Getting Started with Seminole

Getting Started with Seminole

Table of Contents

List of Figures

List of Tables

List of Examples

Introduction

Getting started with Seminole may appear daunting; heck, it *is* daunting, what with a reference guide consisting of hundreds of pages of class, function, variable and constant declarations it's hard to get a grasp of where one even *starts* in attempting to integrate Seminole into a product.

To that end, this guide has been written to help you use Seminole in your application, along with many examples of using the various features of Seminole. All of the example programs referenced in this guide are included in the Seminole source code.

It is important to understand that many of the sample programs are written without all possible error checking in the interest of brevity. Although this guide assumes little knowledge of web technologies it does assume C/C++ programming knowledge.

Furthermore it is assumed that a build environment that supports Seminole is already configured and operational. The minimum build environment is a C++ compiler and Perl interpreter on a supported operating system. Porting Seminole to a new target platform is out of the scope of this document, although the required interfaces are described in the [Reference Guide].

Chapter 1. The Build System

Seminole includes a portable build system written in Perl. The build system can be used to build Seminole itself and, optionally, code that uses Seminole. All of the examples in this guide can be built using the build system.

Like **make**, source directories contain a file (called `Buildfile`) that describes how to build various targets, such as `default` which is the target that is assumed if none is specified for building object code or `clean` for removing built files.

Unlike **make** however the recipes and dependencies are expressed as Perl code with calls to helper routines that implement the "logic" of **make**. For building the examples in this guide it isn't necessary to write a `Buildfile`; each example includes a `Buildfile`.

But a `Buildfile` isn't the only file that is needed. Another file, called a "port" file determines the toolchain and parameters used for a particular build. The port files are located in `ports`. The port file `Seminole` contains the most basic machinery for building Seminole regardless of platform or toolchain. Other port files include `Seminole` but override or add functionality as appropriate.

For example, building using the FreeBSD™ port file includes the `POSIX` port file. The `POSIX` port file includes the common port file `Seminole`. More than one port file can be used during compilation. Port files can include other port files which define default constructs and then override those constructs as needed, similar to a C++ class hierarchy.

So let's jump right in and build one of the examples, start-1. We start by creating a new port file called `ports/start-1`:

```
@APPS = path_list("examples/start-1"); ❶
definitions(samepath($DEFINITION_FILE, 'FreeBSD')); ❷
```

❶ The `@APPS` array contains the list of application specific directories that are built in addition to Seminole. The `path_list` converts the list of paths given to it into a portable form. Alternatively you can use Perl's `File::Spec` package to create a portable file name.

❷ All of the other build parameters are inherited from the platform specific port file. In this case we are building on FreeBSD™.

Once created we can build the default target using that configuration file:

```
$ ./buildit ports/start-1
```

When complete the results of building that port file end up in `built/start-1`. It is possible to build other targets than the implicit `default` target. Another very important target is `clean`, used to remove the contents of `built/start-1`. To build a specific target simply name provide the name of the target following the port file:

```
$ ./buildit ports/start-1 clean
```

In addition to setting variables like `@APPS` which affect the build system, port files can also set build parameters in the code. For example, to help reduce code size we can turn off support for persistent

connections. And to reduce memory usage we can lower the maximum number of MIME entries per request. The build system provides a procedure `config` that adjusts the defaults provided by the `Seminole` port file:

```
@APPS = path_list("examples/start-1");
config(INC_PERSISTENT_CONN => 0, MAX_MIME => 12);
definitions(samepath($DEFINITION_FILE, 'FreeBSD'));
```

It is important that the `config` be called before the parent port file is included.

Before going further let us examine the `Buildfile` from the simplest example, start-1.

```
default => sub
{
  my @finaldeps = app_sources(); ❶

  die "target_binary not defined!\n"
   if (!defined $target_binary);

  my $demobin = $target_binary->('shttpd'); ❷

  stale([$demobin], [$SEMLIB, @finaldeps])  ❸
    and $target_link->($demobin,[$SEMLIB],@finaldeps); ❹
}
```

❶ This line calls the build-helper function `app_sources`. This function builds the list of source files for the application (including any automatically generated ones) and stores them in the `@finaldeps` array.

❷ The target port file defines the `$target_binary` function for computing the target filename given the provided basename, `shttpd`.

❸ The `stale` build-helper determines if the input files are newer than the output file.

❹ If `stale` indicates that the targets are out of date, the link step (defined as `$target_link` in the port file) is performed to build the requested products.

Chapter 2. Embedding Seminole

Getting your application talking HTTP

It's actually pretty easy to get Seminole embedded and running within your larger application. It's a simple matter of initializing the operating system portability layer within Seminole, instantiating an instance of a webserver object then starting it up so it can service requests. This only takes a few lines of code:

Example 2.1. Creating an `Httpd` object

```
#include "seminole.h"

int rc = Httpd::Init();
if (rc != 0)
{
  // failure to initialize the operating system portability layer.
  // This is a critical failure and Seminole will be unable to
  // function.
}

Httpd *server = new Httpd("www.example.net");
if (server == NULL)
{
  // we failed to instantiate a webserver object.
}

server->Start();        // start talking HTTP

// and in the location you shutdown your application, to stop
// the webserver ...

#if HTTPD_INC_SHUTDOWN
 server->Stop(HARD);
#endif
```

With just those few lines of code (plus the appropriate error handling mechanisms appropriate in your situation) your application now speaks HTTP.

Congratulations!

You can now use a web browser to make HTTP requests to your application. But if you notice, there isn't actually any *content* to serve up, all you get are dreaded 404 errors; which strictly speaking isn't all that bad at this point, since your application now speaks HTTP.

> **Experiment**
>
> Compile and run the start-1 application to see the above code in action!

Next, we'll serve some content up to show the boss.

Hello World!

Something to show the boss

The problem with just instantiating and starting Seminole like we did is that Seminole has no concept of what to serve, nor how to serve it up. It needs “handlers” that can generate content for a given request. Without that, all Seminole can do is blindly say “I don't know where the content is!”

All handlers for Seminole derive from the `HttpdHandler` class:

Example 2.2. Creating & Installing an example handler

```
ExampleHandler *handler = new ExampleHandler("/");
if (handler == NULL) // Assuming nothrow new.
{
  // failure to create a handler
}
server->Install(handler);
```

Handlers are responsible for handling a portion of the request space in a webserver (which we cover in the next section) and serving up the appropriate content for the requests it handles. Right now, we'll write a simple handler to just print “Hello World” regardless of the request made—the ubiquitous “Hello World” program, but for Seminole.

First we declare our handler class:

Example 2.3. Declaring a subclass of `HttpdHandler`.

```
class DemoHelloWorldHandler : public HttpdHandler
{
public:
  DemoHelloWorldHandler(const char *prefix);
  ~DemoHelloWorldHandler();
  virtual bool Handle(HttpdRequest *p_req);
};
```

And then the code for our class:

Example 2.4. Implementing a custom handler

```
DemoHelloWorldHandler::DemoHelloWorldHandler(const char *prefix)
{
  mpPrefix = HttpdUtilities::SaveString(prefix);
}

DemoHelloWorldHandler::~DemoHelloWorldHandler()
{
  HttpdOpSys::Free((char *)mpPrefix);
}
```

```
bool DemoHelloWorldHandler::Handle(HttpdRequest *p_req)
{
  if (IsMe(p_req))
  {
    bool is_head = p_req->IsHeadRequest();

    HttpdDynamicOutput output(p_req,is_head);
    p_req->Respond(HTTPD_RESP_OK);
    output.Header("Content-Type","text/html");
    output.HeaderComplete();

    output.Body()->Printf(
                            "<html>\n"
                            "<head>\n"
                            "  <title>Hello World!</title>\n"
                            "</head>\n"
                            "<body>\n"
                            "  <h1>Hello World!<h1>\n"
                            "</body>\n"
                            "</html>\n"
                            "\n"
                          );
    return (true);
  }

  return (false);  // we did NOT or do NOT handle this request
}
```

The constructor just saves that portion of the webserver space we're serving up (more on that in the next section—also see Appendices A and B) and the destructor just destroys the prefix. It's in the `Handle()` method were all the action takes place. While for this example we don't really *need* to call `IsMe`, most handlers will (or call `IsMyPath`—there's a subtle distinction between the two we'll get into in a later section).

Not all requests made to a webserver will result in content generation—in some cases only the data *about* the request will be sent back (see Appendix B for more information about HTTP) so we check to see if the content (considered the “body” in HTTP) needs to be generated, or only the information about the content (called the “header”). Once that distinction is made, based upon the type of request, we output a positive response (`HTTPD_RESP_OK`), plus the type of content (“text/html”), plus additional information that Seminole will generate for us, then (if requested) the actual content. Then an indication that we actually handled the request.

Now granted, the only response you'll now get is a page with “Hello World” but we'll fix that shortly.

Experiment

Compile and run the start-2 application to see the above code in action!

Note

Remember to use the clean build rule after editing the ports file to build a new sample application. When re-using the same ports file and changing only the application to build, a clean is required.

Multiple Handlers

A brief primer on URLs

or, Location location location

An HTTP URL describes the location of a resource retrievable via HTTP; a resource being an HTML document, a picture, sound, or just about anything else that can be transmitted digitally. A URL, such as: `http://www.example.net/aboutus/ceo.html`

has a protocol portion, “http”, a host portion, “www.example.net”, and a “URL path”—“/aboutus/ceo.html”. Webservers typically deal with just the URL path of the URL and the most common technique is to map a portion of the filesystem to match the URL path of requests is receives.

But not all requests have to map to a file, as we saw in the previous demo. And not all requests have to be funneled through a single handler in Seminole. The prefix parameter to the handler constructor gives the location within the URL path that the handler is responsible for.

Yet another demo handler

So let's define and add yet another handler. This time, it will print out the time or the date, depending upon an option given to its constructor:

```
class DemoTimeHandler : public HttpdHandler
{
private:
  bool mpTimeOrDate;

public:
  DemoTimeHandler(const char *prefix,bool TimeOrDate = false);
  ~DemoTimeHandler();
  virtual bool Handle(HttpdRequest *p_req);
};
```

If *`TimeOrDate`* is false, the handler will print a page with the current date, otherwise it prints the time:

```
int rc = Httpd::Init();
if (rc != 0)
{
  // failure to initialize the operating system portability layer.
  // This is a critical failure and Seminole will be unable to
  // function.
}

Httpd *server = new Httpd("www.example.net");
if (server == NULL)
{
  // we failed to instantiate a webserver object.
}
```

```
DemoHelloWorldHandler *phand = new DemoHelloWorldHandler("/");
if (phand == NULL)
{
  // critical failure
}

server->Install(phand);

DemoTimeHandler *ptdhand = new DemoTimeHandler("/time");
if (ptdhand == NULL)
{
  // critical failure
}

server->Install(ptdhand);

ptdhand = new DemoTimeHandler("/date",true);
if (ptdhand == NULL)
{
  // critical failure
}

server->Install(ptdhand);

server->Start(); // start talking HTTP

// and in the location you shutdown your application, to stop
// the webserver ...

#if HTTPD_INC_SHUTDOWN
 server->Stop(HARD);
#endif
```

You'll notice that we installed the `DemoHelloWorldHandler` as "/", one instance of the `DemoTimeHandler` as "/time" and another one as "/date". So, for all requests *other* than those that start with "/time" or "/date" the `DemoHelloWorldHandler` will print "Hello World", while all requests that start with "/time" will pass through the first instance of `DemoTimeHandler` (which in this case will print out the time) and all requests that start with "/date" will pass through the second instance (which will print out the current date).

Table 2.1.

`http://www.example.net/`	"Hello World"
`http://www.example.net/foo`	"Hello World"
`http://www.example.net/time`	current time
`http://www.example.net/timenow`	current time
`http://www.example.net/date`	current date
`http://www.example.net/date/then`	current date

A request will go through each installed handler; each handler will then check to see if it is supposed to handle this request, based upon the URL path it was installed with, via a call to either `IsMe` or `IsMyPath`.

If the handler is responsible for that portion of the URL path, it handles the request and returns true, otherwise it declines the request and returns false.

Code for DemoTimeHandler

And now the code for `DemoTimeHandler`:

Example 2.5. Example Time & Date Handler

```
DemoTimeHandler::DemoTimeHandler(const char *prefix, bool time_or_date)
{
  mpPrefix    = HttpdUtilities::SaveString(prefix);
  mTimeOrDate = time_or_date;
}

DemoTimeHandler::~DemoTimeHandler()
{
  HttpdOpSys::Free((char *)mpPrefix);
}

bool DemoTimeHandler::Handle(HttpdRequest *p_req)
{
  assert(p_req != NULL);

  if (IsMyPath(p_req))
  {
#if HTTPD_HAVE_CLOCK
    char         gmt_time[48];
    char         local_time[48];
    time_t       now;
    struct tm   *ptm;
    const char  *title;
    bool         is_head;

    is_head = p_req->IsHeadRequest();
    HttpdDynamicOutput output(p_req,is_head);
    p_req->Respond(HTTPD_RESP_OK);
    output.Header("Content-Type", "text/html");
    output.HeaderComplete();

    now = time(NULL);
    if (mTimeOrDate)
    {
      ptm = gmtime(&now);
      strftime(gmt_time, sizeof(gmt_time),
               "GMT Date: %A, %B %d, %Y", ptm);
      ptm = localtime(&now);
      strftime(local_time, sizeof(local_time),
               "Local Date: %A, %B %d, %Y", ptm);
      title = "The Date";
    }
    else
    {
```

```
      ptm = gmtime(&now);
      strftime(gmt_time, sizeof(gmt_time),
               "GMT Time: %H:%M:%S", ptm);
      ptm = localtime(&now);
      strftime(local_time, sizeof(local_time),
               "Local Time: %H:%M:%S", ptm);
      title = "The Time";
    }

    output.Body()->Printf
    (
      "<html>\n"
      "<head>\n"
      "  <title>%s</title>\n"
      "</head>\n"
      "<body>\n"
      "  <h1>%s</h1>\n"
      "  <p>%s</p>\n"
      "  <p>%s</p>\n"
      "</body>\n"
      "</html>\n"
      "\n",
      title, title, gmt_time, local_time
    );
#else // No clock supported.
    p_req->Respond(HTTPD_RESP_SRV_ERROR);
#endif

    return (true);
  }

  return (false);
}
```

There is quite a bit going on here, mostly dealing with formatting the time and date, but other than that, the logic here isn't all that much different than for `DemoHelloWorldHandler`—the code checks to see if it's responsible for handing this request, and if so, generates the required output, either the current time or date depending upon how the instance was created.

Of course, using boolean flags like this is poor object oriented programming. It was added to illustrate per-handler data.

The code also tests the compile time directive `HTTPD_HAVE_CLOCK`. This constant is defined for the benefit of Seminole but there is nothing wrong with user-written code using it as well. If the hardware that your application is being written for doesn't have a clock, then there would be no way for this handler to proceed.

You'll notice that if there is no clock, we set a response code of `HTTPD_RESP_SRV_ERROR` (which results in Seminole returning a 500 error response to the web browser) and we return true even though we failed to satisfy the request. Failure to satisfy the request is different than declining the request in the first place, which is an important thing to keep in mind. The `DemoTimeHandler` was responsible for handing the given request and although it could not fulfill the request (in this case, because of the lack of a feature) it still *handled* the request and thus has to return true. A handler can only return false if it is not responsible for handling requests elsewhere in the URL path.

Experiment

Compile and run the start-3 application to see the above code in action!

Chapter 3. Serving Content

Serving Files

The most common way to handle an HTTP request is by sending the contents of a file. Seminole provides a standard handler to do this called `HttpdFileHandler`. The `HttpdFileHandler` class implements the `HttpdHandler` interface to serve files from an abstract filesystem interface (`HttpdFileSystem` and friends).

The major phases of file handling can be overridden by subclasses wishing to change the default behavior. In fact this is exactly how Chapter 6, *The Template System* works.

Before studying the `HttpdFileHandler` class in depth, it is important to understand the way that filesystems are abstracted. A filesystem is described by a subclass of `HttpdFileSystem`. This class can be used to get file information (an instance of `HttpdFileInfo`) from a file name.

Once the file information is obtained, a file (or directory) can be opened using the filesystem object. This open file (or directory) object can then be used to perform I/O on the file.

Seminole supports a read-only filesystem designed for embedded web content that can be stored in read-only storage, called "ROM FS". Some Seminole ports (such as POSIX) have their native filesystem exposed via the `HttpdFileSystem` interface as well.

The build system can be configured, with a few lines in the `Buildfile` of the application, to process the web content in the build tree and convert it into an initialized array for ROM FS:

```
default => sub
{
  my $cpack_o = app_content(symbol => 'SemWeb',
                            filename => 'sitedata'); ❶
  my @appout = app_sources();
  my @finaldeps = ($cpack_o , @appout); ❷

  die "target_binary not defined!\n"
    if (!defined $target_binary);

  my $demobin = $target_binary->('shttpd');

  stale([$demobin],[$SEMLIB, @finaldeps])
    and $target_link->($demobin,[$SEMLIB],@finaldeps); ❸
}
```

❶ The `app_content` build helper does all the work in building the content package in the specified filename with the specified variable name.
❷ The results of the `app_content` helper is added to the `@finaldeps` array.
❸ The `@finaldeps` array is used in the link step as defined by the port.

Once the content is available it can be made into a filesystem object:

```
static HttpdMemoryDataSource builtin(SemWeb, sizeof(SemWeb)); ❶
HttpdRomFileSystem *p_romfs = new HttpdRomFileSystem;❷

int rc = p_romfs->Mount(&builtin); ❸
```

❶ The filesystem data (linked in from the build system) is wrapped up as a data source object.
❷ An instance of `HttpdRomFileSystem` is constructed to represent the content package in SemWeb.
❸ The `Mount` method is invoked to associate the filesystem object with the data source object. If successful the filesystem object may be used.

Once a filesystem instance is ready the `HttpdFileHandler` may be created and associated with an `Httpd` instance.

```
// This associates the root of the filesystem with the "/fs"
// prefix of the URL space.
HttpdHandler *p_hand = new HttpdFileHandler(p_romfs, "/", "/fs");
gpWebServer->Install(p_hand);
```

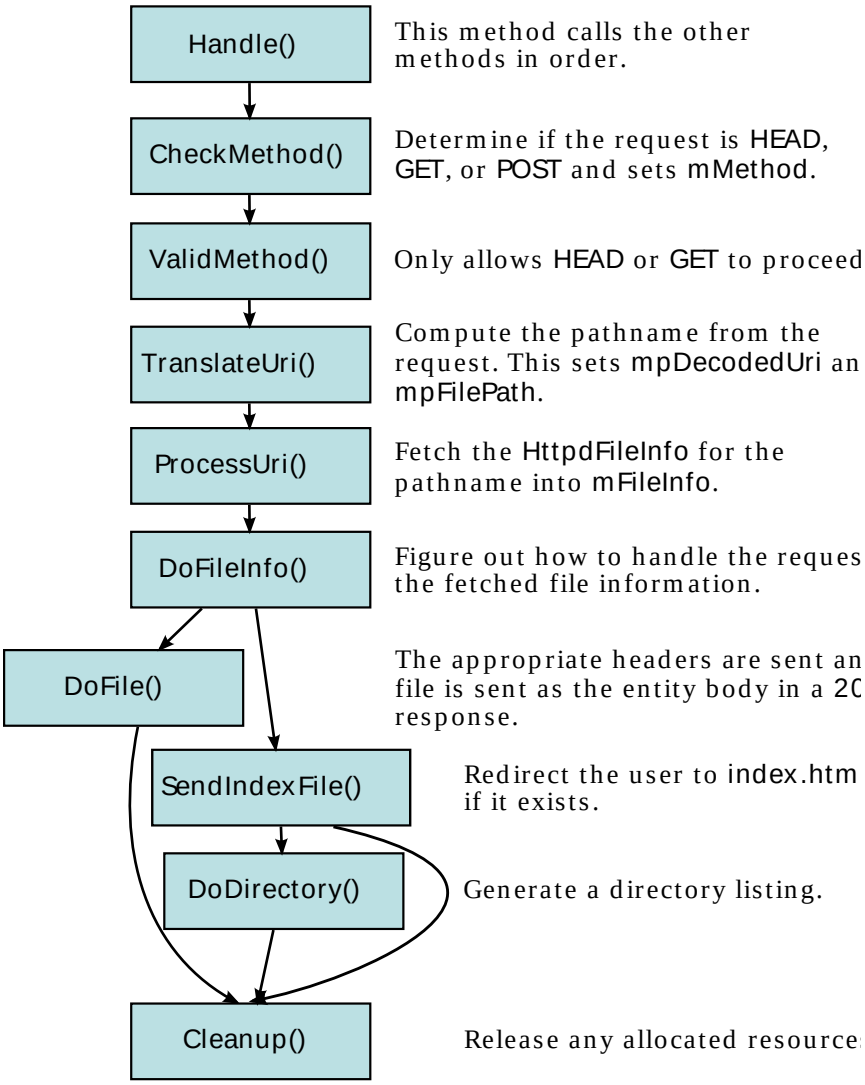
Experiment

Compile and run the file-1 application to see the above code in action!

Customizing the file handler

The `HttpdFileHandler` can be customized by subclassing and overriding methods. There are virtual methods for each of the phases of request processing that can be customized.

Figure 3.1. Overview of `HttpdFileHandler` call chain.



In this example we will make a file handler that is case-insensitive. The normal mode of operation of the ROM filesystem is case sensitivity. The approach is to always lowercase the path name in the `TranslateUri` method.

```
bool DemoFileHandler::TranslateUri(RequestState &state)
{
  char *p_walk;

  // Initialize mpFilePath so that all failure cases are
  // properly handled.
  state.mpFilePath = NULL; ❶

  // Decode the request path.
  state.mpDecodedUri = HttpdUtilities::UriDecode(state.mpReqPath); ❷
  if (state.mpDecodedUri == NULL)
    goto fail;
```

```
  // Lowercase the filename.
  for(p_walk = state.mpDecodedUri; *p_walk != '\0'; p_walk++) ❸
    *p_walk = tolower(*p_walk);

  // Convert to a filesystem path.
  state.mpFilePath = HttpdUtilities::Normalize(state.mpDecodedUri, ❹
                                               mpRootPath);
  if (state.mpFilePath == NULL)
    goto fail;

  return (true);

fail:
  state.mpRequest->Respond(HTTPD_RESP_SRV_ERROR);
  return (false);
}
```

❶ It is critical that this method initialize both the `mpFilePath` and `mpDecodedUri` members of the RequestState structure. In the event the decoding fails this early initialization will prevent returning without initializing `mpFilePath`. See the [Reference Guide] entry for `HttpdFileHandler::TranslateUri` for a detailed explanation

❷ The URL must be decoded in the event there are escaped characters. In this case there is no special processing.

❸ Now that the path portion of the URL is decoded it can be manipulated as desired. The path is lower-cased before being appended to the root of the handler in the event the root of the handler (which is under programmatic control) has case significance.

❹ Now that the path name has been altered it can be assembled with the root path and stored in the state object.

To illustrate the difference the file-2 example creates both a normal `HttpdFileHandler` instance and a `DemoFileHandler` instance on the same objects:

```
static HttpdMemoryDataSource builtin(SemWeb, sizeof(SemWeb));
HttpdRomFileSystem *p_romfs = new HttpdRomFileSystem;
if (p_romfs->Mount(&builtin) != 0)
  failure(__FILE__, __LINE__, "HttpdRomFileSystem::Mount()");

// Mount the demo handler to the root of the URI space.
HttpdHandler *p_demo = new DemoFileHandler(p_romfs, "/", "/");
gpWebServer->Install(p_demo);

// Mount the case-sensitive default handler to the /cs branch of
// URI space.
HttpdHandler *p_hand = new HttpdFileHandler(p_romfs,"/","/cs");
gpWebServer->Install(p_hand);
```

In this configuration the files in the filesystem can be referenced using a mixture of upper and lowercase unless they are accessed through the `/cs` prefix in which case they are case-sensitive.

Experiment

Compile and run the file-2 application to see the above code in action!

Content Preprocessing

Most websites try and give each page a consistent look. Typically this involves a common header and footer for every page. This can be very tedious when creating content. Many webservers provide a solution called “server-side includes.” These are directives that are parsed when a page is requested that allow other files to be included, a bit like the C preprocessor. Seminole allows this at runtime using the template mechanism but also provides another alternative: compile-time preprocessing.

Content may optionally be processed by the Seminole Content Package Generator (SCPG) with no runtime overhead. The most common use of this feature is to add common headers and footers to content. The example file-3 shows how this can be accomplished. There are several changes necessary to add compile-time headers and footers:

1. Add the `preproc` filter to the filter chain list in `content.cfg` for the MIME type of your content files:

```
# HTML Preprocessing
filter text/html - preproc html-squish
```

2. Create the header and footer files that will be included at the top and bottom of every page. In file-3 these are called `header.inc` and `footer.inc`.

3. Add the necessary preprocessor directives to the content files.

There are quite a few preprocessing directives and they are quite powerful. The [Reference Guide] covers these directives in detail.

Let us study the header template file:

```
<html>
<head>
 <title>%[eval env(TITLE)]%</title> ❶
%[if env(REFRESH)]% ❷
 <meta http-equiv="refresh"
          content="%[eval env(REFRESH)]%">
%[endif]%
%[unset REFRESH]% ❸
</head>
<body>
```

❶ The title variable is set in the including file and used to set the title of the HTML document.
❷ Here an option for the page to automatically refresh is checked. The `meta` tag is only included if the including file defines a value to the `REFRESH` preprocessor variable.
❸ Because the preprocessor variables exist outside the scope of a single file it is important to clear the `REFRESH` variable so that other files including this header do not also get a `meta` tag.

Now that we have an understanding of how the header and footer include files work let us turn our attention to using them. The `index.html` file brackets the content with directives for the header and footer:

```
%[set TITLE = Demo Page]% ❶
%[include "header.inc"]% ❷
  <img src="sem-logo1.jpg" align="center">
  <hr>
  This is content from the ROM file system!
%[include "footer.inc"]%
```

❶ Here the `TITLE` preprocessing variable is set so that the header will generate the proper page title.

❷ The `header.inc` file is included to give the page a common header. The file name given to the include directive is relative to the location of the including file. Ideally a relative path should always be provided to the include file.

In a real world usage the common header include file would contain navigational links and perhaps some CSS directives for a professional presentation.

Experiment

Compile and run the file-3 application to see the content preprocessing in action!

Chapter 4. Authentication and authorization

Seminole provides an authentication framework that can be applied to a `HttpdRequest` object. The `HttpdAuthenticator` class is an abstract base class. It must be subclassed with the actual password lookup logic supplied by the user.

For demonstration purposes our authenticator will use a hard-coded list of usernames and passwords. The realm will also be a constant. So let's implement those pure virtual methods:

```
bool DemoFileHandler::Authenticator::GetPassword(const char    *p_user,
                                                 HttpdRequest  *p_request,
                                                 char          *p_buf)
{
  const char *p_password;

  if (strcmp(p_user, "sam") == 0)
    p_password = "i_am36";
  else if (strcmp(p_user, "fred") == 0)
    p_password = "inbed";
  else if (strcmp(p_user, "bob") == 0)
    p_password = "4apples";
  else if (strcmp(p_user, "david") == 0)
    p_password = "mogen";
  else if (strcmp(p_user, "moshe") == 0)
    p_password = "boker";
  else if (strcmp(p_user, "sally") == 0)
    p_password = "aqua";
  else if (strcmp(p_user, "patty") == 0)
    p_password = "yumyum";
  else
    return (false);

  if (strlen(p_password) >= HTTPD_MAX_PASSWD_LENGTH)
    return (false);

  strcpy(p_buf, p_password);
  return (true);
}

void DemoFileHandler::Authenticator::Realm(HttpdRequest *p_request,
                                           char         *p_realm)
{
  strncpy(p_realm,
          "Yoyodyne Electric Banjo 2000",
          HTTPD_MAX_REALM_LENGTH - 1);
  p_realm[HTTPD_MAX_REALM_LENGTH - 1] = '\0';
}
```

As you can see the two methods simply provide data when needed by the authenticator. Now that the authenticator is complete it can be used during the `ProcessUri` phase of a `HttpdFileHandler`:

```
void DemoFileHandler::ProcessUri(RequestState &state)
{
  if (HttpdUtilities::IsUriPathPrefix(state.mpFilePath, "/secure")) ❶
  {
    if (mAuthenticator.Authenticate(state.mpRequest)) ❷
      HttpdFileHandler::ProcessUri(state); ❸
  }
  else
    HttpdFileHandler::ProcessUri(state);
}
```

❶ Here we check for either the directory listing itself (`/secure`), or anything beginning with `/secure`.

❷ If either of the conditions are true the authenticator is applied to the request object.

❸ If authentication was successful, then normal processing is performed. Otherwise, the `HttpdAuthenticator` has answered the request and no further processing is performed.

Experiment

Compile and run the auth-1 application to see the above code in action!

Chapter 5. Interfacing with CGI

A Note on Security

One mistake many novice web programmers make is not validating parameters from the browser properly. It is especially tempting to avoid validation if the validation is done with JavaScript on the client. This is of course not true because a malicious user can disable JavaScript or create their own form without it.

Malicious attempts to confuse a web application are not the only problem. Normal users can also change the expected state of a web application with the use of the browser's back and forward buttons. This additional hazard means that CGI code should never assume that a request will be issued only when it is expected.

Interfacing with CGI in Seminole is quite simple, although there are three different ways that CGI data can be sent to Seminole; two using the `POST` method, and one with the `GET` method. All three methods will be discussed with the following sample HTML:

Example 5.1. Sample HTML form page.

```
<FORM ACTION="/sample.cgi">
 <P>Name: <INPUT NAME="name" TYPE="text"></P>
 <P>Title: <INPUT NAME="title" TYPE="text"></P>
 <P>Age: <INPUT NAME="age" TYPE="text"></P>
 <P><INPUT TYPE="submit" VALUE="Submit"></P>
</FORM>
```

The first method is perhaps the most common method, `POST`:

```
<FORM ACTION="/sample.cgi" method="POST">
  …
</FORM>
```

The browser will default to sending the data with a content type of “application/x-www-form-urlencoded”. To extract the data from the client request, a call to `HttpdCgiParameter::ParsePostData` is made in your handler, with the appropriate checks:

Example 5.2. Getting CGI parameters

```
HttpdCgiParameter *list;

if (p_req->IsPostRequest())
{
  list = HttpdCgiParameter::ParsePostData(p_req);
```

```
  if (list == NULL)
  {
    p_req->Respond(HTTPD_RESP_CLIENT_ERROR);
    return(true);
  }
}
else
{
  p_req->Respond(HTTPD_REP_METHOD_NOT_ALLOWED);
  return(true);
}

const char *name  = list->Find("name");
const char *title = list->Find("title");
const char *age   = list->Find("age");

// ... rest of handler ...
```

Experiment

Compile and run the cgi-1 application to see the above code in action!

The second type of `POST` method, using the encoding type of “multipart/form-data”, is usually used to upload files to the server, but is equally easy to use:

```
<FORM ACTION="/sample.cgi" method="POST" enctype="multipart/form-data">
  …
</FORM>
```

```
HttpdCgiParamater *list;

if (p_req->IsPostRequest())
{
  HttpdMultipartCgiParser parser(p_req);
  if (parser.Parse() != 0)
  {
    p_req->Respond(HTTPD_RESP_METHOD_NOT_ALLOWED);
    return(true);
  }
  list = parser.TakeList();
  if (list == NULL)
  {
    p_req->Respond(HTTPD_RESP_CLIENT_ERROR);
    return(true);
  }
}
else
```

```
{
  p_req->Respond(HTTPD_REP_METHOD_NOT_ALLOWED);
  return(true);
}

const char *name  = list->Find("name");
const char *title = list->Find("title");
const char *age   = list->Find("age");

// ... rest of handler ...
```

The final method uses `GET`:

```
<FORM ACTION="/sample.cgi" method="GET">
  …
</FORM>
```

and should only be used when the data being passed in will *not* change the state of the server (or the device if the form is used for configuration); if the form is being used to check the status of something, or return results from a query (and the same results will always be returned for a given set of input) the `GET` can be used safely. And again, it's easy enough to get the data:

```
HttpdCgiParameter *list;

if (p_req->IsGetRequest())
{
  list = HttpdCgiParameter::ParseUriString(p_req->Query());
  if (list == NULL)
  {
    p_req->Respond(HTTPD_RESP_CLIENT_ERROR);
    return(true);
  }
}

const char *name  = list->Find("name");
const char *title = list->Find("title");
const char *age   = list->Find("age");

// ... rest of handler ...
```

The demo code present for this chapter has examples of not only the various ways of obtaining CGI parameters, but also the location of various bits of information that may be of interest when processing a CGI request, such as the server name, the port ithat the server is running on, and the IP address that the request is coming in from, among other such pieces of information:

```
output.Body()->Printf("Server:  %s:%d\r\n"
                      "Method:  %s\r\n"
                      "Version: %d.%d\r\n"
                      "Path:    %s\r\n"
                      "Query:   %s\r\n"
                      "URI:     %s\r\n",

      p_req->Server()->ServerHost(),
      p_req->Server()->Port(),
      p_req->Method(),
      (p_req->Protocol() >> 8) & 0xFF,
      (p_req->Protocol()     ) & 0xFF,
      p_req->Path(),
      p_req->Query()       ? p_req->Query()       : "",
      p_req->CompleteUri() ? p_req->CompleteUri() : ""
);

output.Body()->Printf(
                      "Client:  %d.%d.%d.%d\r\n"
                      "Headers:\r\n",
                      (p_req->ClientAddr() >> 24) & 0xFF,
                      (p_req->ClientAddr() >> 16) & 0xFF,
                      (p_req->ClientAddr() >>  8) & 0xFF,
                      (p_req->ClientAddr()      ) & 0xFF
                     );
```

Experiment

Compile and run the cgi-2 application to see these additional types of CGI processing in action.

Chapter 6. The Template System

A Brief Overview of using the Template System

Although dynamic content can be generated programmatically using the `HttpdDynamicOutput` class this approach is cumbersome. In most cases the dynamic content being generated is HTML Using `HttpdDynamicOutput` this would require that lots of the HTML for the user interface would be embedded in your source code.

This is bad for several reasons. Changes to the layout and style of the user interface requires code changes. Additionally this prevents the use of standard Web authoring tools.

Templates allow the HTML to be kept in files with embedded directives that refer to elements in your application. Although templates can be used in a custom handler the easiest approach is to subclass a `HttpdFileHandler` object.

```
class DemoHandler : public HttpdFileHandler
{
private:
 int  mHits;

protected:
 bool TranslateUri (RequestState &state);
 void DoFile       (RequestState &state);

public:
 DemoHandler (HttpdFileSystem *p_filesys,
              const char      *p_root,
              const char      *p_prefix);
};

//---------------

DemoHandler::DemoHandler(HttpdFileSystem *p_filesys,
                         const char      *p_root,
                         const char      *p_prefix)
 : HttpdFileHandler(p_filesys,p_root,p_prefix),
   mHits(0)
{}
```

In our example for the template system, we need to keep track of the number of hits we've received. We override `TranslateUri` to map all the requests to the single template file, and `DoFile` which will actually instantiate the template code and run through it.

```
bool DemoHandler::TranslateUri(RequestState &state)
{
  //-------------------------------------------------------
  // everything is going to be fed through default.thtm, but
  // decode the URI since it will be used later
  //-------------------------------------------------------
```

```
  state.mpDecodedUri = HttpdUtilities::UriDecode(state.mpReqPath);
  state.mpFilePath   = HttpdUtilities::SaveString("/default.thtm");

  return ((state.mpDecodedUri != NULL) && (state.mpFilePath != NULL));
}
```

The template file is `default.thtm`. The reason we create a copy of it using `HttpdUtilties::SaveString` is that the destructor will call `HttpdOpSys::Free`, and thus

```
  state.mpFilePath = "/default.thtm";
```

will fail and cause problems.

```
void DemoHandler::DoFile(RequestState &state)
{
  //-------------------------------------------------------------
  // this should always be true, but just in case ...
  //-------------------------------------------------------------

  if (strcmp(state.mFileInfo.MimeType(),"x-server-internal/template") == 0)
  {
    DemoSymbolTable symbols(++mHits);

    (void)HttpdFSTemplateShell::Execute(state,&symbols);
  }
  else
    state.mpRequest->Respond(HTTPD_RESP_SRV_ERROR);
}
```

If the MIME type is `x-server-internal/template` (and for this example it should be) we instantiate a `DemoSymbolTable` object, and call `HttpdFSTemplateShell::Execute` over the file, thus executing the template and generating the output.

Note

The above example has no synchronization for the global variables `mHits` within the handler object. In a real world usage these variables must be properly synchronized if the platform on which Seminole is running is multi-threaded.

Before we get into the details of that, let's take a look at the template file:

```
<html>
  <head>
    <title>Template Demonstration</title>
  </head>

  <body>
```

```
        <h1>Template Demonstration</h1>

        <h2>EVAL example</h2>

        <p>%{eval:evaltest}%</p>

        <h2>LOOP example</h2>

        <ol>
          %{loop:looptest}%
            <li>%{eval:looptest}%</li>
          %{endloop}%
        </ol>

        <h2>IFELSE example</h2>

        <p>You have requested an
            %{if:even}%
                    even
            %{else}%
                    odd
            %{endif}%
        number of times.</p>

        <h2>IFELSE chain example</h2>

        <p>
            %{if:first}%
                    This is the first request.
            %{elseif:two}%
                    This request is a multiple of two.
            %{elseif:three}%
                    This request is a multiple of three.
            %{elseif:five}%
                    This request is a multiple of five.
            %{else}%
                    You have made multiple hits.
            %{endif}%
        </p>

        <h2>END OF LINE</h2>

      </body>
    </html>
```

A fairly straightforward HTML file, with the exception of stuff like

```
<p>%{eval:evaltest}%</p>
```

and

```
<ol>
 %{loop:looptest}%
  <li>%{eval:looptest}%</li>
 %{endloop}%
</ol>
```

The template processor looks for template directives between pairs of `%{` and `}%`. There are several directives, but they all fall into one of three catagories:

Execution	`%{eval:`*`name`*`}%`
Looping	`%{loop:`*`name`*`}%` … `%{endloop}%`

Conditional	`%{if:`*`name`*`}%` … `%{endif}%`
	`%{if:`*`name`*`}%` … `%{else}%` … `%{endif}%`
	`%{if:`*`name`*`}%` … `%{elsif:`*`name`*`}%` … `%{endif}%`
	`%{if:`*`name`*`}%` … `%{elsif:`*`name`*`}%` … `%{else}%` … `%{endif}%`

Inverted conditional	`%{ifnot:`*`name`*`}%` … `%{endif}%`

`name` specifies a particular instance of code to run (or loop or tests of a condition) which is handled in the symbol table, in this case, `DemoSymbolTable`, which is subclassed from `HttpdSymbolTable`:

```
class HttpdSymbolTable
{
public:
 virtual int HandleEval(HttpdEvalCommand          *p_eval);
 virtual int HandleLoop(HttpdLoopCommand          *p_loop);
 virtual int HandleCond(HttpdConditionalCommand   *p_cond);

 static int ReturnBool(bool value);

 virtual ~HttpdSymbolTable();
};
```

The derived `HttpdSymbolTable` needs to define at least one of `HandleEval`, `HandleLoop` or `HandleCond`, if not all (depending upon your needs). The demo code will handle all three, plus keep track of some information used during the processing:

```
class DemoSymbolTable : public HttpdSymbolTable
{
private:

 size_t mLoopCount;
 int    mHits;
```

```
public:
 DemoSymbolTable (int hits)
   : mLoopCount(0) , mHits(hits) { }

 virtual int HandleEval(HttpdEvalCommand        *peval);
 virtual int HandleLoop(HttpdLoopCommand        *ploop);
 virtual int HandleCond(HttpdConditionalCommand *pcond);
};
```

We initialize the symbol table with the number of hits so far. The simplest directive, evaluation is implemented in the `HandleEval` method. In our case it just generates a bit of output to replace the directive call in the template:

```
int DemoSymbolTable::HandleEval(HttpdEvalCommand *peval)
{
  assert(peval != NULL);

  const char *name      = peval->Name();
  HttpdWritable *output = peval->Output();

  if (strcmp(name,"evaltest") == 0)
    return (output->Printf("%s at %s (%d hits)",
                           __DATE__,
                           __TIME__,
                           mHits));
  else if (strcmp(name,"looptest") == 0)
    return (output->Printf("%s", mLoopText[mLoopCount]));
  else
    return (HTTPD_TEMPLATE_UNKNOWN_NAME);
}
```

Here we get the *`name`* being evaluated by the `eval` directive, and the output stream. If the template is evaluating the `evaltest`, it just prints out the date and time the program was compiled, along with the current number of hits. Otherwise, for `looptest` it just prints out the textual representation of the loop count—we'll get into more detail about that later. Just note that the `looptest` under the `eval` directive references a different portion of code than the `looptest` under the `loop` directive.

The next easiest are the conditional tests. The `even` conditional tests if the hit count is even. `first` returns if this is the first request, and `two`, `three` and `five` return if the number of hits are evenly divisible by two, three and five respectively.

```
int DemoSymbolTable::HandleCond(HttpdConditionalCommand *pcond)
{
  assert(pcond != NULL);

  const char *name = pcond->Name();

  if (strcmp(name,"even") == 0)
    return (ReturnBool((mHits % 2) == 0));
  else if (strcmp(name,"first") == 0)
    return (ReturnBool(mHits == 1));
```

```
  else if (strcmp(name,"two") == 0)
    return (ReturnBool((mHits % 2) == 0));
  else if (strcmp(name,"three") == 0)
    return (ReturnBool((mHits % 3) == 0));
  else if (strcmp(name,"five") == 0)
    return (ReturnBool((mHits % 5) == 0));
  else
    return (HTTPD_TEMPLATE_UNKNOWN_NAME);
}
```

The call to `ReturnBool` will turn `true` into `HTTPD_TEMPLATE_TRUE_CASE` and `false` into `HTTPD_TEMPLATE_FALSE_CASE`.

Warning

Do *NOT* return `true` or `false` from `HandleCond`.

And finally we come to the loop directive. And again, it's similar to the other handlers:

```
int DemoSymbolTable::HandleLoop(HttpdLoopCommand *ploop)
{
  assert(ploop != NULL);

  if (strcmp(ploop->Name(),"looptest") == 0)
  {
    for
    (
      mLoopCount = 0;
      mLoopCount < HTTPD_NUMELEM(mLoopText);
      mLoopCount++
    )
    {
      int rc;

      rc = ploop->Iterate();
      if (rc != 0)
        return (rc);
    }
    return (0);
  }
  else
    return (HTTPD_TEMPLATE_UNKNOWN_NAME);
}
```

The demo code just loops for every element in `mLoopText`:

```
static const char *const mLoopText[] =
{
  "one" , "two"   , "three" , "four" , "five" ,
  "six" , "seven" , "eight" , "nine" , "ten"
};
```

HTTPD_NUMELEM is a macro to return the number of elements in an arbitrary array. In this case, we simply loop through each item in `mLoopText`, calling `Iterate`, which runs through the text between `%{loop:`*`name`*`}%` and `%{endloop}%`, which in this case, consists of:

```
<li>%{eval:looptest}%</li>
```

which ends up calling `HandleEval` each time, which (if you go back up and look) will cause the appropriate element of `mLoopText` to be printed.

Note

This example performs template processing by overriding the `TranslateUri` and `DoFile` methods of the `HttpdFileHandler`. This is meant for expositional purposes only; to show what is possible.

In most scenarios template processing is usually carried out by overriding the `SendFile` method. We will see an example of the latter approach in the next section.

Experiment

Compile and run the template-1 application to see the above code in action!

Symbol Maps

Most symbol tables don't need to do anything tricky with the name or keep much state. For these cases there is a helper class, `HttpdSymbolMap` that can be used to add symbols in a table-driven fashion.

Let's assume the following structure is something that we would like to expose via a template:

```
struct UserAccount
{
  char            mUserName[32];
  const char     *mpRole;
  long            mFailedLogins;
  unsigned long   mUniqueId;
  bool            mLoggedIn;
};
```

Instead of subclassing `HttpdSymbolTable` and using a string of if-else statements to decide which fields are displayed we can use a symbol map. The symbol map is an array of HttpdSymbolEntry structures describing each valid symbol and some callbacks to handle them. The structure is defined as follows:

```
struct HttpdSymbolEntry
{
  const char  *mpName;
  size_t       mOffset;
  int        (*mpEvalCommand)(HttpdEvalCommand        *p_cmd,
```

```
                              const void              *p_data);
  int        (*mpLoopCommand)(HttpdLoopCommand        *p_cmd,
                              const void              *p_data);
  int        (*mpCondCommand)(HttpdConditionalCommand *p_cmd,
                              const void              *p_data);
};
```

The three callback routines of the HttpdSymbolEntry are all optional and can be NULL if no function is provided. The callbacks are given a pointer to their associated data (computed using the offset) and the command object. Fortunately, several default callbacks are implemented for some primitive types. Using these built-in callbacks the symbol map for the UserAccount object looks like this:

```
const HttpdSymbolEntry user_account_map[] =
{
  {
    "id",                                   // Symbol name
    offsetof(UserAccount, mUniqueId),       // Offset of field
    HttpdSymbolMap::EvalHexUlong,           // Eval callback
    NULL,                                   // Loop callback
    NULL                                    // Conditional callback
  },

  {
    "login-fail",                           // Symbol name
    offsetof(UserAccount, mFailedLogins),   // Offset of field
    HttpdSymbolMap::EvalLong,               // Eval callback
    NULL,                                   // Loop callback
    NULL                                    // Conditional callback
  },

  {
    "name",                                 // Symbol name
    offsetof(UserAccount, mUserName),       // Offset of field
    HttpdSymbolMap::EvalStringBuffer,       // Eval callback
    NULL,                                   // Loop callback
    NULL                                    // Conditional callback
  },

  {
    "online",                               // Symbol name
    offsetof(UserAccount, mLoggedIn),       // Offset of field
    NULL,                                   // Eval callback
    NULL,                                   // Loop callback
    HttpdSymbolMap::CondBool                // Conditional callback
  },

  {
    "role",                                 // Symbol name
    offsetof(UserAccount, mpRole),          // Offset of field
    HttpdSymbolMap::EvalString,             // Eval callback
    NULL,                                   // Loop callback
    NULL                                    // Conditional callback
```

```
  }
};
```

Important

The names of the symbols in the symbol map must always be in alphabetical order.

Now it is just a matter of making an instance of of the `HttpdSymbolMap` associated with that table and the object:

```
UserAccount *p_some_account = GetAccount();

HttpdSymbolMap sym_map(user_account_map,
                       HTTPD_NUMELEM(user_account_map),
                       p_some_account);
```

Experiment

Compile and run the template-2 application to see the above code in action!

Multiple Symbol Tables

For large and complex applications it's unwieldy to have a single scope of symbols when evaluating templates. Scopes can come and go in a last-in first-out order. A good example of this is during the `HandleLoop` method. For each iteration of a template loop, additional symbols specific to that iteration can be pushed onto the symbol table list.

A small helper class, `HttpdTemplateScope` allows the scoping of the templates to mimic the scope of your C++ code. The `HttpdScopedSymbolMap` combines a symbol map with a template scope. Let's see how that is used in an example.

```
int DemoSymbolTable::HandleLoop(HttpdLoopCommand *p_loop)
{
  if (strcmp(p_loop->Name(), "accounts") == 0)
  {
    for(size_t i = 0; i < HTTPD_NUMELEM(gAccounts); i++)
    {
      // Allocate a new scope and populate it with the symbol map
      // of this account. Further loops can create more nested scopes
      // if desired.
      HttpdScopedSymbolMap ssmap(p_loop->Processor(),
                                 user_account_map,
                                 HTTPD_NUMELEM(user_account_map),
                                 gAccounts + i);

      if (p_loop->Iterate() != 0)
        break;
    }
```

```
    return (0);
  }

  return (HTTPD_TEMPLATE_NOT_HANDLED);
}
```

This will loop over all of the records in the gAccounts array. When the scope containing the `HttpdScopedSymbolMap` exits the symbols it was providing are removed from the template processor. Loops can nest in this fashion as necessary.

Experiment

Compile and run the template-3 application to see the above code in action!

Interfacing Templates with CGI

There are a few classes provided for making CGI parameters exposed as a template symbol table. This behavior can be very useful when filling out web forms, for example. This symbol table can be easily installed once a parameter list is obtained.

```
…
if (strcmp(state.mFileInfo.MimeType(),"x-server-internal/template") == 0)
{
  HttpdCgiParameter *p_params;

  p_params = HttpdCgiParameter::ParseUriString(state.mpRequest->Query()); ❶
  HttpdCgiListSymbols symbols(p_params); ❷

  (void)HttpdFSTemplateShell::Execute(state,&symbols);❸
  HttpdCgiParameter::FreeList(p_params);
}
else
  state.mpRequest->Respond(HTTPD_RESP_SRV_ERROR);
…
```

❶ As with other CGI examples a parameter list is generated. In this case the `GET` method of form posting is used.

❷ The bridge object, of class `HttpdCgiListSymbols` is constructed over the parameter list.

❸ The symbol table is passed to the execution environment. All of the handling is performed by the pre-canned symbol table.

This will allow the intelligent processing of the form within the template.

Experiment

Compile and run the template-4 application to see the above code in action!

Chapter 7. Sessions & Cookies

Maintaining state across requests

Because HTTP is a stateless protocol it is the responsibility of the server to manage the per-client state. Seminole provides two mechanisms to help with this task. The first mechanism, the session manager maintains state objects with a small tag that is efficient to send across the network. The session manager handles the expiration of defunct sessions, as well as the error checking.

The second mechanism is the `HttpdCookie` class. This class provides an interface over the cookie protocol. Cookies are a client-side option offered by some browsers that minimize the overhead in tracking clients. The session manager can use cookies to identify which client goes with which session.

The use of cookies is not required with the session manager however. Instead, CGI parameters can be used. Let's begin by examining the cookie approach.

Let's examine some code. To find the session key using cookies the `HttpdCookies` is used like this:

```
bool DemoSessionHandler::Handle(HttpdRequest *p_req)
{
  assert(p_req != NULL);

  if (IsMe(p_req))
  {
    HttpdCookies  cookie_jar(p_req->Headers());

    while (cookie_jar.NextCookie())
    {
      if (strcmp(cookie_jar.Key(), "session") == 0) ❶
      {
        ExistingSession(p_req, cookie_jar.Value());
        return (true);
      }
    }

    NewSession(p_req); ❷
    return (true);
  }

  return(false);
}
```

❶ If the cookie with the session identifier is present then (assuming the session is valid) this visitor has a previous state.

❷ This visitor has no existing state. A new state object must be created.

The session objects are subclasses of `HttpdSessionObject` and can contain whatever user data is needed. To start with let us simply count the number of visits this browser has performed.

```
class DemoSessionObject : public HttpdSessionObject
{
public:
    unsigned int  mVisitCount;
};

void DemoSessionHandler::NewSession(HttpdRequest  *p_req)
{
  void *p_buffer = HttpdOpSys::Malloc(sizeof(DemoSessionObject)); ❶
  if (p_buffer == NULL)
  {
    p_req->Respond(HTTPD_RESP_SRV_ERROR);
    return;
  }
  DemoSessionObject *p_obj = new(p_buffer) DemoSessionObject;

  p_obj->mVisitCount = 1; ❷

  if (mSessions.Insert(p_obj) != 0) ❸
  {
    delete p_obj;
    p_req->Respond(HTTPD_RESP_SRV_ERROR);
    return;
  }

  // Send the first page.
  char ses_id[HTTPD_SESSION_KEY_LEN]; ❹
  p_obj->SessionId(ses_id);
  mSessions.Unlock(p_obj); ❺
  Output(p_req, "Welcome new visitor! I'll count your visits.", ses_id);
}
```

❶ Allocate memory for the session object on the heap. When the session is expired the session manager will automatically free the object.

❷ Initialize the session and insert it into the manager. Only the user defined fields need initialization.

❸ Insert the session object into the session manager. If this fails destroy the session and handle the failure. Once the object is inserted the session identifier will be valid.

❹ Now the session key can be sent as a cookie back to the visitor along with the response content.

❺ Once inserted the session manager will lock the object to avoid it being immediately removed while further processing is in progress.

Examining the `Output` routine shows that the `HttpdCookies` class is how the session identifier is stored on the client:

```
void DemoSessionHandler::Output
 (
   HttpdRequest   *p_req,
   const char     *p_text,
   const char     *p_ses_id
 )
{
  bool is_head = p_req->IsHeadRequest();
```

```
    HttpdDynamicOutput output(p_req, is_head);
    p_req->Respond(HTTPD_RESP_OK);
    output.Header("Content-Type","text/html");

    if (p_ses_id != NULL) ❶
      HttpdCookies::SendCookie(&output,
                               "session",
                               p_ses_id,
                               "path",
                               Prefix(),
                               (const char *)0);
    output.HeaderComplete();
    output.Body()->Printf("<html><head>\n"
                          "<title>Session &amp; Cookie Demo"
                          "</title></head>\n"
                          "<body>%s</body></html>\n",
                          p_text);
  }
```

❶ The session key is written as a cookie with the `path` attribute set to reflect the path of the handler.

If the session key is provided by the client is must be validated by the session manager before it can be used:

```
 void DemoSessionHandler::ExistingSession
  (
    HttpdRequest  *p_req,
    const char    *p_id
  )
 {
   // Find the session.
   HttpdSessionObject *p_so;
   if (mSessions.Find(p_id, p_so) != 0) ❶
   {
     NewSession(p_req);
     return;
   }
   DemoSessionObject *p_session = (DemoSessionObject *)p_so;

   char  buffer[128];
   sprintf(buffer,
           "Welcome back. You have visited %u times so far.",
           p_session->mVisitCount++);
   mSessions.Unlock(p_so); ❷
   Output(p_req, buffer);
 }
```

❶ The session identifier is passed to the `Find` method of the session manager. This will validate that the session exists and if it does provide a pointer to the session object.

❷ To prevent the session manager from deleting the session while it is being used `Find` keeps a reference count. The `Unlock` method releases that reference count.

Experiment

Compile and run the session-1 application to see the above code in action!

Making sessions more secure

The session-1 example has a slight security flaw. While the session identifiers are difficult to guess if they are compromised (say through packet sniffing) a session can be hijacked. A few additional lines of code will verify that the requesting IP address matches the address of the creator of the session.

This is not done by default because there are some network configurations that can cause problems with this verification (multi-homed clients). However in many cases this additional security is justified. The verification of the IP address requires the addition of the mClientAddr:

```
class DemoSessionObject : public HttpdSessionObject
{
public:
  unsigned int   mVisitCount;
  unsigned int   mVisitCount;
  HttpdIpAddress mClientAddr; ❶
};
```

❶ This field will store the IP address of the creator.

When creating the session the IP address of the client should be assigned to the session object.

```
void DemoSessionHandler::NewSession(HttpdRequest  *p_req)
{
  void *p_buffer = HttpdOpSys::Malloc(sizeof(DemoSessionObject));
  if (p_buffer == NULL)
  {
    p_req->Respond(HTTPD_RESP_SRV_ERROR);
    return;
  }
  DemoSessionObject *p_obj = new(p_buffer) DemoSessionObject;

  p_obj->mVisitCount = 1;
  p_obj->mClientAddr = p_req->ClientAddr(); ❶
  if (mSessions.Insert(p_obj) != 0)
  {
    delete p_obj;
    p_req->Respond(HTTPD_RESP_SRV_ERROR);
    return;
  }

  // Send the first page.
  char ses_id[HTTPD_SESSION_KEY_LEN];
  p_obj->SessionId(ses_id);
  mSessions.Unlock(p_obj);
  Output(p_req, "Welcome new visitor! I'll count your visits.", ses_id);
```

```
}
```

❶ The client address is stored in the session object.

The only remaining step is to verify the IP address:

```
void DemoSessionHandler::ExistingSession
 (
   HttpdRequest  *p_req,
   const char    *p_id
 )
{
  // Find the session.
  HttpdSessionObject *p_so;
  if (mSessions.Find(p_id, p_so) != 0)
  {
    NewSession(p_req);
    return;
  }
  DemoSessionObject *p_session = (DemoSessionObject *)p_so;

  if (p_session->mClientAddr != p_req->ClientAddr()) ❶
  {
    NewSession(p_req);
    return;
  }

  char  buffer[128];
  sprintf(buffer,
          "Welcome back. You have visited %u times so far.",
          p_session->mVisitCount++);
  mSessions.Unlock(p_so);
  Output(p_req, buffer);
}
```

❶ If the address is invalid then consider the session invalid.

Experiment

Compile and run the session-2 application to see the above code in action!

Chapter 8. Drawing Images

Basic drawing

Handlers may return any kind of data. The imaging library is a support package for handlers which allows them to return dynamically created graphical images. This can be especially useful for displaying numeric data graphically.

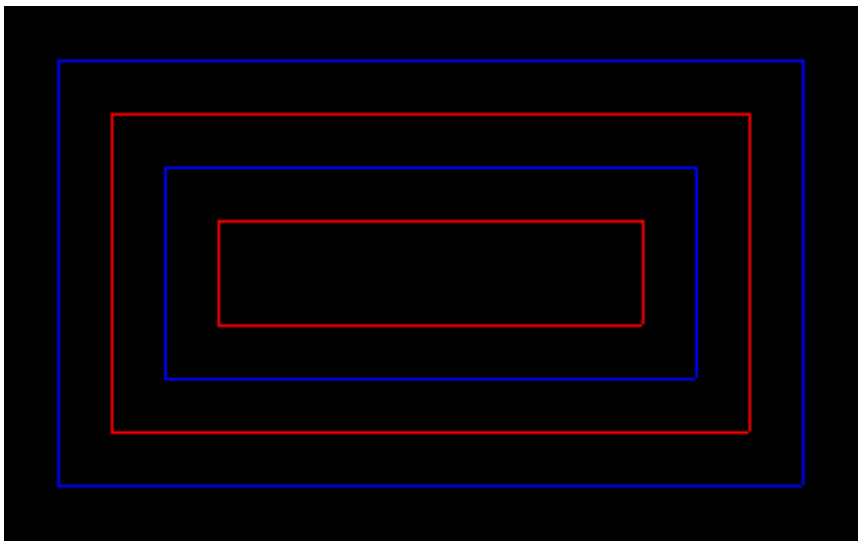
Most of the work to generate a dynamic image is done by the `HttpdGif87aRenderer` class. This class represents a drawing canvas. Once drawing is complete a single method call will handle rendering the canvas to the `HttpdRequest` object.

Building a demo with the following handler is all that is necessary to get a dynamic image handler.

```
bool DemoImageHandler::Handle(HttpdRequest *p_req)
{
  assert(p_req != NULL);

  if (IsMe(p_req))
  {
    HttpdGif87aRenderer rend;
    HttpdColor          red, blue;

    if (rend.Create(320, 200, 8) != 0)
      goto failure;
    if (rend.Color(255, 0, 0, 0, red) != 0)
      goto failure;
    if (rend.Color(0, 0, 255, 0, blue) != 0)
      goto failure;

    HttpdRect  r;
    rend.Size(r);

    boolean flip = false;
    for(unsigned int i = 0; i < 4; i++)
    {
      r.Deflate(20, 20);

      // Draw the background.
      rend.Pen(flip ? red : blue);
      flip = !flip;
      rend.Box(r);
    }

    // Render the output.
    rend.Render(p_req);
    return (true);
  }

  return (false);
```

```
failure:
  p_req->Respond(HTTPD_RESP_SRV_ERROR);
  return (true);
}
```

The above example produces the following image:



Experiment

Compile and run the image-1 application to see the above code in action!

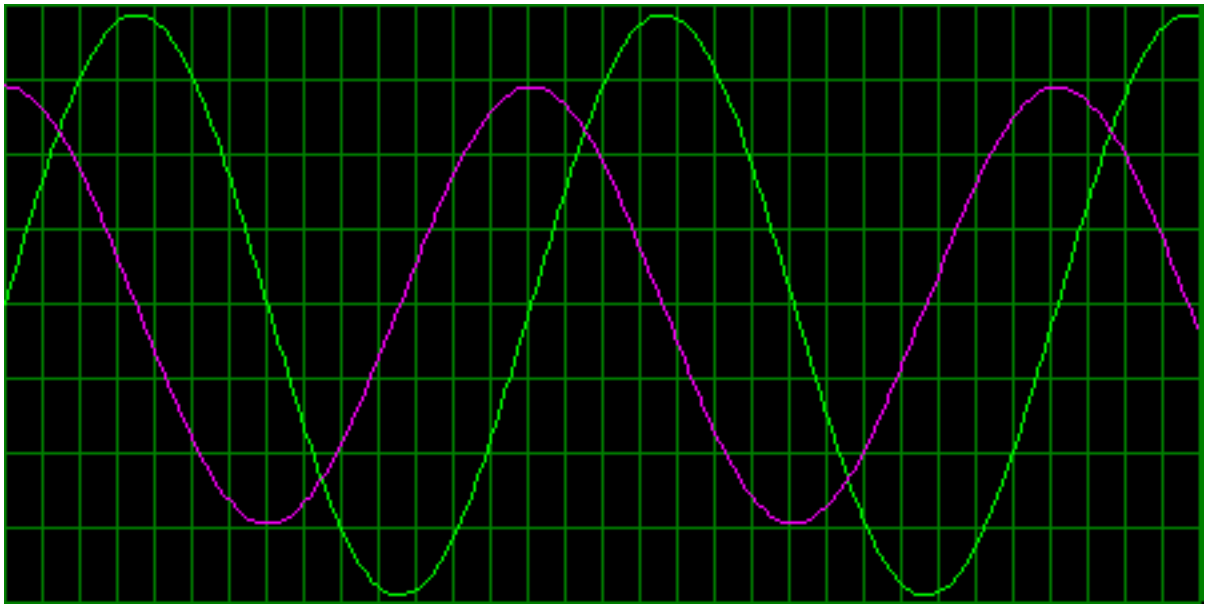
Rendering Numerical Data

Embedded devices are often sampling analog signals. Seminole makes it easy to display those analog signals graphically. The following example will graph a sine wave and an overlapping cosine waveform of differing amplitudes dynamically.

```
bool DemoImageHandler::Handle(HttpdRequest *p_req)
{
  assert(p_req != NULL);

  if (IsMyPath(p_req))
  {
    HttpdGif87aRenderer rend;
    HttpdColor          ltgrn, dkgrn;
    HttpdColor          purple;

    long signal_data[180];

    if (rend.Create(450, 225, 8) != 0)
      goto failure;
    if (rend.Color(0, 255, 0, 0, ltgrn) != 0)
```

```
      goto failure;
    if (rend.Color(0, 128, 0, 0, dkgrn) != 0)
      goto failure;
    if (rend.Color(255, 0, 255, 0, purple) != 0)
      goto failure;

    HttpdRect  r;
    rend.Size(r);

    // Draw the background.
    rend.Pen(dkgrn);
    rend.Box(r);
    rend.Grid(r, 30, 8);

    // Calculate a sine wave.
    for(size_t i = 0; i < HTTPD_NUMELEM(signal_data); i++)
      signal_data[i] = (long )(100 * sin(i / (3.1415 * 4)));

    // Graph the data.
    rend.Pen(ltgrn);
    rend.LineGraph(r,
                   signal_data,
                   HTTPD_NUMELEM(signal_data),
                   -102,
                   102,
                   0);

    // Calculate a cosine wave.
    for(size_t i = 0; i < HTTPD_NUMELEM(signal_data); i++)
      signal_data[i] = (long )(75 * cos(i / (3.1415 * 4)));

    // Graph the data.
    rend.Pen(purple);
    rend.LineGraph(r,
                   signal_data,
                   HTTPD_NUMELEM(signal_data),
                   -102,
                   102,
                   0);

    // Render the output.
    rend.Render(p_req);
    return (true);
  }

  return (false);

failure:
  p_req->Respond(HTTPD_RESP_SRV_ERROR);
  return (true);
}
```

The above example produces the following image:



Experiment

Compile and run the image-2 application to see the above code in action!

Chapter 9. Endpoint Discovery

Finding Endpoints

The most amazing web interface for your product is useless if finding the URL is very difficult. To solve this problem the `HttpdDiscoveryServer` comes to the rescue. When combined with the Java-based discovery client, only a well-known URL (such as a public website) is necessary to find all the available web interfaces on your network.

As with most components in Seminole, the `HttpdDiscoveryServer` is fairly self contained. It does require a configuration structure that describes the information that is to be presented. Below is the configuration structure used in the discovery-1 application:

```
static const char *const dev_class[] =
{
  "demos", ❶
  NULL
};

static const HttpdPair params[] =
{
  { "descr",   "A computer running the Seminole discovery demo." }, ❷
  { "os",      HTTPD_OS_NAME                                      }  ❸
};

static const HttpdDiscoveryServer::Config disc_config =
{
  &HttpdDiscoveryServer::mDefaultNetwork, ❹
  dev_class, ❺
  HTTPD_NUMELEM(params), params ❻
};
```

❶ The discovery client can be configured to only display certain classes of device. This list defines the classes that this device falls under. For the demo we simply use the string `demos`. You should either name them uniquely (i.e. prefixed with your company name) or contact GladeSoft about the appropriate values for your product.

❷ The discovery server may transmit an open-ended list of name/value pairs to the client for display. As a general convention `descr` is a description of the endpoint.

❸ Here we add our own attribute, `os` that has a value of `HTTPD_OS_NAME` which is set by the build system.

❹ In most cases the default network parameter structure can be used.

❻ The attribute table and its length (computed here using the HTTPD_NUMELEM macro) are pointed to by the configuration structure.

Once the configuration structures are in place and you have an instance of `Httpd` running you can start the discovery server:

```
HttpdDiscoveryServer ds(p_webserv, &disc_config); ❶
```

```
rc = ds.Create(); ❷
if (rc != 0)
  …

rc = ds.Start(); ❸
if (rc != 0)
  …
```

❶ Here the discovery server is created and given the address of the webserver, *p_webserv* and the address of the configuration structure, *disc_config*.

❷ Before the discovery server object can be started it must be created. This method should only be invoked once. If it returns an error (non-zero) the object should not be used.

❸ All that is left is to start the server. It can be started and stopped dynamically as necessary.

Experiment

Compile and run the discovery-1 application and run it on one machine on your network. Then build the `discovery_client` target (you will need the JDK installed for this). Take the `built/`*`PORT`*`/lib/Discovery.jar` file and `src/examples/discovery-1/viewapp.html` to another machine on your network and open `viewapp.html`.

The discovery client should locate the other machine running discovery-1 automatically.

Dynamic Data via the Discovery Server

The discovery-1 application is a good example of sending static data, such as the value of `HTTPD_OS_NAME`, to the discovery client. But what if you wanted to send something more dynamic, like the status of the system?

The discovery-2 sends the current time to discovery clients. You can see this change dynamically as long as the discovery client is left open.

Sending dynamic data involves subclassing `HttpdDiscoveryServer`. Overriding the `BuildResponse` it is easy to add any kind of dynamic data that can be computed programatically.

Note

Remember that `BuildResponse` is executed on its own thread. Therefore it is necessary to ensure that any code in this method properly synchronizes with other threads when accessing shared data.

Here is the `BuildResponse` implementation in our subclass, `DemoDiscoveryServer`:

```
int DemoDiscoveryServer::BuildResponse(HttpdCgiWriter *p_writer)
{
  int         rc;
  time_t      now;
  struct tm  *p_lt;
  char        buf[72];

  time(&now);
```

```
  p_lt = localtime(&now);
  strftime(buf, sizeof(buf), "%c", p_lt);

  rc = p_writer->Write("now", buf); ❶
  if (rc != 0)
    return (rc);

  // Don't forget to call the superclass implementation
  // so that the remaining parameters can be written.
  return (HttpdDiscoveryServer::BuildResponse(p_writer)); ❷
}
```

❶ We can write any string data we like into the request. Here the time was formatted into a buffer (`buf`) and it is written to the response with a name of `now`.

❷ The last thing is to call the `HttpdDiscoveryServer` implementation. This should always be called if success is returned from `BuildResponse`.

But the `BuildResponse` method is really only part of the story. In order to avoid constantly rebuilding the beacon packets the `PrepareResponse` avoids rebuilding the packet if nothing has changed.

So we need to override `PrepareResponse` and tell it that it needs to rebuild the outbound packet. The criteria used can be more complex than the example below. Any complex logic can decide if the beacon packet needs rebuilding. But the simplest logic (and the logic we will use for our example) is to always rebuild the packet, like this:

```
int DemoDiscoveryServer::PrepareResponse()
{
  mRebuildResponse = true; ❶
  return (HttpdDiscoveryServer::PrepareResponse()); ❷
}
```

❶ We set the value of the protected data member, `mRebuildResponse` which will cause the packet to be rebuilt. Each time the packet is rebuilt this variable is set back to `false`.

❷ A call to the original `PrepareResponse` takes care of all the details. The original implementation checks the `mRebuildResponse` and will eventually result in a call to `BuildResponse`.

Experiment

Compile and run the discovery-2 application and run it on one machine on your network. Then build the `discovery_client` target (you will need the JDK installed for this). Take the `built/`*`PORT`*`/lib/Discovery.jar` file and `src/examples/discovery-2/viewapp.html` to another machine on your network and open `viewapp.html`.

The discovery client should locate the other machine running discovery-2 automatically. The time should update ever few seconds (the refresh rate is configurable in the client).

Chapter 10. Distributed Authoring

Distributed Authoring

HTTP is no longer a read only protocol. With the advent of WebDAV HTTP can be used as a network filesystem. Many computers and devices are capable of accessing storage exposed via WebDAV. Seminole contains a complete WebDAV implementation including locking.

Making a WebDAV enabled server is just a matter of creating an instance of `HttpdWebDAVHandler` and installing it in an instance of `Httpd`. But before we can make a `HttpdWebDAVHandler` we need to configure it. This is done with a structure that we pass to the handler's constructor.

The configuration structure specifies some limits to prevent valid but impractical client requests from making an excessive impact on the system. Here is an example configuration:

```
static const HttpdWebDAVConfiguration gDAVConfig =
{
  HTTPD_WEBDAV_READ_WRITE           |   // Allowed operations.
  HTTPD_WEBDAV_ALLOW_INFINITE_LOCK,     // Infinite lock timeout allowed.
  64,                                   // Max infinite depth.
  HTTPD_CGI_TIMEOUT                     // Put timeout.
#if HTTPD_INC_WEBDAV_LOCKING
  , 128                                 // Max locks.
  , 14400                               // Max lock duration.
#endif
};
```

An important point to note is that some features add additional fields to the configuration structure. So the preprocessor directive allows this declaration to work in any configuration.

With the configuration structure ready, the next step is to create the handler object like so:

```
HttpdWebDAVHandler *p_hand =
  new HttpdWebDAVHandler
  (
    &gDAVConfig,                // Configuration structure.
    HttpdOpSys::NativeFileSystem(), // The filesystem we are exposing.
    "/tmp",                     // The root in the filesystem.
    "/dav"                      // The prefix in URL space.
  );
```

We are creating the handler with `new` but there is no reason it can't be allocated somewhere else. The `HttpdWebDAVHandler` requires an additional initialization step before it can be used:

```
int rc = p_hand->Create();
if (rc != 0)
  failure();
```

Assuming that `rc` is `0` the handler can be installed in a webserver:

```
gpWebServer->Install(phand);
```

Experiment

Compile and run the webdav-1 application to see the above code in action!

Chapter 11. XML

Processing XML

Seminole includes a comprehensive toolkit for dealing with data in XML format. The toolit is layered from low-level (i.e. streaming) interfaces to high-level DOM tree management. Parsing, querying, changing, and serializing data in XML format is very easy when using the DOM API.

The XML toolkit does not even require a web server instance. XML documents can simply be written into an instance of the parser:

```
HttpdXmlHost        host;
HttpdXmlDomBuilder  document(host);
int                 rc;

// Step 1: Initialize the XML parser.
rc = document.Create();
if (rc != 0)
  failure();

// Step 2: Pump in the XML document.
rc = document.WriteString("<document><node>text</node></document>");
if (rc != 0)
  failure();

// Step 3: Complete parsing.
rc = document.Finish();
if (rc != 0)
  failure();
```

Once the document has been parsed into a DOM data structure values can be queried out of the document using a "path-like" syntax:

```
const char *p_node_value = document.Lookup("document/node");
if (p_node_value != NULL)
  printf("The value of the node is: %s\n");
else
  printf("The node was not found.");
```

Using the document parsed in the initial code sample the code fragment above would print:

```
The value of the node is: text
```

Experiment

Compile and run the xml-1 application to see the above code in action!

XML & HTTP

While the Seminole XML toolkit can be used as a standalone component it also features methods designed for easy integration into a `HttpdHandler` implementation.

When processing a `POST` request the `ReadBody` method can process an XML request body in a single step:

```
HttpdXmlHost        host;
HttpdXmlDomBuilder  document(host);
int                 rc;

// Step 1: Initialize the XML parser.
rc = document.Create();
if (rc != 0)
  failure();

// Step 2: Process the request body.
rc = document.ReadBody(p_request);
if (rc != 0)
  failure();
```

The single call to `ReadBody` handles all the mechanics of a request body in XML format.

Experiment

Compile and run the xml-2 application for a comprehensive example of XML processing.

Chapter 12. WebSockets

WebSockets

WebSockets are an extension to HTTP that allow bidirectional message oriented communication between a client and Seminole. This allows both the client and server to push data to each other without polling. This can save on power consumption and CPU utilization.

WebSocket connection requests are negotiated over HTTP. Once connected the socket that was used for HTTP becomes the transport mechanism for WebSocket messages.

To establish a WebSocket connection requires some additional code in your `HttpdHandler::Handle` overridden method. The following sample performs a WebSockets connection:

```
void MyHandler::Handle(HttpdRequest *p_request)
{
  // First ensure this request is for this handler.
  const char *p_suffix = IsMe(p_request);
  if (p_suffix == NULL)
    return (false);

  // It is -- attempt to force a WebSockets connection. The request is
  // failed if it is not a WebSockets connection request.
  //
  // Note that if the HttpdWebSocket::Setup
  // method fails this method returns true since
  // a proper error response is sent by that method.
  HttpdWebSocket sock;
  if (!HttpdWebSocket::Setup(p_req, sock))
    return (true);

  … // Use the socket.

  // Complete the request when processing is complete.
  sock.Close();
  return (true);
}
```

Of course it is not mandatory that a `HttpdHandler` subclass be dedicated to WebSockets. An alternative approach to handling a request involves testing if the request desires a WebSockets connection:

```
void MyHandler::Handle(HttpdRequest *p_request)
{
  // First ensure this request is for this handler.
  const char *p_suffix = IsMe(p_request);
  if (p_suffix == NULL)
    return (false);

  // It is -- if it is not a WebSockets request then we handle
  // this request “normally.”
```

```
  if (!HttpdWebSocket::IsRequest(p_request))
  {
    … // Handle as a normal HTTP request.
    return (true);
  }

  // It must be a WebSockets request. Attempt to perform a connection.
  // Note that if Connect returns failure then a proper
  // error response has been sent and true is returned.
  HttpdWebSocket sock;
  if (!HttpdWebSocket::Connect(p_req, sock))
    return (true);

  … // Use the socket.

  // Complete the request when processing is complete.
  sock.Close();
  return (true);
}
```

Once connect a `HttpdWebSocket` object can be used to send messages back and forth:

```
for(;;)
{
  // Get the message to send.
  HttpdWebSocket::Message update;
  … // Fill in update with a message to send.

  // Send a status update.
  if (sock.Send(update) != 0)
    return (true);

  // To handle PINGs and check the state of the connection block waiting
  // for a message from the client.
  HttpdWebSocket::Message rx_message;
  switch (sock.Received(rx_message, 1)) // Receive with a 1 second timeout.
  {
    case 0:
      … // Process a message from the client.
      sock.Finish(rx_message);
      // Fallthrough:

    case HttpdOpSys::ERR_NOTREADY:
      … // Handle a timeout.
      continue;

    default:
      … // Handle other errors.
      sock.Close();
      return (true);
  }
}
```

Experiment

Compile and run the websockets-1 application to see the above code in action!

WebSockets (Multiplexed Waiting)

Each WebSocket connection is serviced by a Seminole worker thread. The intention being that these threads spend most of their time blocked waiting for a message from the WebSocket client in `HttpdWebSocket::Received` waiting for an incoming message. However this precludes the ability for the handler to perform other tasks without polling.

If the underlying operating system supports waiting for socket I/O alongside some other object then polling may be avoided. This is especially important for limited power applications. Limited power applications are ideal for the push model of WebSockets when combined with multiplexed I/O.

Experiment

If you have a supported target operating system compile and run the websockets-2 application to see multiplexing in action!

Chapter 13. Debugging

Debugging software is hard. Debugging software in an embedded environment is harder. Network-enabled embedded systems are even harder still. To simplify debugging, Seminole includes a basic tracing facility that highlights the activity of the server. These traces provide an overview of the workings of Seminole without requiring breakpoints or much knowledge of the internals of the webserver.

Tracing

It is important to understand the difference between tracing and logging. Tracing is meant only as a debugging tool and should not be activated except in the event of discovering a problem. It is commonplace to enable tracing when integrating Seminole with your application. However, once operational it should be turned off completely to reduce overhead.

Tracing is enabled by the INC_TRACING build option. Therefore adding the following line to the port file (before the `definitions` statement) will enable tracing:

```
config(INC_TRACING => 1);
```

By default all tracing output is sent to standard out using the `printf` family of standard I/O routines. The format of the trace output depends on the value of the `HTTPD_HAVE_CLOCK` preprocessor macro. This macro is normally defined by the portability layer if the platform has a real-time clock capability. If this capability is enabled the trace entries will include timestamps.

Below is an example trace of the possible output from the file-1 example. The first column of the output is the timestamp (or sequence number). The second column is the source file name issuing the trace, followed by the line number. The 3-digit character string following the line number indicates the kind of the trace entry, followed by the actual data of the trace entry.

Trace entries can nest just as subroutine calls nest. In fact, a trace entry is made by constructing a "trace object" on the stack of the functions that perform tracing. The construction and destruction of these objects are recorded as entry and exit events in the trace log.

```
[00:21:26] sem_httpd.cpp 366 >>> Enter ❶
[00:21:26] sem_httpd.cpp 367 *** Installing handler ❷
[00:21:26] sem_httpd.cpp 368 ### p_handler->Prefix() = / ❸
[00:21:26] sem_httpd.cpp 366 <<< Exit ❹
Server started on host localhost port 8080 ❺
[00:21:26] sem_httpd.cpp 276 >>> Enter
[00:21:26] sem_httpd.cpp 290 *** Calling HttpdSocket::Initialize
[00:21:26] sem_httpd.cpp 294 *** Creating listening socket
[00:21:26] sem_httpd.cpp 125 >>> Enter
[00:21:26] sem_httpd.cpp 155 *** Putting socket in listen mode
[00:21:26] sem_httpd.cpp 125 <<< Exit
[00:21:26] sem_httpd.cpp 298 *** Starting acceptor task
[00:21:26] sem_httpd.cpp 403 >>> Enter
[00:21:26] sem_httpd.cpp 404 *** Starting acceptor thread
[00:21:26] sem_httpd.cpp 405 ### p_server: 00263F80 ❻
```

```
[00:21:26] sem_httpd.cpp 307 *** Server started
[00:21:26] sem_httpd.cpp 276 <<< Exit
…
```

❶ This is a function entry trace event. All further trace entries until the next entry or exit refer to the routine that initiated the trace.
❷ This is a note explicitly in the code for debugging purposes. Often times it is indicative of an operational decision or status value.
❸ This is a value within the code that is being logged for debugging purposes. In this case it is the result of the `HttpdHandler::Prefix` method. The value returned in this case is for the root of the URI space.
❹ This indicates that the matching trace entry event is now closed. The routine that created the trace entry is exiting.
❺ Notice that trace output is intermixed with other data to standard out. If it is desired that trace output be sent somewhere else the code in `common/sem_tracing.cpp` must be modified accordingly.
❻ Tracing will sometimes yield the address of important data structures. This is useful when setting breakpoints. For example, knowing the request object address, the entire request object can be displayed while at a breakpoint in gdb with this command:

```
(gdb) print *(HttpdRequest *)0x263F80
```

Debugging

Tracing is useful when your handler isn't getting called or if the server can't initialize. There are, of course, the harder problems that require more sophisticated debugging techniques. Often times this means producing a debug build of Seminole. This is accomplished by adding the following line to the port file:

```
$DEBUG=1;
```

Setting the `$DEBUG` build variable causes the compiler to build debug code (if supported by the target platform) and enables the `assert` calls in Seminole.

Having a good starting strategy for debugging is very important, especially in a complex mutli-threaded software package like Seminole. If the problem is reproduceable then the best approach is to set a breakpoint in the appropriate `HttpdHandler::Handle` method and step through the operation at a high level until the problem can be pinpointed.

Another common area of difficulty is in the various system calls made by the portability layers. Seminole is tested on a variety of embedded platforms and real time operating systems. However, with other third party code and tasks running it may expose previously dormant problems in the way an operating system interacts with Seminole. Worse, when a new CPU is being used often times subtle bugs can even exist in the compiler and linker.

The important thing to remember when solving these kinds of problems is to locate the problem realizing that it may be in what was thought to be well-tested and correct hardware or software. It is easier to locate the problem if there is less complexity in the system. To that end turning off various Seminole features via build options may provide a clue as to the nature of a problem. Of course, just because turning off a build option makes a problem go away does not mean the problem is fixed.

Fixing symptoms instead of problems will only cause more problems later on down the road. It is important that during the integration of Seminole (and other third party packages) in your embedded system, problems be understood and solved as early as possible.

Appendix A. Structure of a URL

Instead of going over the general format for a URL this appendix instead will cover the format of URLs used by HTTP requests, namely the HTTP URL. It's primarily composed of three portions, the scheme, the host and the path.

```
"http:"
"//" <host> [ ":" <port> ]
"/" <path> [ "?" <query> ] [ "#" <fragment> ]
```

The scheme portion of the URL consists of the letters "http" and a colon. This portion is case insensitive, but is typically written in lower case.

```
 http:
```

The host portion consists of two forward slashes, the hostname or IP address of the server followed by an optional colon and port number. The hostname is case insensitive and can only contain letters, numbers and the dash (this being a DNS restriction) and the port number is an unsigned value between 0 and 65,535. If the port number is *not* given, then port 80 is assumed. The use of a username and password preceding the hostname is *not* allowed for HTTP URLs, even though other net based schemes such as FTP allow such use.

```
//www.example.net
//WWW.Example.NET
//www.example.net:8080
//Www.EXAMPLE.NeT:8080
//192.168.10.10
//192.168.10.10:8080
```

The last portion, the path, is itself made up of three portions, the path segment, the query and the fragment and is case *sensitive*. Using a BNF-like grammar, where rules are separated by their definition with a "=", indentation is used to continue a rule over multiple lines, parentheses are used to group elements, optional elements are enclosed in square brackets, and items marked with an asterisk can repeat zero or more times, this portion of the URL looks like:

```
path            = "/" *path_segments [ "?" query ] [ "#" fragment ]
path_segment    = segment *( "/" segment )
segment         = *pchar *( ";" param )
param           = *pchar
query           = pair [ "&" pair ]
pair            = token "=" *token
token           = *unreserved
fragment        = reserved | unreserved | escape

pchar           = unreserved | escaped |
                  ":" | "@" | "&" | "=" | "+" | "$" | ","
```

```
reserved        = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+" |
                  "$" | ","

unreserved      = alphanum | mark

escaped         = "%" hex hex
hex             = digit | "A" | "B" | "C" | "D" | "E" | "F"
                  "a" | "b" | "c" | "d" | "e" | "f"
mark            = "-" | "_" | "." | "!" | "~" | "*" | "'" |
                  "(" | ")"

alpha           = lowalpha | upalpha

lowalpha        = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
                  "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
                  "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

upalpha         = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
                  "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
                  "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

digit           = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
                  "8" | "9"
```

The path segments are used to organize the resources (web pages, graphics, documents, sounds and any other digital format to be served) and while it has a superficial syntax to that of modern hierarchical file systems, that doesn't mean it has to map one-to-one to files on a filesystem; in many cases a URL doesn't even map to a file but instead to a series of modules or components that generate the requested resource on the fly at the time that the request is made.

Some examples of URLs:

```
http://www.example.net/
http://WWW.Example.NET:80/
```

These two refer to the same resource and are equivalent. Remember, the host portion is case insensitive, and the default port for HTTP is 80.

```
http://www.example.net/corp/aboutus/CEO.html
```

A typical example of a URL. Because the path portion of the URL is case *sensitive* the following two URLs are technically different:

```
http://www.example.net/corp/aboutus/CEO.html
http://www.example.net/corp/aboutus/ceo.html
```

Remember—the host portion is case *sensitive* while the path portion is case *insensitive*. Some web servers (notably the one that runs the AOL webserver) treat the path portion as case insensitive; that technically is breaking the standard.

```
http://www.example.net/corp;version3/aboutus;current/CEO.html
```

Here we see the parameter values in the path segment; in this example, it is assumed that the webserver uses the parameter portion to denote a particular version of each path segment, so you have “version 3” of the “corp” path segment, and the “current” version of the “aboutus” path segment.

In reality, the parameter portion of path segments are rarely, if ever, used. They're defined though, and *could* be used in this manner; the actual use of a parameter in a path segment is implementation dependent.

```
http://www.example.net/corp/aboutus/CFO.html#picture
```

Here the URL includes a fragment, pointing to a named location in the page. The fragment portion is never (or *should* never) be sent to the webserver; the client uses it to position the page in the browser such that the named location is visible to the user.

```
http://s.example.org/prod/toy/list?s=yes&c=no
```

This example includes a query, which is comprised to two name/value pairs.

```
http://s.Exmaple.ORG:8080/prod%24;3/toy;kids%20things/list;3.2?stocked=yes&color
```

This is an example that contains all possible portions of a URL hostname, port number, path segments with parameters, a query and a fragment specifier, plus a few escaped characters.

See Appendix B for information on how HTTP URLs are used to request pages from a web server.

Appendix B. A Brief Overview of HTTP

This appendix is just a brief overview of HTTP—for a more indepth look at the protocol, you are encouraged to read O'Reilly's *The Essential Guide to HTTP* and for the final say in things, IETF RFC-2616.

HTTP is a text-based protocol, meaning that the actual messages exchanged between the client (typically a web browser) and the server (Apache, or in this case, Seminole) are in human readable lines of text. The underlying protocols (TCP/IP) provide the actual transport of a reliable byte (octet) oriented stream between the client and server.

There are currently three versions of HTTP in existence—0.9, 1.0 and 1.1. This overview primarily covers versions 1.0 and above as version 0.9 is deprecated and should not be used any more (although Seminole does include support for it for legacy clients). The differences between 0.9 and later versions will be covered at the end of this appendix.

The Basic Protocol

The best way to explain the protocol is to show how a request is made by a client (a web browser) to the server. Given the following URL:

```
http://www.example.net:8080/corp/aboutus/CEO.html
```

a browser will typically pick it apart as:

host:	`www.example.net`
port:	`8080`
path:	`/corp/aboutus/CEO.html`

The client will then establish a TCP/IP connection on the given port (in this example, port 8080). Once the connection has been established, the client will then send the request (and since HTTP is a text-based protocol, it's human readable):

```
GET /corp/aboutus/CEO.html HTTP/1.0\r\n
Host: www.example.net:8080\r\n
Accept: text/html, text/xml, text/*; q=0.5,\r\n
        image/jpeg, image/png; q=0.8, image/gif; q=0\r\n
        */*; q=0.2\r\n
Accept-Encoding: gzip; q=1.0, identity; q=0.5, *;q=0\r\n
Referer: http://www.example.org/other/links.html\r\n
User-Agent: X-zilla/8.0 (a new experience in browsing)\r\n
\r\n
```

The first line is the method being used to retrieve the resource, the path of the resource and the protocol version being used. The rest of the lines, called “headers,” are additional information given by the client to the server and most are optional (there are some that are mandatory in `HTTP/1.1`, like the `Host:` line, but most clients that speak `HTTP/1.0` include this line). In this instance, the lines are:

Host	Which website (or domain) we are obtaining the resource from. With this, it is possible to serve multiple websites from a single IP address.
Accept	Media types that the client supports, and the preferences. In the example above, media types of text/html and text/xml are the most preferred text types, although any text type (if available) will be accepted. Of image types, the preferred type are JPEGs, then PNGs; images of type GIF are *not* to be served. All other types of media are accepted.
Accept-Encoding	Formats the client prefers the data in. If at all possible, compress using the gzip format, otherwise, just send the data as is. Other encoding formats are not supported or wanted.
Referer	The client followed a link from the given resource to make this request.
User-Agent	This is how the client identifies itself. Don't expect much truth to this field, nor any real format.

The request is terminated with a blank line (signified by the lone "\r\n"). Once the request is made, the web server, using the information given, will find (or generate) the requested resource and return a response:

```
HTTP/1.1 200 Okay\r\n
Content-Type: text/html\r\n
Content-Length: 3306\r\n
Date: Tue, 22 Feb 2004 06:21:17 GMT\r\n
Server: Seminole\r\n
Last-Modified: Fri, 16 May 2003 07:50:43 GMT\r\n
\r\n
<html>…more content from page
```

The response from the server starts with a response code (in this case, a return code of 200, which indicates success), then additional data about the request, a blank line indicating the end of the header section of the response, then the actual data of the resource being requested.

Content-Type	The type of data being returned, in this case, an HTML document.
Content-Length	The number of bytes being sent in the response body.
Date	The current date
Server	The server software
Last-Modified	The last time the resource was changed.

Methods

In the example exchange above, the client program sent

```
GET /corp/aboutus/CEO.html HTTP/1.0\r\n
```

This is the most common request type made—the `GET` method. The intent of the `GET` method is to simply retrieve the resource in question and is intended to be a "safe"—i.e. not change the status of the resource on the server. There are other methods defined in HTTP, such as `HEAD`, which differs from `GET` (or should only differ) in that only the headers of the request should be returned and not the resource itself. This method is typically used to check if a resource has changed since it was last requested. But other than the lack of the resource not being sent, there *should* not be any difference between a `GET` request and a `HEAD` request.

Another common method is `POST`, which is used when a user fills out an HTML form and submits the data to the server. The use of `POST` typically means additional processing on the part of the server and some side effect (a resource generated, email sent off, an order being saved to a database) has occurred. This is also one of two commands (currently) were data is sent *to* the server from the client, and is almost always directed towards a CGI script running on the server. A typical `POST` request usually looks like:

```
POST /cgi-bin/search HTTP/1.1\r\n
Host: search.example.net\r\n
Content-Type: application/x-www-form-urlencoded\r\n
Content-Length: 54\r\n
Accept: text/html, text/xml, text/*; q=0.5,\r\n
        image/jpeg, image/png; q=0.8, image/gif; q=0\r\n
        */*; q=0.2\r\n
Accept-Encoding: gzip; q=1.0, identity; q=0.5, *;q=0\r\n
Referer: http://www.example.org/other/links.html\r\n
User-Agent: X-zilla/8.0 (a new experience in browsing)\r\n
\r\n
query=%22Seminole+sales+price%22+%2BWindows&lang=en-US
```

Note that the client is sending data (which starts after the blank line), and that the headers include the content type and length.

There are other methods, described in the sections that follow.

HTTP Response codes

The response codes for HTTP are three digits long and are divided into five major categories:

1xx—Informational based codes, and only applicable to HTTP/1.1 and above (see Appendix B, section 4.2).

2xx—Success, the request can be fulfilled. The most common response code is "200".

3xx—Redirection. The resource requested is not currently available at the given location, but instead is located elsewhere, the location given by the server in the Location: header (which is an absolute URL to the new location). There are distinctions between a temporary move in location (302) and permanent change in location (301).

4xx—Client error. The request could not be fulfilled for some reason. The second most common response code is the dreaded 404—the resource requested could not be found.

5xx—Server error. The webserver itself had problems fulfilling the request.

The actual response codes are defined in the sections below.

The different versions of HTTP

Beginning with version 1.0, HTTP requests are both backward and forward compatible—any headers either the client or the server don't understand are simply ignored and behavior will fall back towards the earlier version of the protocol. Response codes are three digits, and are grouped such that similar responses always start with the same digit (such that successful responses will always start with “2” and that client errors will always start with “4”) so that while an older browser might not understand the exact meaning of say, “444” (which isn't currently defined) but it will understand that a request was not successful and that it was the request itself that was in error.

HTTP/1.0

All headers are optional, and the minimum request required is

```
<method> <path> HTTP/1.0\r\n
\r\n
```

The only methods defined for HTTP/1.0 are `GET`, `HEAD` and `POST`.

Also, each request made requires a separate connection to the webserver, so a page that consists of HTML plus three graphics requires four separate TCP/IP connections (although the client can certainly do four parallel connections at the same time).

The response codes defined for HTTP/1.0 are:

200	The resource was found and is being returned
201	The resource was created
202	The request has been accepted for processing, but the processing has not been completed. See RFC-2616 section 10.2.3 for more information.
204	The server has fulfilled the request, but there is no resource to return.

301	The resource has been moved to a new location permanently, and any further request *should* be made to the new location.
302	The resource has been moved temporarily to a new location, and further requests should be made at the *original* location.
304	The resource has not been modified since the last request, so the previous copy, if cached, can be used.

400	The client made a syntax error in making a request. This is typically the case when a new browser is being written, or a human is attempting to make a request by hand.
401	The client did not include the proper `Authorization:` header to make the request.

	On most browsers, this will pop up a user/password message box for the user to fill out and retry the request.
403	The request is to a valid resource, but the server is not serving it up and an `Authorization:` header will not help; usually the permissions to the resource have been revoked in some way from the webserver.
404	The resource requested could not be found on the server.
500	The server encountered some internal condition that is preventing it from continuing, such as an out-of-memory condition.
501	The server has not implemented the method being used to request the resource.
502	Bad Gateway, see RFC-2616 section 10.5.3 for more information.
503	The server is currently out of service, due to being overloaded or maintenance. This *should* be a temporary situation so the request should be attempted at a later time.

HTTP/1.1

All headers with the exception of `Host:` are optional, so the minimum request required is:

```
<method> <path> HTTP/1.1\r\n
Host: www.example.net\r\n
\r\n
```

There are several new methods defined for HTTP 1.1 which allow one to create new resources, delete existing resources and a trace mechanism for troubleshooting proxy problems; these methods are beyond the scope of this document and those curious can check out RFC-2616 or O'Reilly's *Definitive Guide to HTTP* for more information on these methods.

HTTP 1.1 also allows multiple requests to be made over a single TCP/IP connection, so that a page that contains three graphics can be transferred over a single connection; each request being made over the same TCP/IP connection once the previous one has finished.

There are also mechanisms within HTTP 1.1 to resume transfers, or to send the requested range (in bytes) of a resource.

HTTP 1.1 also defines several more response codes:

100	The client can continue with sending a response body to the given request. See RFC-2616 section 8.2.3 for more information.
101	The server is able to switch to a different protocol as requested by the client. See RFC-2616 section 10.1.2.

203	Used by web proxies to inform the client that the information about the resource is not the definitive copy from the original server, but was gathered from other sources.
205	The server has fulfilled the request and the client should reset the document (say, clear out any form elements) that is being displayed to the user.
206	The server is sending back a partial response as requested.

300	The requested resource is stored in multiple locations, possibly in different formats; the client is requested to select an appropriate one.
303	The resource can be found in a different location and should be retrieved at the new location using `GET`. This is the preferred response if a result of a `POST` is a redirection to a new resource that must use `GET`.
305	The requested resource must be obtained through a proxy at the given location.

402	Reserved for future use (payment is required)
405	The method used to request the resource is not allowed.
406	The resource cannot be fulfilled because the resource is not in a format acceptable to the client.
407	The client did not provide sufficient credentials to use the proxy server.
408	The client timed out making the request.
409	The resource could not be fulfilled due to a conflict (see RFC-2616 section 10.4.10)
410	The resource requested used to exist, but is now permanently gone.
411	The server can't complete the request unless the length of the request (for instance, a POST method) is given.
412	The constraints (preconditions) for the resource cannot be met by the server.
413	The request is too large to handle by the server. This may be a permanent or temporary condition.
414	The URL given by the client is too long for the server to handle.
415	The client made a request in a format that is not supported for that method by the server.
416	The requested range made by the client is invalid.
417	The expectation given by the Expect request-header field could not be met by the server.

504	The server, acting as a proxy, received a timeout from an upstream server.
505	The server does not support the HTTP protocol version that was made in the request.

HTTP/0.9

HTTP 0.9 was the original protocol developed by Tim Berners-Lee and did not have any support for protocol version, content negotiation or any form of metadata about the requested resource. This protocol version is *very* simple and not robust at all. A request is made:

```
GET /corp/aboutus/CEO.html\r\n
```

Note: there is no blank line following the request, nor a protocol version nor any other information about the request. Then the server responds immediately with the resource:

```
<html>…more content from page
```

No response code, no additional information about the request, and it's up to the client to figure out the media type of the response (if a request for an image was made and no such image existed, the server could send back HTML to describe the result when the client was not expecting HTML).

Seminole includes support for HTTP 0.9 only for legacy clients; new clients should not use this version for requests.