using database 124c⟩
  return nerr;
}

static NEOERR *db_read_session(const char *name, const char *child,
                               HDF *target, gboolean recursive,
                               cgi_state_t *cgi_state)
{
  NEOERR *nerr = STATUS_OK;
  ⟨Read session using database 125b⟩
  return nerr;
}

static NEOERR *db_update_session(const char *root_name, HDF *root,
                                 HDF **unique_root, cgi_state_t *cgi_state)
{
  NEOERR *nerr = STATUS_OK;
  ⟨Update session using database 126a⟩
  return nerr;
}
```

123f ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁121a 125a▷

```
static cgi_cleanup_cb prev_cleanup;
```

123g ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁123e 124a▷

```
NEOERR *prepare_db_session(cgi_state_t *cgi_state)
{
  NEOERR *nerr = STATUS_OK;
```

123g ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁123e 124a▷

```
  if(henv != SQL_NULL_HENV && !cgi_state->session_cb_data) {
    cgi_state->session_cb_data = calloc(1, sizeof(db_conn));
    if(!cgi_state->session_cb_data)
      return nerr_raise_errno(NERR_NOMEM, "Session support");
    nerr = open_db(cgi_state->session_cb_data);
    if(nerr == STATUS_OK) {
      cgi_state->create_session = db_create_session;
      cgi_state->read_session = db_read_session;
      cgi_state->update_session = db_update_session;
      prev_cleanup = cgi_state->cleanup;
      cgi_state->cleanup = finish_db_session;
    } else {
      close_db(cgi_state->session_cb_data);
      free(cgi_state->session_cb_data);
      cgi_state->session_cb_data = NULL;
    }
  }
  return nerr;
}
```

124a ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁123g 127b▷

```
void finish_db_session(cgi_state_t *cgi_state)
{
  close_db(cgi_state->session_cb_data);
  cgi_state->create_session = NULL;
  cgi_state->read_session = NULL;
  cgi_state->update_session = NULL;
  cgi_state->session_cb_data = NULL;
  if(prev_cleanup)
    (*prev_cleanup)(cgi_state);
}
```

A session is created by adding a new child of the root, and relying on the uniqueness constraint on the name to ensure that a unique name is generated. This is done by updating the name to a new random name until the uniqueness constraint has been satisfied. Session names are similar to the file mode: a date stamp followed by a few random printable characters. There is no need to put a great deal of effort into making this secure; the GLib thread-safe global random number generator is used for this.

124b ⟨*CGI Database Session Parameters* 124b⟩≡ (114b)

```
guint64 root_id;
```

124c ⟨*Create session using database* 124c⟩≡ (123e) 133a▷

```
char sessid[2+8+6+1];
guint64 newid;
db_conn *conn = cgi_state->session_cb_data;
SQLRETURN ret;
gboolean dbcerr = FALSE;

nerr_op(create_sessval(conn, 1, NULL, NULL, &newid));
if(nerr != STATUS_OK)
  return nerr;
while(1) {
  int i;
  char c;
  time_t now;
```

124c ⟨*Create session using database* 124c⟩≡ (123e) 133a▷

```
  time(&now);
  sprintf(sessid, "CS%08X", (int)now);
  for(i = 0; i < 6; i++) {
    do {
      c = g_random_int_range(48, 122);
    } while(!isalnum(c));
    sessid[i + 10] = c;
  }
  sessid[16] = 0;
  ret = db_prep("update sessval set name = ? where id = ?");
  if(!SQL_SUCCEEDED(ret))
    return db_err(FALSE);
  ret = db_binds(1, sessid);
  if(SQL_SUCCEEDED(ret))
    ret = db_bind64(2, newid);
  if(SQL_SUCCEEDED(ret)) {
    char code[7];
    ret = SQLExecute(conn->stmt);
    if(!SQL_SUCCEEDED(ret) &&
       SQLGetDiagField(SQL_HANDLE_STMT, conn->stmt, 1, SQL_DIAG_SQLSTATE,
                       code, sizeof(code), NULL) == SQL_SUCCESS &&
       !strcmp(code, "23000")) { /* 23000 == integrity constraint */
      db_next();
      continue;
    }
  }
  db_commit();
  if(!SQL_SUCCEEDED(ret)) {
    ⟨Set nerr for ODBC error 123d⟩
  }
  db_next();
  break;
}
conn->root_id = newid;
cgi_state->sessionid = g_strdup(sessid);
```

Rather than always using the sesion name to find an ID, the root is stored in the state as soon as it is known. If the session is newly created, it is known after creation. Otherwise, it needs to be queried.

125a ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁123f 126b▷

```
static NEOERR *set_root_id(db_conn *conn, const char *name)
{
  SQLRETURN ret;
  NEOERR *nerr;

  if(conn->root_id)
    return STATUS_OK;
  ret = db_prep("select id from sessval where name = ? and parent = 1");
  if(SQL_SUCCEEDED(ret))
    ret = db_binds(1, name);
  if(SQL_SUCCEEDED(ret))
    ret = SQLExecute(conn->stmt);
  if(SQL_SUCCEEDED(ret))
    ret = SQLFetch(conn->stmt);
  if(SQL_SUCCEEDED(ret))
    ret = SQLGetData(conn->stmt, 1, SQL_C_UBIGINT, &conn->root_id,
                     sizeof(conn->root_id), NULL);
  nerr = SQL_SUCCEEDED(ret) ? STATUS_OK : db_err(FALSE);
  db_next();
  return nerr;
}
```

125b ⟨*Read session using database* 125b⟩≡ (123e) 129▷

```
db_conn *conn = cgi_state->session_cb_data;
nerr = set_root_id(conn, cgi_state->sessionid);
if(nerr != STATUS_OK)
  return nerr;
```

126a ⟨*Update session using database* 126a⟩≡ (123e) 131a▷

```
db_conn *conn = cgi_state->session_cb_data;
nerr = set_root_id(conn, cgi_state->sessionid);
if(nerr != STATUS_OK) {
  if(!root)
    nerr_ignore(&nerr);
  return nerr;
}
```

Reading a value from the session requires following the hierarchical path. When searching for a single value, no recursion is necessary. If only a partial path is found, and the remander should still be created, `partial` can be non-NULL to point to what still needs to be created. Otherwise, a zero `id` indicates that it was not found.

126b ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁125a 127a▷

```
static NEOERR *db_nodeid(db_conn *conn, guint64 root, const char *name,
                         guint64 *id, const char **partial)
{
  char *s, *e;
  SQLRETURN ret;
  NEOERR *nerr = STATUS_OK;
  char *modname;

  *id = root;
  if(!name || !*name)
    return STATUS_OK;
  if(*name == '.')
    return nerr_raise_msg("Invalid node name");
  modname = s = strdup(name);
  if(!modname)
    return nerr_raise_msg_errno("No memory for scan");
  while(1) {
    e = strchr(s, '.');
    if(e)
      *e = 0;
    ret = db_prep("select id from sessval where parent = ? and name = ?");
    if(SQL_SUCCEEDED(ret))
      ret = db_bind64(1, *id);
    if(SQL_SUCCEEDED(ret))
      ret = db_binds(2, s);
    if(SQL_SUCCEEDED(ret))
      ret = SQLExecute(conn->stmt);
    if(SQL_SUCCEEDED(ret))
      ret = SQLFetch(conn->stmt);
    if(SQL_SUCCEEDED(ret))
      ret = SQLGetData(conn->stmt, 1, SQL_C_UBIGINT, id, sizeof(*id), NULL);
    else if(ret == SQL_NO_DATA) {
      db_next();
      if(partial)
        *partial = name + (int)(s - modname);
      else
        *id = 0;
      free(modname);
      return STATUS_OK;
    }
```

126b ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁125a 127a▷

```
    if(!SQL_SUCCEEDED(ret)) {
      nerr = db_err(FALSE);
      db_next();
      free(modname);
      return nerr;
    }
    db_next();
    if(!e)
      break;
    s = e + 1;
    if(!*s || *s == '.') {
      free(modname);
      return nerr_raise_msg("Invalid node name");
    }
  }
  free(modname);
  return STATUS_OK;
}
```

Immediate recursion is not possible when sharing a single statement handle, so recursion is done by following the HDF after reading. The HDF routines will be allocating strings and other information anyway, so only a single shared buffer is used to read the values. Each call reads only a single value and the names of all of the children. In order to emulate the non-recursive read mode, a counter is passed in to indicate the maximum recursion depth, if non-negative.

127a ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁126b 130b▷

```
static NEOERR *db_readtree(db_conn *conn, guint64 root, const char *name,
                           HDF *hdf, int recurse, GString **_buf)
{
  SQLRETURN ret;
  NEOERR *nerr = STATUS_OK;
  GString *buf = NULL;

  if(name) {
    nerr = db_nodeid(conn, root, name, &root, NULL);
    if(nerr != STATUS_OK || !root)
      return nerr;
  }
  if(!_buf) {
    _buf = &buf;
    buf = g_string_sized_new(80);
  }
  ⟨Set hdf's value from database 128a⟩
  if(nerr == STATUS_OK && recurse) {
    ⟨Insert children of hdf from database 128b⟩
  }
  if(buf)
    g_string_free(buf, TRUE);
  return nerr;
}
```

127b ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁124a 135e▷

```
SQLRETURN SQLGetGString(HSTMT *stmt, int pno, GString *buf)
{
  SQLLEN lenret;
  SQLRETURN ret;

  ret = SQLGetData(stmt, 1, SQL_C_CHAR, buf->str + buf->len,
                   buf->allocated_len - buf->len, &lenret);
  if(!SQL_SUCCEEDED(ret))
```

127b ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁124a 135e▷

```
    return ret;
  if(lenret != SQL_NULL_DATA) {
    if(lenret + buf->len > buf->allocated_len - 1) {
      int olen = buf->allocated_len - 1 - buf->len;
      g_string_set_size(buf, lenret + buf->len + 1);
      ret = SQLGetData(stmt, 1, SQL_C_CHAR, buf->str + olen,
                       buf->allocated_len - olen, NULL);
    }
    buf->len += lenret;
  }
  return SQL_SUCCESS;
}
```

128a ⟨*Set* `hdf`*'s value from database* 128a⟩≡ (127a)

```
ret = db_prep("select value, valext from sessval where id = ?");
if(SQL_SUCCEEDED(ret))
  ret = db_bind64(1, root);
if(SQL_SUCCEEDED(ret))
  ret = SQLExecute(conn->stmt);
if(SQL_SUCCEEDED(ret))
  ret = SQLFetch(conn->stmt);
if(SQL_SUCCEEDED(ret)) {
  guint64 extid;
  SQLLEN lenret;
  g_string_truncate(*_buf, 0);
  ret = SQLGetGString(conn->stmt, 1, *_buf);
  if(SQL_SUCCEEDED(ret)) {
    ret = SQLGetData(conn->stmt, 2, SQL_C_UBIGINT, &extid, sizeof(extid),
                     &lenret);
    while(SQL_SUCCEEDED(ret) && lenret != SQL_NULL_DATA) {
      db_next();
      ret = db_prep("select value, ext from sessvalext where id = ?");
      if(SQL_SUCCEEDED(ret))
        ret = db_bind64(1, extid);
      if(SQL_SUCCEEDED(ret))
        ret = SQLExecute(conn->stmt);
      if(SQL_SUCCEEDED(ret))
        ret = SQLFetch(conn->stmt);
      if(SQL_SUCCEEDED(ret))
        ret = SQLGetGString(conn->stmt, 1, *_buf);
      if(SQL_SUCCEEDED(ret))
        ret = SQLGetData(conn->stmt, 2, SQL_C_UBIGINT, &extid, sizeof(extid),
                         &lenret);
    }
  }
  if(!SQL_SUCCEEDED(ret))
    nerr = db_err(FALSE);
  else if((*_buf)->len)
    nerr = hdf_set_value(hdf, NULL, (*_buf)->str);
} else if(ret != SQL_NO_DATA)
  nerr = db_err(FALSE);
db_next();
```

128b ⟨*Insert children of* `hdf` *from database* 128b⟩≡ (127a)

```
HDF *c;

ret = db_prep("select name from sessval where parent = ?");
if(SQL_SUCCEEDED(ret))
  ret = db_bind64(1, root);
if(SQL_SUCCEEDED(ret))
  ret = SQLExecute(conn->stmt);
```

128b ⟨*Insert children of* `hdf` *from database* 128b⟩≡ (127a)

```
if(SQL_SUCCEEDED(ret)) {
  while(1) {
    ret = SQLFetch(conn->stmt);
    if(ret == SQL_NO_DATA) {
      ret = SQL_SUCCESS;
      break;
    }
    g_string_truncate(*_buf, 0);
    ret = SQLGetGString(conn->stmt, 1, *_buf);
    if(!SQL_SUCCEEDED(ret))
      break;
    nerr = hdf_get_node(hdf, (*_buf)->str, &c);
    if(nerr != STATUS_OK)
      break;
  }
}
if(!SQL_SUCCEEDED(ret))
  nerr = db_err(FALSE);
db_next();
if(nerr != STATUS_OK)
  return nerr;
for(c = hdf_obj_child(hdf); c; c = hdf_obj_next(c))
  db_readtree(conn, root, hdf_obj_name(c), c,
              recurse > 0 ? recurse - 1 : recurse, _buf);
```

Since the wrapper already clears the HDF as needed, the reader now only needs to call the recursive function to read the database.

129 ⟨*Read session using database* 125b⟩+≡ (123e) ◁125b 134a▷

```
guint64 root_id = conn->root_id;

if(child) {
  nerr = db_nodeid(conn, root_id, name, &root_id, NULL);
  if(nerr != STATUS_OK || !root_id)
    return nerr;
  name = child;
}
nerr = db_readtree(conn, root_id, name, target, recursive ? -1 : 1, NULL);
if(!name)
  cgi_state->session = target;
if(debug) {
  cgi_printf("<pre>Read Session %s:\n", name ? name : cgi_state->sessionid);
  STRING ps;
  string_init(&ps);
  nerr = hdf_dump_str(target, "", 2, &ps);
  if(nerr == STATUS_OK && ps.buf)
    cgi_puts_html_escape(cgi_state, ps.buf, FALSE);
  else
    nerr_ignore(&nerr);
  string_clear(&ps);
  cgi_puts("--------\n</pre>\n");
}
```

Updating consists both of inserting new values and removing old values. The simpler of the two is removal, so that is covered first. With cascade deletes, no recursion is necessary. Since cascade deletes cannot be guaranteed, the deletion function recurses by reading an array of child IDs from the database, and then deleting those before trying to delete the current node. It first changes the name of the node being operated on to avoid additional nodes being created mid-operation. This still isn't guaranteed to work; this code will always have a small race condition when deleting nodes. For a small potential relief of this, the node is renamed before deletion. Its ID could still already be read into another process, though, so this is not

a complete cure.

130a ⟨*CGI Database Session Support Configuration* 120b⟩+≡ (135d 149b) ◁120b

```
# Set to non-blank if the CGI session table supports cascading deletes
#dbcascade =
```

130b ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁127a 130d▷

```
static gboolean dbcascade = FALSE;
```

130c ⟨*Set up database connection* 121c⟩+≡ (121b) ◁121c

```
if(*getconf("dbcascade", ""))
  dbcascade = TRUE;
```

130d ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁130b 132a▷

```
static NEOERR *destroy_node(db_conn *conn, guint64 root_id)
{
  SQLRETURN ret;
  gboolean dbcerr = FALSE;
  gboolean did_delete;

  while(1) {
    if(!dbcascade) {
      /* rename to reduce chance of problems (ignore error) */
      char tname[22];
      sprintf(tname, ".%llu", (ullong)root_id);
      ret = db_prep("update sessval set name = ? where id = ?");
      if(SQL_SUCCEEDED(ret))
        ret = db_binds(1, tname);
      if(SQL_SUCCEEDED(ret))
        ret = db_bind64(2, root_id);
      if(SQL_SUCCEEDED(ret))
        ret = SQLExecute(conn->stmt);
      db_next()
    }
    ret = db_prep("delete from sessval where id = ?");
    if(SQL_SUCCEEDED(ret))
      ret = db_bind64(1, root_id);
    if(SQL_SUCCEEDED(ret))
      ret = SQLExecute(conn->stmt);
    if(SQL_SUCCEEDED(ret)) {
      db_next();
      if(dbcascade) {
        db_commit();
        return STATUS_OK;
      }
      did_delete = TRUE;
    }
    if(dbcascade) {
      NEOERR *nerr = db_err(FALSE);
      db_next();
      return nerr;
    }
    db_next();
    ret = db_prep("delete from sessvalext where parent = ?");
    if(SQL_SUCCEEDED(ret))
      ret = db_bind64(1, root_id);
    if(SQL_SUCCEEDED(ret))
      ret = SQLExecute(conn->stmt);
    db_next();
```

130d ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁130b 132a▷

```
    GArray *children = g_array_new(FALSE, FALSE, sizeof(guint64));
    ret = db_prep("select id from sessval where parent = ?");
    if(SQL_SUCCEEDED(ret))
      ret = db_bind64(1, root_id);
    if(SQL_SUCCEEDED(ret))
      ret = SQLExecute(conn->stmt);
    if(SQL_SUCCEEDED(ret)) {
      while(1) {
        guint64 cid;

        ret = SQLFetch(conn->stmt);
        if(ret == SQL_NO_DATA) {
          ret = SQL_SUCCESS;
          break;
        }
        if(!SQL_SUCCEEDED(ret))
          break;
        ret = SQLGetData(conn->stmt, 1, SQL_C_UBIGINT, &cid, sizeof(cid), NULL);
        if(!SQL_SUCCEEDED(ret))
          break;
        g_array_append_val(children, cid);
      }
      db_next();
    }
    while(children->len) {
      destroy_node(conn, g_array_index(children, guint64, children->len - 1));
      g_array_remove_index(children, children->len - 1);
    }
    g_array_free(children, TRUE);
    if(did_delete) {
      db_commit();
      return STATUS_OK;
    }
  }
}
```

131a ⟨*Update session using database* 126a⟩+≡ (123e) ◁126a 131b▷

```
if(!root) {
  guint64 root_id;

  nerr = db_nodeid(conn, conn->root_id, root_name, &root_id, NULL);
  if(nerr != STATUS_OK || !root_id)
    return nerr;
  return destroy_node(conn, root_id);
}
```

Inserting requires deleting first as well, but first the root node for insertion must be found and possibly created. This creates another race condition wherein other readers may try to read an incomplete entry. This race condition is alleviated by using a temporary name for the new node(s), and even parenting it to one, until it is ready to be linked into the whole. Any unneeded new nodes after all have been inserted can simply be deleted. As a shortcut, an attempt is made to directly update a node without children; if this fails, then the long-form insertion procedure is followed.

131b ⟨*Update session using database* 126a⟩+≡ (123e) ◁131a 132b▷

```
SQLRETURN ret;

if(!hdf_obj_child(root)) {
  guint64 udn;
  SQLLEN nrows;
  const char *val = hdf_obj_value(root);
```

131b ⟨*Update session using database* 126a⟩+≡ (123e) ◁131a 132b▷

```
  if(val && !*val)
    val = NULL;
  if((!val || strlen(val) <= sessval_length) &&
     nerr_op_ok(db_nodeid(conn, conn->root_id, root_name, &udn, NULL)) && udn) {
    ret = db_prep("delete from sessvalext where parent = ?");
    if(SQL_SUCCEEDED(ret))
      ret = db_bind64(1, udn);
    if(SQL_SUCCEEDED(ret))
      ret = SQLExecute(conn->stmt);
    db_next();
    if(SQL_SUCCEEDED(ret))
      ret = db_prep("update sessval set value = ? where id = ? and name = ?");
    if(SQL_SUCCEEDED(ret))
      ret = db_binds(1, val);
    if(SQL_SUCCEEDED(ret))
      ret = db_bind64(2, udn);
    if(SQL_SUCCEEDED(ret)) {
      val = strrchr(root_name, '.');
      if(!val)
        val = root_name;
      else
        val = val + 1;
      ret = db_binds(3, val);
    }
    if(SQL_SUCCEEDED(ret))
      ret = SQLExecute(conn->stmt);
    if(SQL_SUCCEEDED(ret) &&
       SQL_SUCCEEDED(SQLRowCount(conn->stmt, &nrows)) &&
       nrows == 1) {
      db_next();
      db_trans(SQL_COMMIT);
      return STATUS_OK;
    }
  }
}
```

The top node is the only one that is treated specially; all others can be inserted below it recursively without worrying about anything being yanked out in mid-stream. If the insertion point is the root node, a new session is being created, so there is also no need for any extra hoop jumping.

132a ⟨*CGI Database Session Variables* 114e⟩+≡ (113b) ◁130d

```
static NEOERR *insert_children(db_conn *conn, guint64 id, HDF *hdf)
{
  NEOERR *nerr = STATUS_OK;
  guint64 cid;

  for(hdf = hdf_obj_child(hdf); hdf; hdf = hdf_obj_next(hdf)) {
    nerr_op(create_sessval(conn, id, hdf_obj_name(hdf), hdf_obj_value(hdf),
                           &cid));
    if(nerr != STATUS_OK)
      return nerr;
    if(hdf_obj_child(hdf)) {
      nerr = insert_children(conn, cid, hdf);
      if(nerr != STATUS_OK)
        return nerr;
    }
  }
  return STATUS_OK;
}
```

132b ⟨*Update session using database* 126a⟩+≡ (123e) ◁131b 133b▷

```
guint64 inserted_vals;

if(root_name) {
  nerr_op(create_sessval(conn, 1, NULL, hdf_obj_value(root), &inserted_vals));
  if(nerr != STATUS_OK)
    return nerr;
} else
  inserted_vals = conn->root_id;
nerr = insert_children(conn, inserted_vals, root);
if(nerr != STATUS_OK) {
  NEOERR *nerr2 = destroy_node(conn, inserted_vals);
  nerr_ignore(&nerr2);
  return nerr;
}
if(!root_name)
  return nerr;
```

133a ⟨*Create session using database* 124c⟩+≡ (123e) ◁124c 134b▷

```
if(nerr == STATUS_OK)
  nerr = insert_children(conn, conn->root_id, cgi_state->session);
if(nerr != STATUS_OK) {
  NEOERR *nerr2 = destroy_node(conn, conn->root_id);
  nerr_ignore(&nerr2);
}
```

The actual insertion is done by scanning for the insertion point, and inserting. This leaves open a race where a hierarchy could be in the process of being deleted as it is scanned; this is bypassed by placing the scan in a loop and ensuring that the name does not change while it is being updated.

133b ⟨*Update session using database* 126a⟩+≡ (123e) ◁132b

```
GString *buf = g_string_sized_new(80);
while(1) {
  guint64 lastroot;
  const char *createstr, *s = NULL, *e; /* s init to shut gcc up */
  gboolean dbcerr = FALSE;

  nerr = db_nodeid(conn, conn->root_id, root_name, &lastroot, &createstr);
  if(nerr != STATUS_OK) {
    NEOERR *nerr2 = destroy_node(conn, inserted_vals);
    nerr_ignore(&nerr2);
    g_string_free(buf, TRUE);
    return nerr;
  }
  if(!createstr && !unique_root) {
    nerr = destroy_node(conn, lastroot);
    nerr_ignore(&nerr);
    continue;
  }
  if(createstr) {
    for(s = createstr; (e = strchr(s, '.')); s = e + 1) {
      g_string_truncate(buf, 0);
      g_string_append_len(buf, s, (int)(e - s));
      nerr_op(create_sessval(conn, lastroot, buf->str, NULL, &lastroot));
      if(nerr != STATUS_OK)
        break;
    }
    if(nerr == STATUS_OK && unique_root)
      nerr_op(create_sessval(conn, lastroot, s, NULL, &lastroot));
  }
  char unbuf[22];
```

133b ⟨*Update session using database* 126a⟩+≡ (123e) ◁132b

```
  if(nerr == STATUS_OK && unique_root) {
    sprintf(unbuf, "%llu", (ullong)inserted_vals);
    s = unbuf;
  }
  if(nerr != STATUS_OK) {
    nerr_ignore(&nerr);
    continue;
  }
  ret = db_prep("update sessval set parent = ? where id = ?");
  if(SQL_SUCCEEDED(ret))
    ret = db_bind64(1, lastroot);
  if(SQL_SUCCEEDED(ret))
    ret = db_bind64(2, inserted_vals);
  if(SQL_SUCCEEDED(ret))
    ret = SQLExecute(conn->stmt);
  db_next();
  if(SQL_SUCCEEDED(ret))
    ret = db_prep("update sessval set name = ? where id = ?");
  if(SQL_SUCCEEDED(ret))
    ret = db_binds(1, s);
  if(SQL_SUCCEEDED(ret))
    ret = db_bind64(2, inserted_vals);
  if(SQL_SUCCEEDED(ret))
    ret = SQLExecute(conn->stmt);
  db_next();
  if(SQL_SUCCEEDED(ret)) {
    db_commit();
    return STATUS_OK;
  }
  SQLEndTran(SQL_HANDLE_DBC, conn->dbc, SQL_ROLLBACK);
  sleep(1);
}
```

Like almost everything else, cleaning up old sessions is much harder than with the simple files. The expiration date needs to be stored somewhere, for one. The value for the session root is a convenient, currently unused spot. The database itself cannot be relied upon to do proper date translation, so all operations with this date stamp must use the raw number of seconds since the UNIX epoch. Since the root session is always read before updating, only root reads and creates will update this timestamp.

134a ⟨*Read session using database* 125b⟩+≡ (123e) ◁129

```
if(!name && nerr == STATUS_OK) {
  SQLRETURN ret;
  ⟨Update root timestamp in database 134c⟩
}
```

134b ⟨*Create session using database* 124c⟩+≡ (123e) ◁133a

```
⟨Update root timestamp in database 134c⟩
```

134c ⟨*Update root timestamp in database* 134c⟩≡ (134)

```
guint64 tval;
time_t now;

time(&now);
tval = (guint64)now;
ret = db_prep("update sessval set value = ? where id = ?");
if(SQL_SUCCEEDED(ret))
  ret = db_bind64(1, tval);
if(SQL_SUCCEEDED(ret))
```

134c ⟨*Update root timestamp in database* 134c⟩≡ (134)

```
  ret = db_bind64(2, conn->root_id);
if(SQL_SUCCEEDED(ret))
  ret = SQLExecute(conn->stmt);
db_next();
db_trans(SQL_COMMIT);
```

To clean up then, is to search for root nodes with an old timestamp. Root nodes without any timestamp may be left-over garbage as well, so if these are found two scans in a row, they are removed as well. It is still possible that two scans complete before a temporary node is used correctly, but it is sufficiently unlikely that no additional effort will be made to correct this race condition.

135a ⟨*(build.nw) C Executables* 105a⟩+≡ ◁105a

```
delete_old_db_sessions \
```

135b ⟨*delete_old_db_sessions.c* 135b⟩≡

```
⟨(build.nw) Common C Header (imported)⟩

const char *config_root = "/etc";
const char *config_path = "db_session.conf";

⟨(cs-glib.nw) Help Function (imported)⟩

SQLHENV henv;
const char *dbconnect;

int main(int argc, const char **argv)
{
  ⟨(cs-glib.nw) Common Mainline Variables (imported)⟩

  ⟨(cs-glib.nw) Read configuration (imported)⟩
  clean_db_sessions();
  return 0;
}
```

135c ⟨`delete_old_db_sessions` *Description* 135c⟩≡

```
Keep session table clean
```

135d ⟨`delete_old_db_sessions` *Configuration* 135d⟩≡

```
⟨CGI Database Session Support Configuration 120b⟩
# How long to wait before next expired session check (seconds)
#clean_interval=60

# How long to wait before expiring sessions (seconds)
#expire_time=1200
```

135e ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁127b

```
void clean_db_sessions(void)
{
  db_conn _conn, *conn;
  NEOERR *nerr;
  SQLRETURN ret;
  int interval = getconf_int("clean_interval", 60);
  int expire_time = getconf_int("expire_time", 20*60);
  GArray *null_ids = NULL, *del_ids = NULL;
```

135e ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁127b

```
if(expire_time < 0)
  expire_time = 0;
if(interval > expire_time)
  interval = expire_time;
if(interval < 5)
  interval = 5;
init_db_cgi_session();
if(henv == SQL_NULL_HENV)
  die_msg("Specify a database connection string");
conn = &_conn;
memset(conn, 0, sizeof(*conn));
die_if_err(open_db(conn));
GString *query = g_string_new("");
g_string_printf(query, "%s from sessval " \
                    "where parent = 1 and id <> 1 and value is not null " \
                      "and cast(value as %s) < ?",
                dbcascade ? "delete" : "select id", id_field_type);
while(1) {
  time_t now;
  guint64 exp_ts;

  time(&now);
  exp_ts = now - expire_time;
  ret = db_prep(query->str);
  if(SQL_SUCCEEDED(ret))
    ret = db_bind64(1, exp_ts);
  if(SQL_SUCCEEDED(ret))
    SQLExecute(conn->stmt);
  if(!dbcascade && SQL_SUCCEEDED(ret)) {
    guint64 id;

    while(SQL_SUCCEEDED(SQLFetch(conn->stmt))) {
      if(!SQL_SUCCEEDED(SQLGetData(conn->stmt, 1, SQL_C_UBIGINT, &id,
                                   sizeof(id), NULL)))
        continue;
      if(!del_ids)
        del_ids = g_array_new(FALSE, FALSE, sizeof(guint64));
      g_array_append_val(del_ids, id);
    }
    db_next();
    if(del_ids)
      while(del_ids->len) {
        id = g_array_index(del_ids, guint64, del_ids->len - 1);
        g_array_remove_index(del_ids, del_ids->len - 1);
        nerr = destroy_node(conn, id);
        nerr_ignore(&nerr);
      }
  } else
    db_next();
  ret = db_exec("select id from sessval where parent = 1 and value is null");
  if(SQL_SUCCEEDED(ret)) {
    GArray *prev_null_ids = null_ids;
    guint64 id;
    null_ids = NULL;
    while(SQL_SUCCEEDED(SQLFetch(conn->stmt))) {
      if(!SQL_SUCCEEDED(SQLGetData(conn->stmt, 1, SQL_C_UBIGINT, &id,
                                   sizeof(id), NULL)))
        continue;
      if(!null_ids)
        null_ids = g_array_new(FALSE, FALSE, sizeof(guint64));
      g_array_append_val(null_ids, id);
    }
    db_next();
    if(prev_null_ids && null_ids) {
```

135e ⟨*CGI Database Session Functions* 117e⟩+≡ (113b) ◁127b

```
        int i, j;

        for(i = 0; i < null_ids->len; i++) {
          id = g_array_index(null_ids, guint64, i);
          for(j = 0; j < prev_null_ids->len; j++)
            if(id == g_array_index(prev_null_ids, guint64, j)) {
              nerr = destroy_node(conn, id);
              g_array_remove_index_fast(prev_null_ids, j);
              if(nerr == STATUS_OK) {
                g_array_remove_index_fast(null_ids, i);
                i--;
              }
              nerr_ignore(&nerr);
              break;
            }
        }
      }
      if(prev_null_ids)
        g_array_free(prev_null_ids, TRUE);
    } else
      db_next();
    sleep(interval);
  }
}
```

Chapter 11

CAS Authentication

The Central Authentication Service (CAS) is in use at Indiana Univeristy for university-wide login. CAS uses a single web server for authentication of other web sites. IU's CAS conforms mostly to version 1 of the protocol[1]; the only known issue is that the `service` parameter cannot be URL-encoded. The IU server also takes an additional parameter (*cas_svc*) to indicate the authentication realm.

There is an Apache module for CAS authentication, but it shares many of the limitations of CAS itself. The primary limitation is that CAS does not support POST parameters. If authentcation fails during a POST request, all parameters are lost. The CAS module is also marked as deprecated on the official CAS site.

Instead, code provided here will save the CGI parameters in the session state and restore them when CAS returns. This still loses uploads, as saving the uploaded file in the session state would only work for very small file uploads. Also, clients that issue requests other than GET, HEAD, and POST may be confused by the redirection, so these are not supported, either. However, it does help the most common case. In addition, it is possible to query the status of CAS authentication without forcing the authentication to take place, so that can be used for the unsupported methods and uploads before simply rejecting the request.

138a ⟨*CAS Support Configuration* 138a⟩≡ (149b) 143a▷

```
# The CAS server URL; must end in a slash
# CAS is disabled if this is blank.
#cas_url =

# The CAS service type
# This is not used if blank
#cas_svc =
```

138b ⟨*cas.c* 138b⟩≡

```
⟨(build.nw) Common C Header (imported)⟩

static const char *cas_url;
static const char *cas_svc;

⟨CAS Support Variables 140c⟩
⟨CAS Support Functions 139b⟩

void init_cas(void)
{
  cas_url = getconf("cas_url", NULL);
  if(cas_url && !*cas_url)
    cas_url = NULL;
  if(cas_url) {
```

[1] http://www.jasig.org/cas/protocol

138b ⟨*cas.c* 138b⟩≡

```
    cas_svc = getconf("cas_svc", NULL);
    if(cas_svc && !*cas_svc)
      cas_svc = NULL;
    ⟨Initialize CAS globals 141d⟩
  }
  init_cgi_session();
}

NEOERR *check_cas(cgi_state_t *cgi_state, gboolean *needs_redirect)
{
  NEOERR *nerr = STATUS_OK;
  *needs_redirect = FALSE;
  ⟨Check CAS credentials and return STATUS_OK if OK 145b⟩
  return nerr;
}

NEOERR *cas_redirect(cgi_state_t *cgi_state)
{
  NEOERR *nerr = STATUS_OK;
  ⟨Redirect to CAS 141a⟩
  return nerr;
}

NEOERR *update_cas(cgi_state_t *cgi_state, gboolean *did_redirect)
{
  NEOERR *nerr = check_cas(cgi_state, did_redirect);

  if(nerr == STATUS_OK && *did_redirect)
    nerr = cas_redirect(cgi_state);
  return nerr;
}
```

139a ⟨*(cs-glib.nw) Library* cs-supt *Members* 8a⟩+≡ ◁113a

```
cas.o
```

To obtain CAS credentials, the CGI application must first obtain a ticket by redirecting the browser to the CAS site, which will then redirect back to the CGI application when credentials are obtained. This cannot be done using `wget`, `curl` or something like that from the CGI application, because it relies on cookies in the user's browser to support persistent logins. It also presents a unified login user interface when persistent credentials are not already present. To perform the redirect, the CGI application's URL must be passed to CAS. This is computed using the CGI environment.

139b ⟨*CAS Support Functions* 139b⟩≡ (138b) 144b▷

```
char *compute_myurl(cgi_state_t *cgi_state)
{
  GString *buf;
  ⟨Compute my URL into buf 139c⟩
  char *ret = buf->str;
  g_string_free(buf, FALSE);
  if(debug) {
    cgi_puts("My URL: ");
    cgi_puts_html_escape(cgi_state, ret, FALSE);
    cgi_puts("<br>\n");
  }
  return ret;
}
```

139c ⟨*Compute my URL into* `buf` 139c⟩≡ (139b) 140a▷

```
buf = g_string_new(cgi_env_val("CGI.HTTPS", NULL) ? "https" : "http");
g_string_append(buf, "://");
g_string_append(buf, cgi_header_val("Host", "localhost"));
g_string_append(buf, cgi_env_val("CGI.RequestURI", "/"));
```

Since the parameters are saved in the session, the URL passed to CAS has all parameters stripped off.

140a ⟨*Compute my URL into* `buf` 139c⟩+≡ (139b) ◁139c 140b▷

```
char *s = strchr(buf->str, '?');
if(s) {
  *s = 0;
  buf->len = (int)(s - buf->str);
}
```

This is replaced by a pointer to the saved parameters. Parameters are saved in the session under `cas_parms`, under a unqiue parent name. This is only done if there are any parameters to save, so the existence of the CAS ticket is presumed to mean that this is a return from a parameterless run.

140b ⟨*Compute my URL into* `buf` 139c⟩+≡ (139b) ◁140a 141c▷

```
const char *pno = cgi_parm_val("cas.pno", NULL);

if(pno) {
  g_string_append(buf, "?cas.pno=");
  g_string_append(buf, pno);
} else if(hdf_obj_child(cgi_state->cgi_parms) &&
          !cgi_parm_val("casticket", NULL)) {
  HDF *pno_obj;
  NEOERR *nerr = update_session_state("cas_parms", cgi_state->cgi_parms,
                                      &pno_obj, cgi_state);
  if(nerr == STATUS_OK) {
    g_string_append(buf, "?cas.pno=");
    g_string_append(buf, hdf_obj_name(pno_obj));
    hdf_destroy(&pno_obj);
  } else if(debug)
    die_if_err(nerr);
  else {
    nerr_ignore(&nerr);
    g_string_free(buf, TRUE);
    return NULL;
  }
}
```

Some requests take parameters from the headers, as well. In particular, conditional requests and range requests are signaled by headers. In order to support these, known relevant headers are stored in the CGI parameters. The request method itself is relevant as well, so it is also stored in the CGI parameters. This means that no like-named CGI parameters may exist, and that the parameter restoration must store the parameters back into the appropriate header HDF parameters. For this purpose, the `_HTTP_` hierarchy is reserved.

140c ⟨*CAS Support Variables* 140c⟩≡ (138b) 143b▷

```
const char * const save_headers[] = {
  ⟨HTTP Headers to Save on CAS Redirect 140d⟩
  NULL
};
```

140d ⟨*HTTP Headers to Save on CAS Redirect* 140d⟩≡ (140c)

```
"If",
"IfMatch",
"IfModifiedSince",
"IfNoneMatch",
"IfRange",
"IfUnmodifiedSince",
"Range",
```

141a ⟨*Redirect to CAS* 141a⟩≡ (138b) 141b▷

```
HDF *http_headers = cgi_env_obj("HTTP");
HDF *cgi_headers = NULL;
if(http_headers) {
  const char * const * h, *v;

  for(h = save_headers; *h; h++)
    if((v = hdf_get_value(http_headers, *h, NULL))) {
      if(!cgi_headers)
        nerr_op(hdf_get_node(cgi_state->cgi_parms, "_HTTP_", &cgi_headers));
      nerr_op(hdf_set_value(cgi_headers, *h, v));
    }
}
if(nerr != STATUS_OK)
  return nerr;
```

Also, if the request method is not GET or POST, the request method itself is a parameter which is lost. Normally, the request will simply be dropped, but for this library, the method will be saved just like the HTTP headers.

141b ⟨*Redirect to CAS* 141a⟩+≡ (138b) ◁141a 142a▷

```
if(cgi_state->req_method != HTTP_REQ_GET &&
   cgi_state->req_method != HTTP_REQ_POST &&
   !nerr_op_ok(cgi_parm_set("_HTTP_._Request",
                            cgi_env_val("CGI.RequestMethod", NULL))))
  return nerr;
```

In addition to the pointer to the parameters, the session information itself must be given.

141c ⟨*Compute my URL into* `buf` 139c⟩+≡ (139b) ◁140b

```
g_string_append_session_cgi_parms(cgi_state, !pno, buf, TRUE);
```

File uploads are lost in the process, unless their state is adequately saved in the standard parameter hierarchy. For standard uploads, this can be accomplished by setting `Config.Upload.Unlink` to zero.

141d ⟨*Initialize CAS globals* 141d⟩≡ (138b) 143c▷

```
hdf_set_int_value(local_config, "Config.Upload.Unlink", 0);
```

This will leave a lot of temporary files lying around, though, so a directory cleaner similar to the state cleaner is recommended (maybe even the state cleaner itself). In addition, the per-run cleanup routine can scan for files and remove them.

141e ⟨*Perform cleanups requiring valid CGI parameters* 141e⟩≡ (21b)

```
if(!hdf_get_int_value(cgi_state->hdf, "Config.Upload.Unlink", 1)) {
  HDF *parm, *subparm;
  const char *fn;
  for(parm = hdf_obj_child(cgi_state->cgi_parms); parm;
                                                  parm = hdf_obj_next(parm)) {
    for(subparm = hdf_obj_child(parm); subparm; subparm = hdf_obj_next(subparm))
      if((fn = hdf_get_value(subparm, "FileName", NULL)))
        unlink(fn);
    if((fn = hdf_get_value(parm, "FileName", NULL)))
      unlink(fn);
  }
}
```

The redirect itself is done using the CGI kit redirection built-in. The return code indicates that the CGI application should do no rendering of its own.

142a ⟨*Redirect to CAS* 141a⟩+≡ (138b) ◁141b

```
/* note: IU CAS does not expect/support URL-encoding */
char *myurl = compute_myurl(cgi_state);
#if 0
if(myurl) {
  char *s;
  cgi_url_escape(myurl, &s);
  free(myurl);
  myurl = s;
}
#endif
if(!myurl)
  return nerr_raise_msg_errno("Can't compute return URL");
cgi_status(302);
print_session_cookie(cgi_state, FALSE);
if(cas_svc)
  cgi_redirect_uri(cgi_state->cgi, "%slogin?cassvc=%s&casurl=%s",
                   cas_url, cas_svc, myurl);
else
  cgi_redirect_uri(cgi_state->cgi, "%slogin?casurl=%s", cas_url, myurl);
free(myurl);
```

When the redirection returns, the `casticket` parameter is added by CAS to indicate at least partially successful login. It is then up to the CGI application to validate that ticket and turn it into a user ID, by yet again calling the CAS server. This time, though, the user's browser does not need to be involved. The ticket itself is sent to the CAS server, which will stamp the ticket as invalid, and reply with text indicating whether or not it was valid before stamping.

To perform that callback, libcurl[2] is used. Since it uses SSL, a certificate authority might also need to be specified. Note that the SSL libary used by curl should probably match any other SSL libraries against which the program is linked; otherwise the program may freeze on initial contact.

142b ⟨*(build.nw) Common C Includes* 8b⟩+≡ ◁114a

```
#include <curl/curl.h>
```

142c ⟨*(build.nw) makefile.vars* 7g⟩+≡ ◁113e

```
EXTRA_CFLAGS += $(shell curl-config --cflags)
EXTRA_LDFLAGS += $(shell curl-config --libs)
```

[2] From cURL, `http://curl.haxx.se`

143a ⟨*CAS Support Configuration* 138a⟩+≡ (149b) ◁138a 145c▷

```
# The SSL Certificate Authority file to use with CAS
# If blank, use a reasonable default system CA file
#cas_ca =
```

143b ⟨*CAS Support Variables* 140c⟩+≡ (138b) ◁140c 146a▷

```
static const char *cas_ca = NULL;
```

143c ⟨*Initialize CAS globals* 141d⟩+≡ (138b) ◁141d 146b▷

```
cas_ca = getconf("cas_ca", NULL);
curl_global_init(CURL_GLOBAL_ALL);
```

Setup for validation starts the same way as setup for redirection: a suitable URL is built. The only difference is that the operation changes (from login to validate) and the CAS ticket (`cas_ticket`) is appended.

143d ⟨*Validate CAS ticket with cURL* 143d⟩≡ (146d) 143e▷

```
GString *val_url = g_string_new(cas_url);
GString *val_ret = NULL;
g_string_append(val_url, "validate?");
if(cas_svc) {
  g_string_append(val_url, "cassvc=");
  g_string_append(val_url, cas_svc);
  g_string_append_c(val_url, '&');
}
g_string_append(val_url, "casticket=");
/* IU CAS doesn't support URL-encoding */
g_string_append/*_uri_escaped*/(val_url, cas_ticket);
g_string_append(val_url, "&casurl=");
char *myurl = compute_myurl(cgi_state);
g_string_append/*_uri_escaped*/(val_url, myurl);
free(myurl);
if(debug) {
  cgi_puts("Validating with ");
  cgi_puts_html_escape(cgi_state, val_url->str, FALSE);
  cgi_puts("<br>\n");
}
```

Then, the cURL setup itself is done. The results will be retrieved into memory, specifically into a `GString` buffer.

143e ⟨*Validate CAS ticket with cURL* 143d⟩+≡ (146d) ◁143d

```
CURL *valhandle = curl_easy_init();
if(valhandle) {
  curl_easy_setopt(valhandle, CURLOPT_URL, val_url->str);
  if(cas_ca)
    curl_easy_setopt(valhandle, CURLOPT_CAINFO, cas_ca);
  curl_easy_setopt(valhandle, CURLOPT_SSL_VERIFYPEER, 1);
  ⟨Finish setting up cURL validation 144a⟩
  if(!curl_easy_perform(valhandle) && val_ret && val_ret->len) {
    if(debug) {
      cgi_puts("validation result:\n");
      cgi_puts_html_escape(cgi_state, val_ret->str, FALSE);
      cgi_puts("<br>\n");
    }
    ⟨Check cURL validation results 145a⟩
  }
```

143e ⟨*Validate CAS ticket with cURL* 143d⟩+≡ (146d) ◁143d

```
  curl_easy_cleanup(valhandle);
}
g_string_free(val_url, TRUE);
```

Retrieval into a buffer requires the use of a write callback. Just in case, the progress indication is turned off as well. That way, no output will be generated on the standard streams.

144a ⟨*Finish setting up cURL validation* 144a⟩≡ (143e) 144c▷

```
curl_easy_setopt(valhandle, CURLOPT_WRITEFUNCTION, (curl_write_callback)write_gstr);
curl_easy_setopt(valhandle, CURLOPT_WRITEDATA, &val_ret);
curl_easy_setopt(valhandle, CURLOPT_NOPROGRESS, 1);
```

144b ⟨*CAS Support Functions* 139b⟩+≡ (138b) ◁139b 144d▷

```
static ssize_t write_gstr(void *ptr, size_t size, size_t nmemb, void *_str)
{
  GString **str = _str;
  size *= nmemb;
  if(!size)
    return 0;
  if(*str)
    g_string_append_len(*str, ptr, size);
  else
    *str = g_string_new_len(ptr, size);
  return nmemb;
}
```

If debugging is enabled, the CURL debug output should be printed to the CGI stream. This requires setting up another callback.

144c ⟨*Finish setting up cURL validation* 144a⟩+≡ (143e) ◁144a

```
if(debug) {
  curl_easy_setopt(valhandle, CURLOPT_VERBOSE, 1);
  curl_easy_setopt(valhandle, CURLOPT_DEBUGFUNCTION, cgi_curl_debug);
  curl_easy_setopt(valhandle, CURLOPT_DEBUGDATA, cgi_state);
}
```

144d ⟨*CAS Support Functions* 139b⟩+≡ (138b) ◁144b 147b▷

```
int cgi_curl_debug(CURL *curl, curl_infotype it, char *buf, size_t len,
                   void *cbdata)
{
  cgi_state_t *cgi_state = cbdata;
  const char *its;

  switch(it) {
    case CURLINFO_TEXT:
    default:
      its = "info";
      break;
    case CURLINFO_HEADER_IN:
      its = "head";
      break;
    case CURLINFO_HEADER_OUT:
      its = "hout";
      break;
    case CURLINFO_DATA_IN:
      its = "rdat";
      break;
```

144d ⟨*CAS Support Functions* 139b⟩+≡ (138b) ◁144b 147b▷

```
    case CURLINFO_DATA_OUT:
      its = "wdat";
      break;
  }
  cgi_printf("curl%s: %.*s<br>\n", its, (int)len, buf);
  return 0;
}
```

The buffer contents from the validation should be two lines of plain text. The first line is either “yes” or “no”, indicating whether or not the ticket was valid. The second line is either blank or the name of the user authenticated by the ticket. Once the “yes” has been confirmed, and the user ID has been verified to be plain text, the user ID is copied into the CGI state. That’s all there is to CAS authentication, at least version one as implemented at IU.

145a ⟨*Check cURL validation results* 145a⟩≡ (143e)

```
char *s = val_ret->str;
/* first line == yes means valid */
if(!memcmp(s, "yes\r\n", 5)) {
  /* second line == user ID */
  s += 5;
  int i;
  char name_valid = 1;
  for(i = 0 ; s[i] && s[i] != '\r' && s[i] != '\n'; i++)
    if(!isalnum(s[i])) {
      name_valid = 0;
      break;
    }
  s[i] = 0;
  if(name_valid)
    cas_userid = s;
}
```

Before any of this can begin, though, the CGI environment must be parsed. This is done using the session initializer. The full session support is only necessary to support CAS, so if CAS is disabled, the session support could be disabled as well, but it is intended that the CAS calls completely replace the session calls, so the session call is always used. Nonetheless, if the CAS URL isn’t specified, then nothing else needs to be done.

145b ⟨*Check CAS credentials and return* `STATUS_OK` *if OK* 145b⟩≡ (138b) 146c▷

```
nerr = update_cgi_session(cgi_state);
if(nerr != STATUS_OK || !cgi_state->sessionid)
  return nerr;
if(!cas_url)
  return STATUS_OK;
nerr = read_session_state("cas", NULL, FALSE, cgi_state);
nerr_ignore(&nerr);
```

This is a rather expensive process, so the credentials are retained for a while before doing another redirect. The credentials are saved in the session information (`cas.userid`). Even though the process is expensive, eventually the credentials should be re-verified. This is done on a configurable interval. When the user ID is saved in the session, the current date plus the expiration interval is stored with it (`cas.exp`).

145c ⟨*CAS Support Configuration* 138a⟩+≡ (149b) ◁143a

```
# The maximum seconds between ticket renewals
# Note that multiple hosts verifying the same ticket must have synchronized
# clocks
# Setting this too short (e.g. 0) will force continual renewals
#cas_renew_interval = 600
```

146a ⟨*CAS Support Variables* 140c⟩+≡ (138b) ◁143b

```
static guint cas_renew_interval;
```

146b ⟨*Initialize CAS globals* 141d⟩+≡ (138b) ◁143c

```
cas_renew_interval = getconf_int("cas_renew_interval", 600);
```

146c ⟨*Check CAS credentials and return* `STATUS_OK` *if OK* 145b⟩+≡ (138b) ◁145b 146d▷

```
const char *cas_userid = cgi_session_val("cas.userid", NULL);
if(cas_userid && cgi_state->userid && !strcmp(cgi_state->userid, cas_userid)) {
  time_t expiration = cgi_session_val_int("cas.exp", -1);
  if(expiration >= 0) {
    time_t now;
    time(&now);
    if(now < expiration)
        return STATUS_OK;
  }
}
```

146d ⟨*Check CAS credentials and return* `STATUS_OK` *if OK* 145b⟩+≡ (138b) ◁146c 147a▷

```
const char *cas_ticket = cgi_parm_val("casticket", NULL);
if(cas_ticket) {
  ⟨Validate CAS ticket with cURL 143d⟩
  if(cas_userid) {
    ⟨Update session user from CAS user 146e⟩
  }
  if(val_ret)
    g_string_free(val_ret, TRUE);
}
```

In addition to setting the user information, a successful CAS logon must also restore any saved CGI parameters. After it does so, it removes them from the session state.

146e ⟨*Update session user from CAS user* 146e⟩≡ (146d)

```
nerr_op(cgi_session_set("cas.userid", cas_userid));
if(cas_renew_interval) {
  char nbuf[22];
  time_t now;
  time(&now);
  now += cas_renew_interval;
  snprintf(nbuf, sizeof(nbuf), "%llu", (ullong)now);
  cgi_session_set("cas.exp", nbuf);
}
nerr_op(cgi_session_set("Session.RemoteUser", cas_userid));
nerr_op(update_session_state("Session.RemoteUser",
                             cgi_session_obj("Session.RemoteUser"), NULL,
                             cgi_state));
nerr_op(cgi_env_set("CGI.RemoteUser", cas_userid));
if(nerr == STATUS_OK)
  cas_userid = cgi_state->userid = cgi_env_val("CGI.RemoteUser", NULL);
```

146e ⟨*Update session user from CAS user* 146e⟩≡ (146d)

```
else
  cas_userid = NULL; /* may as well try again; no memory this time */
const char *pno = cgi_parm_val("cas.pno", NULL);
if(pno && nerr == STATUS_OK)
  nerr = read_session_state("cas_parms", pno, TRUE, cgi_state);
if(pno && nerr == STATUS_OK) {
  HDF *parms = cgi_session_obj("cas_parms"), *obj = NULL;
  if(parms)
    obj = hdf_get_obj(parms, pno);
  if(obj) {
    HDF *head = hdf_get_obj(obj, "_HTTP_");
    if(head) {
      const char *req;
      nerr_op(hdf_copy(cgi_state->hdf, "HTTP", obj));
      if((req = cgi_env_val("HTTP._Request", NULL))) {
        nerr_op(cgi_env_set("CGI.RequestMethod", req));
        cgi_state->req_method = http_req_id(req, strlen(req));
      }
      hdf_remove_tree(obj, "_HTTP_");
    }
    hdf_remove_tree(cgi_state->hdf, "Query");
    nerr_op(hdf_copy(cgi_state->hdf, "Query", obj));
    if(nerr != STATUS_OK) {
      nerr_ignore(&nerr);
      hdf_remove_tree(cgi_state->hdf, "Query");
    }
    nerr = update_session_state("cas_parms", NULL, &obj, cgi_state);
    if(debug) {
      cgi_puts("Restored ");
      cgi_dump_hdf(cgi_state);
    }
  }
}
nerr_op(update_session_state("cas", cgi_session_obj("cas"), NULL, cgi_state));
nerr_ignore(&nerr);
```

Finally, if and only if any of that failed, a new redirect must be initiated. On the other hand, if it succeeded, the standard remote user parameter is updated.

147a ⟨*Check CAS credentials and return* `STATUS_OK` *if OK* 145b⟩+≡ (138b) ◁146d

```
if(cas_userid)
  return STATUS_OK;
*needs_redirect = TRUE;
return STATUS_OK;
```

One other operation that has been neglected here is the logout operation. For this, the user is once again redirected to CAS. First, though, the session must be destroyed.

147b ⟨*CAS Support Functions* 139b⟩+≡ (138b) ◁144d

```
gboolean logout_cas(cgi_state_t *cgi_state)
{
  if(cas_url) {
    cgi_status(302);
    logout_cgi_session(cgi_state, TRUE);
    cgi_redirect_uri(cgi_state->cgi, "%slogout", cas_url);
    return TRUE;
  }
  return FALSE;
}
```

Chapter 12

Sample Program

A very small test program is provided here to exercise the code a little bit. It uses a template for the body, but a hard-coded header.

148a ⟨*(build.nw) C Test Support Executables* 148a⟩≡

```
test.cgi \
```

148b ⟨*test.cgi.c* 148b⟩≡

```
⟨(build.nw) Common C Header (imported)⟩

⟨CGI Message Overrides 16a⟩

const char *tmpl;

static NEOERR *do_cgi(cgi_state_t *cgi_state)
{
  ⟨CGI Per-Run Variables 17b⟩

  die_if_err(init_cgi_run(cgi_state));
  die_if_err(prepare_db_session(cgi_state));
  gboolean did_redirect;
  nerr = update_cas(cgi_state, &did_redirect);
  if(nerr != STATUS_OK || did_redirect)
    return nerr;
  if(cgi_parm_val("action.logout", NULL)) {
    if(logout_cas(cgi_state))
      return STATUS_OK; /* redirects */
    cgi_status(200);
    cgi_puts("Status: 200\n");
    logout_cgi_session(cgi_state, TRUE);
    cgi_puts("Content-type: text/plain\n\nGood bye.");
    return STATUS_OK;
  }
  cgi_status(200);
  cgi_puts("Status: 200\n");
  print_session_cookie(cgi_state, FALSE);
  set_session_tmpl_vars(cgi_state, FALSE, FALSE);
  cgi_puts("Content-type: text/html\n\n");
  nerr = tmpl ? tmpl_file_to_cgi(tmpl, cgi_state, FALSE) :
                tmpl_to_cgi("<?cs include:\"html_prefix.cs\"?>"
                            "<?cs include:\"html_macros.cs\"?>"
                            "</head><body>\n"
                            "Hello, "
                            "<?cs alt:CGI.RemoteUser?>stranger<?cs /alt ?>"
                            "<br/>\n"
```

148b ⟨*test.cgi.c* 148b⟩≡

```
                                "<?cs call:form(\"f\", 0) ?>\n"
                                "<?cs var:CGI.SessionForm ?>"
                                "<input type=\"submit\" value=\"Press Me\"><br/>\n"
                                "<input type=\"submit\" name=\"action.logout\""
                                       "value=\"Log Out\"><br/>\n"
                                "</form></body></html>\n", cgi_state, FALSE);
  return nerr;
}

const char *config_root = ".";
const char *config_path = "test.cgi.conf";

⟨(cs-glib.nw) Help Function (imported)⟩

int main(int argc, const char **argv)
{
  ⟨CGI Mainline Variables 17a⟩

  g_thread_init(NULL);
  ⟨(cs-glib.nw) Read configuration (imported)⟩
  ⟨(cs-glib.nw) Set up templates (imported)⟩
  tmpl = getconf("tmpl", NULL);
  init_db_cgi_session();
  init_cas();
  run_cgi(do_cgi);
  return 0;
}
```

149a ⟨`test.cgi` *Description* 149a⟩≡ (149d)

```
Simple CGI test program
```

149b ⟨`test.cgi` *Configuration* 149b⟩≡ (149d)

```
# The template file to display (default: built-in test template)
#tmpl =

⟨(cs-glib.nw) Template Parameter Configuration (imported)⟩
⟨CGI Support Configuration 7e⟩
⟨CGI Session Support Configuration 98e⟩
⟨CGI Database Session Support Configuration 120b⟩
⟨CAS Support Configuration 138a⟩
```

149c ⟨*(build.nw) Plain Files* 7h⟩+≡ ◁116a

```
test.cgi.conf \
```

149d ⟨*test.cgi.conf* 149d⟩≡

```
# Sample test.cgi configuration file

# ⟨test.cgi Description 149a⟩
test.cgi {
  ⟨test.cgi Configuration 149b⟩
}
```

In addition, here is a simple shell script which can take a body plus headers and pass it to a CGI program.

149e ⟨*(build.nw) Script Executables* 149e⟩≡

```
pass-http \
```

150a ⟨*pass-http* 150a⟩≡

```
#!/bin/sh

export SERVER_NAME=whatever
export SERVER_SOFTWARE=whatever
export GATEWAY_INTERFACE=CGI/1.1
export SERVER_PORT=80
read -r REQUEST_METHOD REQUEST_URI SERVER_PROTOCOL
echo "${REQUEST_METHOD} ${REQUEST_URI} ${SERVER_PROTOCOL}" >&2
PATH_INFO=${REQUEST_URI#/cgi-bin/*/}
if [ "x$PATH_INFO" = "x$REQUEST_URI" ]; then
  SCRIPT_NAME=
else
  SCRIPT_NAME="${REQUEST_URI#/cgi-bin/}"
  SCRIPT_NAME="${SCRIPT_NAME%%/*}"
  SCRIPT_NAME=/cgi-bin/"${SCRIPT_NAME}"
  PATH_INFO="${REQUEST_URI#${SCRIPT_NAME}}"
fi
# no attempt made to do ISINDEX
QUERY_STRING="${PATH_INFO#*\?}"
test "x$QUERY_STRING" = "x$PATH_INFO" && QUERY_STRING=
PATH_INFO="${PATH_INFO%%\?*}"
export REQUEST_METHOD REQUEST_URI PATH_INFO SERVER_PROTOCOL QUERY_STRING SCRIPT_NAME
export REMOTE_HOST=localhost
export REMOTE_ADDR=127.0.0.1
export REMOTE_USER=$(id -un)
lh=
while IFS= read -r h; do
  printf %s\\n "$h" >&2
  h="${h%
  test -z "$h" && break
  if [ -n "$lh" -a "x${h# }" != "x$h" ]; then
    eval "$lh=\"\$$lh\$h\""
  else
    lh="${h%%:*}"
    h="${h#*:}"
    while [ "x${h# }" != "x$h" ]; do
      h="${h# }"
    done
    lh="$(echo $lh | tr '[a-z]-' '[A-Z]_')"
    case $lh in
      CONTENT_LENGTH) h=0 ;; # unknown
      CONTENT_TYPE) ;;
      *) lh=HTTP_$lh ;;
    esac
    eval $lh=\"\$h\"
    export $lh
  fi
done
trap "rm /tmp/body.$$" 0
cat >/tmp/body.$$
export CONTENT_LENGTH=$(stat -c %s /tmp/body.$$)
echo Content-Length: ${CONTENT_LENGTH} >&2
test ${CONTENT_LENGTH} -eq 0 && unset CONTENT_LENGTH
"$@" < /tmp/body.$$
```

This can be used to execute the test CGI with an almost valid HTTP request:

150b ⟨*(build.nw) Test Scripts* 150b⟩≡

```
test-cgi \
```

150c ⟨*test-cgi* 150c⟩≡

```
#!/bin/sh

grep '^<<test-http-.*>>=$' *.nw | \
  sed 's/\([^:]*\):[^<]*<<\(.*\)>>=$/\1 \2/' | \
    sort -u | while read f cn; do
      echo
      echo "*** $cn ***"
      echo
      notangle -R"$cn" "$f" | ./pass-http ./test.cgi debug=1 cgi_session_dir=
done
```

151a ⟨*test-http-get* 151a⟩≡

```
GET /?some_parameter=y HTTP/1.1
```

151b ⟨*(build.nw) makefile.rules* 25a⟩+≡ ◁25a

```
test-cgi: test.cgi pass-http
```

Appendix A

Usage

Many of the features of this document can be used independently of the build system, but it is intended that this document be depended on by anything that uses it.

To use the HTML macros, include `html_macros.cs` into a ClearSilver file. It does not output any text. The macros are:

form(*fname*, *hasfiles*) simply displays the form start tag for a form of the given name. The content type is set to multipart, as required for file uploads, if *hasfiles* is true.

text(*varn*, *default*) displays a text entry box for the given CGI parameter name, initialized to the CGI parameter's current value if present or *default* if not.

rtext(*varn*,*seq*,*default*) displays a text entry box for the given CGI parameter name, initialized to the CGI parameter's current *seq*th value if present, or *default* if not. The *seq*th value is determined by looking at the children of *varn*, starting at 0. This is how ClearSilver usually encodes multiple values of the same name.

hidden(*varn*) displays hidden form entries for all values of *varn*. This may display nothing if *varn* has no value.

file(*varn*) displays a consistently sized file entry box for *varn*.

radio(*varn*, *value*, *default*) displays a radio box for the CGI parameter *varn*, whose value when checked will be *value*. The entry will be checked if *default* is true and no value exists for *varn*; otherwise it will be checked if *varn* currently has any value equal to *value*.

checkbox(*varn*) displays a checkbox for the CGI parameter *varn*, whose returned value will be Y if checked. The box will be checked if *varn* exists and is non-blank.

date(*varn*, *default*) displays a date entry box for the CGI parameter *varn*. The default value is set as with text boxes. In fact, this is just a text box whose class is `datesel`, on the assumption that JavaScript will be used to convert this into something fancier.

submit(*action*) displays a submission button whose name is action.*action* and whose value (i.e. label) is retrieved from the HDF parameter labels.*action*. It is assumed that the CGI will use the button name rather than its value.

optdiv_onload(), optdiv_start(), optdiv_opt(*label*, *div*), optdiv_end() See below.

print_session_parms() displays the session CGI parameters as hidden form entries; see discussion of sessions below.

add_session_parms(*first*) displays the session CGI parameters in a form suitable for adding to a URL string, using a question mark or ampersand to separate from the URL depending on *first*. See discussion of sessions below.

print_session_cookies() displays the HTTP header required to set session cookies; see discussion of sessions below.

A facility is provided for displaying HTML one division at a time, like tabs. A selection widget is used like a drop-down menu to select which division to show. This requires JavaScript; if JavaScript is not available, all divisions will be shown in the order they appear in the HTML. The JavaScript file, `html_prefix.js`, must be available on the web server. The web server relative path is defined in the configuration parameter `server_script_path`, which defaults to `/js`. The JavaScript file is included in all HTML by extending `html_prefix.cs`.

To use the division selector, add a macro call `optdiv_onload()` to the code executed on body load (e.g. in the onload parameter of the body tag). Then, in the main HTML, wherever the selection widget is to appear, define this widget using the macro `optdiv_start()`, followed by, for each division, `optdiv_opt(label, div)` with the text to show in the selector and the division number to show. The divisions are named div1, div2, etc. and the option value is the number only (i.e. 1, 2, etc.). Finally, close out the selection widget with a call to `optdiv_end()`. Naturally, the HTML to be shown or hidden must then be in divisions with the appropriate id values.

FastCGI support is provided by a few functions, and revolves around the `cgi_state` variable passed into those functions. The standard mainline variables have this added if you use ⟨*CGI Mainline Variables* 17a⟩ instead. The callback gets this passed in as its only parameter. Some useful entries in this structure are:

- `HDF *hdf` is the global parameters with the CGI parameters overlaid on top.
- `HDF *cgi_parms` is a quick reference to the CGI parameters only.
- `HDF *cgi_cookies` is a quick reference to the cookies only.
- `HDF *headers` is a quick reference to the HTTP headers.
- `const char *userid` is a quick reference to the known user name. This is NULL if the name is unknown.
- `const char *method` is a quick reference to the request method.
- `const char *body_enc` is a quick reference to the body encoding type. If the cooked input functions are used, this can be safely ignored.
- `gint64 body_len` is a quick reference to the body's content-length. If the length is unknown, it is set to -1.
- `const char *body_type` is a quick reference to the body's content-type.
- `HDF *session` contains any session values current read in. It is also a quick reference to `hdf`'s Session parameter.
- `cgi_upload_cb upload_cb` is a set of callbacks for upload control. This structure contains `open`, `write`, `progress`, and `close` callbacks, along with user `data`. Any may be left NULL to use the default action. Each callback has the HDF specifying the parameter currently being uploaded (with its value indicating the file name, and its children indicating some header fields, such as `Length`, `ModTime`, `CreationTime`, `AccessTime`, and `Type`) as its first parameter, and the `cgi_state` as the second parameter. They all return a ClearSilver error structure. The `open` callback takes no additional parameters, and is expected to open the file and store its descriptor in `data`. It can use any CGI parameters already known from `cgi_parms`. The `write` callback takes a `guint64` offset,

a buffer pointer, and an `int` length for the data to write. All data must be written before returning success. The `progress` callback is a simplified version of the `write` callback which only takes the offset and length parameters; this is the equivalent of the standard callback. The `close` callback is called when all is done, and has only the errors returned by previous phases as its additional parameter, so that it knows whether or not it should delete the final result.

- `cgi_cleanup_cb cleanup` is a function to call when the current thread is complete; it returns nothing and takes the `cgi_state` as its only parameter. Users of this are expected to implement manual chaining by storing the previous value and calling it if non-NULL.
- `create_session`, `session_cg_data`, `read_session`, and `update_session` are overrides for session management; see the main document for details on how these work.
- `xmlDocPtr xmlbody` is a parsed XML body, if there was one and the request method was not one of GET, HEAD, POST, or PUT.
- `xmlNsPtr dav_ns` is a pointer to a namespace named `DAV:`, if a node was processed which uses that namespace. It can be used with `check_ns` to check an XML node's namespace.

The CGI state's HDF can be printed when debugging using:

```
void cgi_dump_hdf(cgi_state_t *cgi_state)
```

Reading the various parameters from `cgi_state` is made easier with macros:

```
const char *cgi_env_val(const char *name,
                        const char *def)
```

```
gint64 cgi_env_val_int(const char *name,
                       gint64 def)
```

```
NEOERR *cgi_env_set(const char *name,
                    const char *value)
```

```
NEOERR *cgi_env_set_int(const char *name,
                        gint64 value)
```

```
HDF *cgi_env_obj(const char *name)
```

These operate on the `hdf` entry. They read character and integer values, set character and integer values, and return the object pointer, respectively.

```
const char *cgi_header_val(const char *name,
                           const char *def_val)
```

```
gint64 cgi_header_val_int(const char *name,
                          gint64 def_val)
```

```
NEOERR *cgi_header_set(const char *name,
                       const char *val)
```

```
NEOERR *cgi_header_set_int(const char *name,
                           gint64 val)
```

```
HDF *cgi_header_obj(const char *name)
```

These operate on the `headers` entry. They read character and integer values and set character and integer values, and return the object pointer, respectively. There is no hierarchy in the headers initially, but code may add children to help with parsing, so the `cgi_header_obj` function may still be useful.

```
const char *cgi_parm_val(const char *name,
                         const char *def)
```

```
gint64 cgi_parm_val_int(const char *name,
                        gint64 def)
```

```
NEOERR *cgi_parm_set(const char *name,
                     const char *value)
```

```
NEOERR *cgi_parm_set_int(const char *name,
                         gint64 value)
```

```
HDF *cgi_parm_obj(const char *name)
```

These operate on the `cgi_parms` entry. They read character and integer values, set character and integer values, and return the object pointer, respectively.

```
void for_each_cgi_parm(const char *name,
                       for_each_cb f,
                       ...)
```

```
void for_each_cgi_parm_obj(HDF *obj,
                           for_each_cb f,
                           ...)
```

These iterate on all values of a single or multi-valued CGI parameter, calling the callback for each object. The first takes the CGI parameter name, and the second takes the root object. The callback takes the object as its first argument, and further macro arguments as its remaining arguments.

```
const char *cgi_cookie_val(const char *name,
                           const char *def)
```

```
gint64 cgi_cookie_val_int(const char *name,
                          gint64 def)
```

```
NEOERR *cgi_cookie_set(const char *name,
                       const char *value)
```

```
NEOERR *cgi_cookie_set_int(const char *name,
                           gint64 value)
```

```
HDF *cgi_cookie_obj(const char *name)
```

These operate on the `cgi_cookies` entry. They read character and integer values, set character and integer values, and return the object pointer, respectively.

```
const char *cgi_session_val(const char *name,
                            const char *def_val)
```

```
gint64 cgi_session_val_int(const char *name,
                           gint64 def_val)
```

```
NEOERR *cgi_session_set(const char *name,
                        const char *value)
```

```
NEOERR *cgi_session_set_int(const char *name,
                            gint64 value)
```

```
HDF *cgi_session_obj(const char *name)
```

These operate on the `session` entry. They read character and integer values, set character and integer values, and return the object pointer, respectively.

Printing output and retrieving environment and PUT data is done via macros which use `cgi_state`:

```
int cgi_puts(const char *str)
```

This is the `fputs` equivalent.

```
int cgi_putc(char c)
```

This is the `fputc` equivalent.

```
int cgi_write(const char *buf,
              int length)
```

This is the `fwrite` equivalent.

```
int cgi_printf(const char *format,
               ...)
```

This is the `fprintf` equivalent.

```
int cgi_vprintf(const char *fmt,
                va_list parms)
```

This is the `vfprintf` equivalent.

```
void cgi_status(int stat)
```

This should be called when printing the Status header, in addition to actually printing that header.

```
int cgi_raw_gets(char *buf,
                 int length)
```

This is the `fgets` equivalent. Do not use this; use `cgi_gets` instead.

```
int cgi_raw_read(char *buf,
                 int length)
```

This is the `fread` equivalent, with an object size of one. Only use this if you want to bypass automatic decoding. Otherwise, use `cgi_read`.

```
const char *cgi_getenv(const char *name)
```

This is the `getenv` equivalent.

For encoded input streams, decoding versions of the input functions are also provided; since their side effects are more complex, they return a NEOERR rather than a simple flag:

```
NEOERR *cgi_decode_gets(cgi_state_t *cgi_state,
                        char *buf,
                        int length)
```

```
NEOERR *cgi_gets(char *buf,
                 int length)
```

These are the `fgets` equivalent.

```
NEOERR *cgi_decode_read(cgi_state_t *cgi_state,
                        char *buf,
                        int length,
                        int *retlength)
```

```
NEOERR *cgi_read(char *buf,
                 int length,
                 int *retlength)
```

These are the `fread` equivalent, with an object size of one.

In addition, adding the ⟨*CGI Message Overrides* 16a⟩ chunk to any C file overrides the `die_if_err` and other `die_` macros to display their output using the above macros instead of printing to standard error. They do not, however, prevent exiting the entire FastCGI program rather than just the current thread.

Since most output is for HTML, a few HTML-related escape functions are provided as well. The HTML may be a template, so CGI-equivalents of the template output function are also provided.

```
NEOERR *cgi_puts_url_escape(cgi_state_t *cgi_state,
                            const char *s)
```

Prints `s` using the standard URL-encoding. Unlike the standard's recommendation, space is encoded using the plus sign rather than `%20`.

```
NEOERR *cgi_puts_html_escape(cgi_state_t *cgi_state,
                             const char *s,
                             gboolean is_js)
```

Prints `s` using HTML-encoding. Backslashes and single quotes are also backslash-escaped if the `is_js` flag is TRUE. Any specials other than less-than, greater-than, ampersand, and quote are quoted using numeric escapes.

```
GString *g_string_append_html_escaped(GString *buf,
                                      const char *s,
                                      gboolean is_js)
```

Escapes `s` into `buf` using HTML-encoding. See `cgi_puts_html_escape` for details.

```
NEOERR *tmpl_to_cgi(const char *tmpl,
                    cgi_state_t *cgi_state,
                    gboolean raw)
```

This is the equivalent of `tmpl_string`, using the default CGI parameters overlayed on the global parameters.

```
NEOERR *tmpl_file_to_cgi(const char *tmpl,
                         cgi_state_t *cgi_state,
                         gboolean raw)
```

This is the equivalent of `tmpl_file`, using the default CGI parameters overlayed on the global parameters.

```
NEOERR *tmpl_to_cgi_hdf(const char *tmpl,
                        HDF *parms,
                        cgi_state_t *cgi_state,
                        gboolean raw)
```

This is the equivalent of `tmpl_string`, except that rather than using the default CGI parameters, a different set of parameters can be provided.

```
NEOERR *tmpl_file_to_cgi_hdf(const char *tmpl,
                             HDF *parms,
                             cgi_state_t *cgi_state,
                             gboolean raw)
```

This is the equivalent of `tmpl_file`, except that rather than using the default CGI parameters, a different set of parameters can be provided.

```
NEOERR *cgi_output_json(cgi_state_t *cgi_state,
                        HDF *hdf)
```

Prints the given `hdf` parameter and its children in JSON format.

```
NEOERR *normalize_path_obj(HDF *obj,
                           cgi_state_t *cgi_state,
                           const char *relative_to,
                           gboolean utf_8,
                           gboolean strip_cgi_script)
```

Finally, since HTTP requests and CGI parameters deal with paths, a function is provided to normalize those paths. It can optionally normalize UTF-8 paths, can deal with absolute or potentially relative (non-NULL `relative_to`) paths, and optionally strip the CGI path to produce something similar to `PATH_INFO`.

```
void init_cgi(void)
```

```
void run_cgi(cgi_callback_t callback)
```

```
NEOERR *init_cgi_run(cgi_state_t *cgi_state)
```

```
gboolean cgi_fork(cgi_state_t *cgi_state)
```

The basic structure of a FastCGI program consists of a mainline which does global setup and calls `init_cgi`, and a callback function which is called in a new thread for every FastCGI connection. The callback function is invoked via `run_cgi`, which takes a function pointer which returns an error pointer and takes only `cgi_state` as its parameter. The first function to call in the callback is `init_cgi_run`, which copies the global configuration and merges the CGI parameters. If a process is needed instead of a thread, the thread can call `cgi_fork` as its first operation and execute the rest of its code if the return value was TRUE. If the program is run as a standard CGI program, the callback is not called in the main thread instead of a new one. To distinguish the two, the global variable `gboolean is_cgi` can be used. If the `run_cgi` callback returns an error, the error is printed. How this is printed is controlled by variables in the CGI state's hdf: `header_printed` (non-blank to indicate HTTP header has already been printed), `error_status` (set to override HTTP status), and `error_tmpl` (set to a template to print instead of the default; the error text is in `error_text`).

The upload enhancements include the callback mechanism described above, and the fact that it supports any content encodings now. XML uploads are automatically processed and WebDAV requests are checked. It is enabled by default. The callbacks should be set in the global initialization. Further checks in the XML can take advantage of the namespace checker function.

```
gboolean check_ns(xmlNodePtr node,
                  xml_name_t ns,
                  xmlNsPtr *nscache)
```

XML uploads are automatically parsed and WebDAV requests are checked as much as possible. Further checks in the XML can take advantage of the namespace checker function, which caches the last namespace to possibly avoid doing a string comparison every time. Also, for convenience, the name codes for the DAV and CALDAV namespaces are macros named `DAV_NS` and `CALDAV_NS`, respectively.

All of the above configuration options can be included in an application's configuration file using ⟨*CGI Support Configuration* 7e⟩:

```
# Location of JavaScript on the server
#server_script_path = /js

# Maximum number of FastCGI threads
#max_cgi_threads = 64

# Set to non-blank to override message to print on errors
# This is evaluated as a ClearSilver template in the CGI environment; the
# text of the error message is in the error_text variable.
# The default message prints:
#    <html><body><h3>An error occurred while processing your request:</h3><pre>
#    <?cs var:error_text ?></pre></body></html>
#error_tmpl = <html><body><h3>An error occurred while processing ...

# Set to non-blank to override the status to return on errors
#error_status = 500 Internal Server Error

# Set to non-blank to disable printing the HTTP header for error messages
# This is usually set by any template which prints an HTTP header.
#header_printed =

# If set, limit XML request bodies (not PUT or POST) to this size, in bytes
#max_xml_body = 100000
```

```
void init_cgi_session(void)
```

```
NEOERR *update_cgi_session(cgi_state_t *cgi_state)
```

The session support is enabled by calling `init_cgi_session` instead of `init_cgi`. In the callback, `update_cgi_session` is called. It can also be called in upload callbacks; calling it multiple times will not affect operation. If it is called in an upload callback, the form which generated the upload should be sure to set session hidden variables before the file upload widget. This initializes a basic session with the following parameters:

Session.RemoteAddress The remote user's advertised IP address, or 127.0.0.1 if none is advertised.

Session.UserAgent The remote user's advertised user agent, or telnet if none is advertised.

Session.RemoteUser The remote user's identity, if known, or blank otherwise.

```
NEOERR *update_session_state(const char *root_name,
                             HDF *root,
                             HDF **unique_root,
                             cgi_state_t *cgi_state)
```

Any additional parameters can be set in `cgi_state->session` and then updated in the persistent state using `update_session_state`. This function takes the name of the root of the session parameters to update, the session parameter root object pointer (e.g. `cgi_session_obj(parm)`), a uniqueness pointer, and the `cgi_state`. The uniqueness pointer can be set to point to an unitialized HDF object pointer, in which case instead of updating the named parameter itself, a child will be added to the named parameter and its values will be the values passed in. The child is guaranteed to have a unqiue name among all of the children of that named node; the child object is returned in the uniqueness pointer so that its name, among other things, can be queried.

```
NEOERR *read_session_state(const char *root_name,
                           const char *child_name,
                           gboolean recursive,
                           cgi_state_t *cgi_state)
```

Since only the required session parameters are read from persistent state, any new parameters must also be explicitly read using `read_session_state`. This normally only obtains the named parameter, so if it has children, the `recursive` flag must be set. The name of the parameter is passed in as two parts, either of which may be NULL, to make hierarchical parameter retrieval easier.

These sessions are stored in files; this requires configuration from ⟨*CGI Session Support Configuration* 98e⟩ to configure a directory with `cgi_session_dir` (`/var/cache/cgi-sessions` by default). They are touched on every access, so a program can expire old sessions by examining the modification time. One such program is `delete_old_files`, which takes the session directory, expiration time in seconds, and optionally the time to next scan for expired sessions, in seconds (by default only scan once).

```
# The default directory where local sessions are stored
# Clearing this parameter disables local file session support
#cgi_session_dir = /var/cache/cgi-sessions
```

Keeping track of the session on the client side is done using cookies and/or CGI parameters. In general, the functions to print CGI parameters will print nothing if cookies are detected, unless their `force` parameter is set to TRUE. Likewise, if a cookie is already set, there is no need to set it again, so the cookie printers will also print nothing if a cookie is detected. They will also print nothing if a CGI parameter is set indicating that cookies were not available at some point, so there is no need to keep trying. Either way, a `force` parameter overrides this logic. The functions are:

```
NEOERR *print_session_cgi_parms(cgi_state_t *cgi_state,
                                gboolean force)
```

Prints CGI parameters as hidden inputs.

```
void g_string_append_session_cgi_parms(cgi_state_t *cgi_state,
                                       gboolean first,
                                       GString *buf,
                                       gboolean force)
```

Tacks CGI parameters to a string as if it were a URL being built. The first parameter generates a question mark if `first` is TRUE, or an ampersand otherwise.

```
NEOERR *print_session_cookie(cgi_state_t *cgi_state,
                             gboolean force)
```

Prints the HTTP header required to set the cookie.

```
NEOERR *set_session_tmpl_vars(cgi_state_t *cgi_state,
                              gboolean force_cookie,
                              gboolean force_form)
```

Sets `CGI.SessionParms` and `CGI.SessionCookies` so that their values can be used in ClearSilver templates to print the cookie header and/or hidden form entries using `print_session_cookies`, `print_session_parms`, and `add_session_parms`.

```
void logout_cgi_session(cgi_state_t *cgi_state,
                        gboolean print_cookies)
```

If the client expects to log out, the session can be destroyed using `logout_cgi_session`, which destroys the session and optionally prints the HTTP header required to clear the cookies.

```
void init_db_cgi_session(void)
```

```
NEOERR *prepare_db_session(cgi_state_t *cgi_state)
```

To use the database sessions instead of files, create the table using `sessdbcreate.sql` (possibly edited), and set up an ODBC driver. Use ⟨*CGI Database Session Support Configuration* 120b⟩ to add the configuration parameters for the connection string and cascading delete capability to the configuration file (`dbconnect` and `dbcascade`, which default to blank). Call `init_db_cgi_session` instead of `init_cgi_session`, and call `prepare_db_session` before `update_cgi_session`. This will set up the session callback overrides, so no other code can use these without saving and manual chaining. While the database routines could be used for other purposes as well, this session support is not intended for that. The database environment is not exported. Also, if `dbconnect` is blank, the database functions will be disabled and sessions will revert to files. To guarantee this will not happen, manually check that the parameter is not blank or missing. The database equivalent of `delete_old_files` is `delete_old_db_sessions`, which takes the connection string and timeout parameters in ClearSilver form rather than as basic parameters:

```
# ODBC Database connection string; blank to disable
# Format is generally DSN=<dsn>[; UID=<username>; PWD=<password>]
# Or apparently also FILEDSN=<filename>
#dbconnect =

# The SQL data type used for database table keys
#id_field_type = bigint

# The maximum length of a session value before extension
# 0 means there is no maximum
#sessval_length = 126

# The maximum length of an extension before further extension
# 0 means there is no maximum
#strext_length = 32768

# Set to non-blank if the CGI session table supports cascading deletes
#dbcascade =

# How long to wait before next expired session check (seconds)
#clean_interval=60

# How long to wait before expiring sessions (seconds)
#expire_time=1200
```

To use the database connection for other purposes, the database connection can be retrieved from `cgi_state->session_cb_data` as a `db_conn` pointer after `preapare_db_session`. The following macros can then be used to execute SQL, assuming that `conn` contains a pointer to the database connection:

```
SQLRETURN db_exec(const char *sql)
```

Execute SQL directly.

```
SQLRETURN db_prep(const char *sql)
```

Prepare SQL for binding and later execution.

```
SQLRETURN db_binds(int param,
                   const char *str)
```

Bind a string to a parameter position. If the string pointer is NULL, the value will be bound to SQL null.

```
SQLRETURN db_bind64(int param,
                    guint64 &val)
```

Bind a 64-bit integer to a parameter position.

```
SQLRETURN db_bindsl(int param,
                    const char *str,
                    SQLLEN len)
```

Bind a string of a given length to a parameter position. The length must be a variable which will not change until the binding is no longer needed.

```
SQLRETURN db_bindnull(int param)
```

Bind SQL null to a parameter position.

```
SQLRETURN db_next(void)
```

Prepare the shared statement handle for a new SQL statement.

```
SQLRETURN db_trans(SQLSMALLINT type)
```

Execute a commit (`SQL_COMMIT`) or rollback (`SQL_ROLLBACK`) on the current transaction.

```
void db_commit(void)
```

If the current `ret` return code is a successful SQLRETURN value, commit the current transaction. Also, set the `dbcerr` local variable to `TRUE`; this is to distinguish error types when interpreting ODBC error codes.

```
void db_commit_keeperr(void)
```

Commit the current transaction, regardless of the value of `ret`, but update `ret` to an error (and `dbcerr` to `TRUE`) if it was a successful SQLRETURN value.

```
NEOERR *db_err(gboolean dbcerr)
```

Convert the current SQL error stack for the connection to a NEOERR. The `dbcerr` switch selects which handle to use; this is usually `FALSE` (i.e., the statement handle), but is `TRUE` for connection-level operations, such as commits.

The following support functions are provided as well; any other operations should be done using direct ODBC calls with `conn->stmt` and `conn->dbc`.

```
NEOERR *odbc_msg(SQLSMALLINT htype,
                 SQLHANDLE h)
```

Convert all pending ODBC errors on a handle to a NEOERR. This takes the handle (env, stmt or dbc) and the handle type (`SQL_HANDLE_ENV`, `SQL_HANDLE_STMT` or `SQL_HANDLE_DBC`).

```
SQLRETURN SQLGetGString(HSTMT *stmt,
                        int pno,
                        GString *buf)
```

Retrieve a character type result into a dynamic string. The result is appended, so truncation may be necessary before calling.

```
void init_cas(void)
```

```
NEOERR *check_cas(cgi_state_t *cgi_state,
                  gboolean *needs_redirect)
```

```
NEOERR *cas_redirect(cgi_state_t *cgi_state)
```

```
NEOERR *update_cas(cgi_state_t *cgi_state,
                   gboolean *did_redirect)
```

Finally, to use CAS with sessions, add ⟨*CAS Support Configuration* 138a⟩ to the configuration file, which allows setting `cas_url` and `cas_svc` to enable CAS. Call `init_cas` in the mainline, and call `update_cas` in the callback instead of `update_cgi_session`. If it is desirable to check if valid CAS credentials exist without forcing a CAS redirect if they don't, call `check_cas` instead. In either case, the redirection variable is updated depending on whether or not a redirect was required. For `check_cas`, it is expected that the caller will eventually run `cas_redirect` to perform the actual redirect, and for `update_cas`, the caller is expected to exit if a redirect occurred. Either may update session parameters if CAS authentication succeeds.

```
# The CAS server URL; must end in a slash
# CAS is disabled if this is blank.
#cas_url =

# The CAS service type
# This is not used if blank
#cas_svc =

# The SSL Certificate Authority file to use with CAS
# If blank, use a reasonable default system CA file
#cas_ca =

# The maximum seconds between ticket renewals
# Note that multiple hosts verifying the same ticket must have synchronized
# clocks
# Setting this too short (e.g. 0) will force continual renewals
#cas_renew_interval = 600
```

The CAS redirection saves the current set of CGI parameters, a few HTTP headers, and the request method in the session state for restoration after CAS returns. This means that new CGI parameters can be introduced into the HDF (e.g. using `cgi_parm_set`) before calling `cas_redirect` or `update_cas`,

and they can be removed afterwards. These parameters would then reappear only on return from CAS by `update_cas` or `check_cas`. For example, this could be used to save HTTP headers not normally saved, or to set a flag indicating that an upload was aborted due to CAS redirection. Since saving HTTP headers is done anyway, they can also be saved by being added as comma-terminated strings to ⟨*HTTP Headers to Save on CAS Redirect* 140d⟩, or by storing the header normally stored at `HTTP.`*parameter* in the CGI parameter `_HTTP_.`*parameter*. Note that this also means that the `_HTTP_` parameter and the hierarchy underneath it are reserved for use in this manner.

See the sample program in the previous chapter for an example which uses database sessions and CAS.

Appendix B

Code Dependencies

This application depends on several tools and libraries not included in this document. In addition to the *Generic ClearSilver and GLib Module Build Support* document (cs-glib.nw) and its dependencies, FastCGI support depends on libfcgi (tested with 2.4.0) and CAS support depends on libcurl (tested with 7.18.0). The database session support requires ODBC development libraries (tested with unixODBC and iODBC). The content encoding support requires zlib (tested with 1.2.3.3), and optionally a recent libarchive (tested with 2.7.902a, which is the minimum version).

And here are the library-supplied datatypes used, for the highlighter.

166 ⟨*(build.nw) Known Data Types* 9a⟩+≡ ◁114c

```
SQLHENV,SQLHDBC,SQLSTATE,SQLRETURN,SQLHSTMT,SQLSMALLINT,SQLINTEGER,SQLLEN,%
SQLHANDLE,GThreadPool,z_stream,archive,archive_entry,xmlDocPtr,xmlNodePtr,%
xmlNsPtr,xmlAttrPtr,FCGX_Request,%
```

Appendix C

Code Index