Contents

About This Guide ................................................................. 13
  Who Should Use This Guide ............................................. 13
  Using the Documentation .................................................. 14
  How This Guide Is Organized ............................................ 16
  Documentation Conventions .............................................. 18
  Product Support .............................................................. 19

Chapter 1 Syntax and Use of obj.conf ........................................ 21
  How the Server Handles Requests from Clients ...................... 22
    HTTP Basics .............................................................. 22
    NSAPI Filters .......................................................... 24
    Steps in the Request-handling Process .............................. 24
    Directives for Handling Requests .................................. 25
  Dynamic Reconfiguration ................................................. 26
  Server Instructions in obj.conf ........................................ 26
    Summary of the Directives .......................................... 27
  Configuring HTTP Compression .......................................... 30
  The Object and Client Tags .............................................. 32
    The Object Tag ......................................................... 32
      Objects that Use the name Attribute ............................ 33
      Objects that Use the ppath Attribute ........................... 33
    The Client Tag ....................................................... 34
    Client Tag Parameters ............................................... 34
  Variables Defined in server.xml ....................................... 37
  Flow of Control in obj.conf ............................................ 37
    AuthTrans .............................................................. 38
    NameTrans .............................................................. 38
<table>
<thead>
<tr>
<th>Chapter 4 Creating Custom Filters</th>
<th>103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future Compatibility Issues</td>
<td>103</td>
</tr>
<tr>
<td>The NSAPI Filter Interface</td>
<td>104</td>
</tr>
<tr>
<td>Filter Methods</td>
<td>104</td>
</tr>
<tr>
<td>C Prototypes for Filter Methods</td>
<td>104</td>
</tr>
<tr>
<td>insert</td>
<td>105</td>
</tr>
<tr>
<td>remove</td>
<td>106</td>
</tr>
<tr>
<td>flush</td>
<td>106</td>
</tr>
<tr>
<td>read</td>
<td>106</td>
</tr>
<tr>
<td>write</td>
<td>107</td>
</tr>
<tr>
<td>writev</td>
<td>107</td>
</tr>
<tr>
<td>sendfile</td>
<td>107</td>
</tr>
<tr>
<td>Position of Filters in the Filter Stack</td>
<td>108</td>
</tr>
<tr>
<td>Filters that Alter Content-Length</td>
<td>110</td>
</tr>
<tr>
<td>Creating and Using Custom Filters</td>
<td>111</td>
</tr>
<tr>
<td>Write the Source Code</td>
<td>111</td>
</tr>
<tr>
<td>Compile and Link</td>
<td>112</td>
</tr>
<tr>
<td>Load and Initialize the Filter</td>
<td>112</td>
</tr>
<tr>
<td>Instruct the Server to Insert the Filter</td>
<td>113</td>
</tr>
<tr>
<td>Restart the Server</td>
<td>114</td>
</tr>
<tr>
<td>Test the Filter</td>
<td>114</td>
</tr>
<tr>
<td>Overview of NSAPI Functions for Filter Development</td>
<td>114</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5 Examples of Custom SAFs and Filters</th>
<th>115</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples in the Build</td>
<td>116</td>
</tr>
<tr>
<td>AuthTrans Example</td>
<td>117</td>
</tr>
<tr>
<td>Installing the Example</td>
<td>117</td>
</tr>
<tr>
<td>Source Code</td>
<td>118</td>
</tr>
<tr>
<td>NameTrans Example</td>
<td>119</td>
</tr>
<tr>
<td>Installing the Example</td>
<td>120</td>
</tr>
<tr>
<td>Source Code</td>
<td>120</td>
</tr>
<tr>
<td>PathCheck Example</td>
<td>123</td>
</tr>
<tr>
<td>Installing the Example</td>
<td>123</td>
</tr>
<tr>
<td>Source Code</td>
<td>124</td>
</tr>
<tr>
<td>ObjectType Example</td>
<td>126</td>
</tr>
<tr>
<td>Installing the Example</td>
<td>127</td>
</tr>
<tr>
<td>Source Code</td>
<td>127</td>
</tr>
</tbody>
</table>
Chapter 6 Creating Custom Server-parsed HTML Tags ........................................... 149
Define the Functions that Implement the Tag ..................................................... 150
Write an Initialization Function ................................................................. 154
Load the New Tag into the Server ............................................................... 154

Chapter 7 NSAPI Function Reference ................................................................. 155
NSAPI Functions (in Alphabetical Order) ...................................................... 155
   CALLOC ................................................................. 156
cinfo_find ................................................................. 156
condvar_init ................................................................. 157
condvar_notify ................................................................. 158
condvar_terminate ............................................................... 158
condvar_wait ................................................................. 159
crit_enter ................................................................. 159
crit_exit ................................................................. 160
crit_init ................................................................. 160
crit_terminate ............................................................... 161
daemon_atrestart ............................................................ 161
fc_open ................................................................. 162
fc_close ................................................................. 163
filebuf_buf2sd .............................................................. 163
filebuf_close .............................................................. 164
filebuf_getc .............................................................. 165
filebuf_open .............................................................. 165
filebuf_open_nostat .......................................................... 166
filter_create .............................................................. 167
filter_find .............................................................. 168
filter_insert .............................................................. 169
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter_layer</td>
<td>169</td>
</tr>
<tr>
<td>filter_name</td>
<td>170</td>
</tr>
<tr>
<td>filter_remove</td>
<td>170</td>
</tr>
<tr>
<td>flush</td>
<td>171</td>
</tr>
<tr>
<td>FREE</td>
<td>172</td>
</tr>
<tr>
<td>func_exec</td>
<td>172</td>
</tr>
<tr>
<td>func_find</td>
<td>173</td>
</tr>
<tr>
<td>func_insert</td>
<td>173</td>
</tr>
<tr>
<td>insert</td>
<td>174</td>
</tr>
<tr>
<td>log_error</td>
<td>175</td>
</tr>
<tr>
<td>MALLOC</td>
<td>176</td>
</tr>
<tr>
<td>net_flush</td>
<td>177</td>
</tr>
<tr>
<td>net_ip2host</td>
<td>177</td>
</tr>
<tr>
<td>net_read</td>
<td>178</td>
</tr>
<tr>
<td>net_sendfile</td>
<td>179</td>
</tr>
<tr>
<td>net_write</td>
<td>180</td>
</tr>
<tr>
<td>netbuf_buf2sd</td>
<td>181</td>
</tr>
<tr>
<td>netbuf_close</td>
<td>181</td>
</tr>
<tr>
<td>netbuf_getc</td>
<td>182</td>
</tr>
<tr>
<td>netbuf_grab</td>
<td>182</td>
</tr>
<tr>
<td>netbuf_open</td>
<td>183</td>
</tr>
<tr>
<td>nsapi_module_init</td>
<td>183</td>
</tr>
<tr>
<td>NSAPI_RUNTIME_VERSION</td>
<td>184</td>
</tr>
<tr>
<td>NSAPI_VERSION</td>
<td>185</td>
</tr>
<tr>
<td>param_create</td>
<td>185</td>
</tr>
<tr>
<td>param_free</td>
<td>186</td>
</tr>
<tr>
<td>pblock_copy</td>
<td>186</td>
</tr>
<tr>
<td>pblock_create</td>
<td>187</td>
</tr>
<tr>
<td>pblock_dup</td>
<td>187</td>
</tr>
<tr>
<td>pblock_find</td>
<td>188</td>
</tr>
<tr>
<td>pblock_findval</td>
<td>189</td>
</tr>
<tr>
<td>pblock_free</td>
<td>189</td>
</tr>
<tr>
<td>pblock_nniinsert</td>
<td>190</td>
</tr>
<tr>
<td>pblock_nvinsert</td>
<td>190</td>
</tr>
<tr>
<td>pblock_pb2env</td>
<td>191</td>
</tr>
<tr>
<td>pblock_pblock2str</td>
<td>191</td>
</tr>
<tr>
<td>pblock_pinsert</td>
<td>192</td>
</tr>
<tr>
<td>pblock_remove</td>
<td>193</td>
</tr>
<tr>
<td>pblock_str2pblock</td>
<td>193</td>
</tr>
<tr>
<td>PERM_CALLOC</td>
<td>194</td>
</tr>
<tr>
<td>PERM_FREE</td>
<td>195</td>
</tr>
<tr>
<td>PERM_MALLOC</td>
<td>195</td>
</tr>
<tr>
<td>PERM_REALLOC</td>
<td>196</td>
</tr>
</tbody>
</table>
PERM_STRDUP .................................................. 197
prepare_nsapi_thread ........................................ 197
protocol_dump822 ............................................. 198
protocol_set_finfo ........................................... 199
protocol_start_response ...................................... 199
protocol_status .............................................. 200
protocol_uri2url ............................................. 201
protocol_uri2url_dynamic .................................... 202
read .......................................................... 203
REALLOC ....................................................... 204
remove ......................................................... 204
request_get_vs ............................................... 205
request_header ............................................... 205
request_stat_path ............................................ 206
request_translate_uri ....................................... 207
sendfile ....................................................... 208
session_dns ................................................... 209
session_maxdns ............................................... 209
shexp_casecmp ............................................... 210
shexp_cmp .................................................... 210
shexp_match .................................................. 211
shexp_valid .................................................. 212
STRDUP ......................................................... 213
system_errno .................................................. 213
system_fclose ............................................... 214
system_flock .................................................. 215
system_fopenRO .............................................. 215
system_fopenRW .............................................. 216
system_fopenWA .............................................. 216
system_fread .................................................. 217
system_fwrite ................................................ 217
system_fwrite_atomic ....................................... 218
system_gmtime ............................................... 219
system_localtime ............................................ 219
system_lseek .................................................. 220
system_rename ................................................ 221
system_ulock .................................................. 221
system_unix2local .......................................... 222
systhread_attach ............................................. 222
systhread_current ............................................ 223
systhread_getdata ........................................... 223
systhread_newkey ............................................ 224
systhread_setdata ........................................... 224
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>systhread_sleep</td>
<td>225</td>
</tr>
<tr>
<td>systhread_start</td>
<td>225</td>
</tr>
<tr>
<td>systhread_timerset</td>
<td>226</td>
</tr>
<tr>
<td>USE_NSAPI_VERSION</td>
<td>226</td>
</tr>
<tr>
<td>util_can_exec</td>
<td>228</td>
</tr>
<tr>
<td>util_chdir2path</td>
<td>228</td>
</tr>
<tr>
<td>util_chdir2path</td>
<td>229</td>
</tr>
<tr>
<td>util_cookie_find</td>
<td>229</td>
</tr>
<tr>
<td>util_env_find</td>
<td>230</td>
</tr>
<tr>
<td>util_env_free</td>
<td>230</td>
</tr>
<tr>
<td>util_env_replace</td>
<td>231</td>
</tr>
<tr>
<td>util_env_str</td>
<td>231</td>
</tr>
<tr>
<td>util_getline</td>
<td>232</td>
</tr>
<tr>
<td>util_hostname</td>
<td>232</td>
</tr>
<tr>
<td>util_is_mozilla</td>
<td>233</td>
</tr>
<tr>
<td>util_is_url</td>
<td>233</td>
</tr>
<tr>
<td>util_jtoa</td>
<td>234</td>
</tr>
<tr>
<td>util_later_than</td>
<td>234</td>
</tr>
<tr>
<td>util_sh_escape</td>
<td>235</td>
</tr>
<tr>
<td>util_snprintf</td>
<td>235</td>
</tr>
<tr>
<td>util_sprintf</td>
<td>236</td>
</tr>
<tr>
<td>util_strcasecmp</td>
<td>237</td>
</tr>
<tr>
<td>util_strftime</td>
<td>237</td>
</tr>
<tr>
<td>util_strncasecmp</td>
<td>237</td>
</tr>
<tr>
<td>util_uri_escape</td>
<td>238</td>
</tr>
<tr>
<td>util_uri_is_evil</td>
<td>238</td>
</tr>
<tr>
<td>util_uri_parse</td>
<td>239</td>
</tr>
<tr>
<td>util_uri_unescape</td>
<td>239</td>
</tr>
<tr>
<td>util_vsprintf</td>
<td>239</td>
</tr>
<tr>
<td>util_vsnprintf</td>
<td>240</td>
</tr>
<tr>
<td>util_cookie_find</td>
<td>240</td>
</tr>
<tr>
<td>util_chdir2path</td>
<td>241</td>
</tr>
<tr>
<td>util_env_str</td>
<td>242</td>
</tr>
<tr>
<td>vs_alloc_slot</td>
<td>243</td>
</tr>
<tr>
<td>vs_get_data</td>
<td>243</td>
</tr>
<tr>
<td>vs_get_default_httpd_object</td>
<td>244</td>
</tr>
<tr>
<td>vs_get_doc_root</td>
<td>244</td>
</tr>
<tr>
<td>vs_get_httpd_objset</td>
<td>245</td>
</tr>
<tr>
<td>vs_get_id</td>
<td>245</td>
</tr>
<tr>
<td>vs_get_mime_type</td>
<td>246</td>
</tr>
<tr>
<td>vs_lookup_config_var</td>
<td>246</td>
</tr>
<tr>
<td>vs_register_cb</td>
<td>247</td>
</tr>
<tr>
<td>vs_set_data</td>
<td>247</td>
</tr>
<tr>
<td>vs_translate_uri</td>
<td>248</td>
</tr>
<tr>
<td>write</td>
<td>248</td>
</tr>
<tr>
<td>writev</td>
<td>249</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>Data Structure Reference</td>
</tr>
<tr>
<td></td>
<td>Privatization of Some Data Structures</td>
</tr>
<tr>
<td></td>
<td>Session</td>
</tr>
<tr>
<td></td>
<td>pbblock</td>
</tr>
<tr>
<td></td>
<td>pb_entry</td>
</tr>
<tr>
<td></td>
<td>pb_param</td>
</tr>
<tr>
<td></td>
<td>Session-&gt;client</td>
</tr>
<tr>
<td></td>
<td>Request</td>
</tr>
<tr>
<td></td>
<td>stat</td>
</tr>
<tr>
<td></td>
<td>shmem_s</td>
</tr>
<tr>
<td></td>
<td>cinfo</td>
</tr>
<tr>
<td></td>
<td>sendfiledata</td>
</tr>
<tr>
<td></td>
<td>Filter</td>
</tr>
<tr>
<td></td>
<td>FilterContext</td>
</tr>
<tr>
<td></td>
<td>FilterLayer</td>
</tr>
<tr>
<td></td>
<td>FilterMethods</td>
</tr>
<tr>
<td></td>
<td>FilterMethods</td>
</tr>
<tr>
<td>9</td>
<td>Using Wildcard Patterns</td>
</tr>
<tr>
<td></td>
<td>Wildcard Patterns</td>
</tr>
<tr>
<td></td>
<td>Wildcard Examples</td>
</tr>
<tr>
<td>10</td>
<td>Time Formats</td>
</tr>
<tr>
<td>11</td>
<td>Dynamic Results Caching Functions</td>
</tr>
<tr>
<td></td>
<td>dr_cache_destroy</td>
</tr>
<tr>
<td></td>
<td>dr_cache_init</td>
</tr>
<tr>
<td></td>
<td>dr_cache_refresh</td>
</tr>
<tr>
<td></td>
<td>dr_net_write</td>
</tr>
<tr>
<td></td>
<td>fc_net_write</td>
</tr>
<tr>
<td>12</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td></td>
<td>Compliance</td>
</tr>
<tr>
<td></td>
<td>Requests</td>
</tr>
<tr>
<td></td>
<td>Request Method, URI, and Protocol Version</td>
</tr>
<tr>
<td></td>
<td>Request Headers</td>
</tr>
<tr>
<td></td>
<td>Request Data</td>
</tr>
<tr>
<td></td>
<td>Responses</td>
</tr>
<tr>
<td></td>
<td>HTTP Protocol Version, Status Code, and Reason Phrase</td>
</tr>
<tr>
<td></td>
<td>Response Headers</td>
</tr>
<tr>
<td></td>
<td>Response Data</td>
</tr>
<tr>
<td></td>
<td>Buffered Streams</td>
</tr>
</tbody>
</table>
About This Guide

This guide discusses how to use Netscape Server Application Programmer’s Interface (NSAPI) to build plugins that define Server Application Functions (SAFs) to extend and modify Sun™ Open Net Environment (Sun ONE) Web Server 6.1. The guide also provides a reference of the NSAPI functions you can use to define new plugins.

This preface contains the following topics:

• Who Should Use This Guide
• Using the Documentation
• How This Guide Is Organized
• Documentation Conventions
• Product Support

Who Should Use This Guide

The intended audience for this guide is the person who develops, assembles, and deploys NSAPI plugins in a corporate enterprise. This guide assumes you are familiar with the following topics:

• HTTP
• HTML
• NSAPI
• C programming
• Software development processes, including debugging and source code control
Using the Documentation

The Sun ONE Web Server manuals are available as online files in PDF and HTML formats from the following location:

http://docs.sun.com/db/prod/s1websrv#hic

The following table lists the tasks and concepts described in the Sun ONE Web Server manuals.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Sun ONE Web Server Documentation Roadmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>For Information About</td>
<td>See the Following</td>
</tr>
<tr>
<td>Late-breaking information about the software and documentation</td>
<td>Release Notes</td>
</tr>
<tr>
<td>Getting started with Sun ONE Web Server, including hands-on exercises that introduce server basics and features (recommended for first-time users)</td>
<td>Getting Started Guide</td>
</tr>
<tr>
<td>Performing installation and migration tasks:</td>
<td></td>
</tr>
<tr>
<td>• Installing Sun ONE Web Server and its various components, supported platforms, and environments</td>
<td></td>
</tr>
<tr>
<td>• Migrating from Sun ONE Web Server 4.1 or 6.0 to Sun ONE Web Server 6.1</td>
<td></td>
</tr>
</tbody>
</table>


### Table 1  Sun ONE Web Server Documentation Roadmap

<table>
<thead>
<tr>
<th>For Information About</th>
<th>See the Following</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performing the following administration tasks:</td>
<td><strong>Administrator’s Guide</strong></td>
</tr>
<tr>
<td>• Using the Administration and command-line interfaces</td>
<td></td>
</tr>
<tr>
<td>• Configuring server preferences</td>
<td></td>
</tr>
<tr>
<td>• Using server instances</td>
<td></td>
</tr>
<tr>
<td>• Monitoring and logging server activity</td>
<td></td>
</tr>
<tr>
<td>• Using certificates and public key cryptography to secure the server</td>
<td></td>
</tr>
<tr>
<td>• Configuring access control to secure the server</td>
<td></td>
</tr>
<tr>
<td>• Using Java™ 2 Platform, Enterprise Edition (J2EE™ platform) security features</td>
<td></td>
</tr>
<tr>
<td>• Deploying applications</td>
<td></td>
</tr>
<tr>
<td>• Managing virtual servers</td>
<td></td>
</tr>
<tr>
<td>• Defining server workload and sizing the system to meet performance needs</td>
<td></td>
</tr>
<tr>
<td>• Searching the contents and attributes of server documents, and creating a text search interface</td>
<td></td>
</tr>
<tr>
<td>• Configuring the server for content compression</td>
<td></td>
</tr>
<tr>
<td>• Configuring the server for web publishing and content authoring using WebDAV</td>
<td></td>
</tr>
<tr>
<td>Using programming technologies and APIs to do the following:</td>
<td><strong>Programmer’s Guide</strong></td>
</tr>
<tr>
<td>• Extend and modify Sun ONE Web Server</td>
<td></td>
</tr>
<tr>
<td>• Dynamically generate content in response to client requests</td>
<td></td>
</tr>
<tr>
<td>• Modify the content of the server</td>
<td></td>
</tr>
</tbody>
</table>
How This Guide Is Organized

This guide has the following chapters:

- **Chapter 1, “Syntax and Use of obj.conf”**
  This chapter describes the configuration file \texttt{obj.conf}. The chapter discusses the syntax and use of directives in this file, which instruct the server how to process HTTP requests.

- **Chapter 2, “SAFs in the magnus.conf File”**
  This chapter discusses the SAFs you can set in the configuration file \texttt{magnus.conf} to configure the Sun ONE Web Server during initialization.

- **Chapter 3, “Creating Custom SAFs”**
  This chapter discusses how to create your own plugins that define new SAFs to modify or extend the way the server handles requests.

- **Chapter 4, “Creating Custom Filters”**
  This chapter discusses how to create your own custom filters that you can use to intercept, and potentially modify, incoming content presented to or generated by another function.

- **Chapter 5, “Examples of Custom SAFs and Filters”**
  This chapter describes examples of custom SAFs to use at each stage in the request-handling process.
• Chapter 6, “Creating Custom Server-parsed HTML Tags”
  This chapter explains how to create custom server-parsed HTML tags.

• Chapter 7, “NSAPI Function Reference”
  This chapter presents a reference of the NSAPI functions. You use NSAPI functions to define SAFs.

• Chapter 8, “Data Structure Reference”
  This chapter discusses some of the commonly used NSAPI data structures.

• Chapter 9, “Using Wildcard Patterns”
  This chapter lists the wildcard patterns you can use when specifying values in obj.conf and various predefined SAFs.

• Chapter 10, “Time Formats”
  This chapter lists time formats.

• Chapter 11, “Dynamic Results Caching Functions”
  This chapter explains how to create a results caching plugin.

• Chapter 12, “Hypertext Transfer Protocol”
  This chapter gives an overview of HTTP.

• Appendix A, “Alphabetical List of NSAPI Functions and Macros”
  This appendix provides an alphabetical list of NSAPI functions and macros.
Documentation Conventions

This section describes the types of conventions used throughout this guide.

- **File and directory paths**
  These are given in UNIX® format (with forward slashes separating directory names). For Windows versions, the directory paths are the same, except that backslashes are used to separate directories.

- **URLs** are given in the format:
  
  http://server.domain/path/file.html

  In these URLs, *server* is the server name where applications are run; *domain* is your Internet domain name; *path* is the server's directory structure; and *file* is an individual file name. Italic items in URLs are placeholders.

- **Font conventions** include:
  - The *monospace* font is used for sample code and code listings, API and language elements (such as function names and class names), file names, path names, directory names, and HTML tags.
  - *Italic* monospace type is used for code variables.
  - *Italic* type is also used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
  - *Bold* type is used as either a paragraph lead-in or to indicate words used in the literal sense.

- **Installation root directories** are indicated by `install_dir` in this guide.

  By default, the location of `install_dir` is as follows:
  - On UNIX-based platforms: `/opt/SUNWwbsvr/`
  - On Windows: `C:\Sun\WebServer6.1`
Product Support

If you have problems with your system, contact customer support using one of the following mechanisms:

- The online support web site at:
  
  http://www.sun.com/supporttraining/

- The telephone dispatch number associated with your maintenance contract

Please have the following information available prior to contacting support. This helps to ensure that our support staff can best assist you in resolving problems:

- Description of the problem, including the situation where the problem occurs and its impact on your operation.

- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem.

- Detailed steps on the methods you have used to reproduce the problem.

- Any error logs or core dumps.
Product Support
Chapter 1

Syntax and Use of obj.conf

The obj.conf configuration file contains directives that instruct the Sun™ Open Net Environment (Sun ONE) Web Server how to handle HTTP and HTTPS requests from clients and service web server content such as native server plugins and CGI programs. You can modify and extend the request-handling process by adding or changing the instructions in obj.conf.

All obj.conf files are located in the instance_dir/config directory, where instance_dir is the path to the installation directory of the server instance. There is one obj.conf file for each virtual server class, unless several virtual server classes are configured to share an obj.conf file. Whenever this guide refers to "the obj.conf file," it refers to all obj.conf files or to the obj.conf file for the virtual server class being described.

By default, the obj.conf file for the initial virtual server class is named obj.conf, and the obj.conf files for the administrator-defined virtual server classes are named virtual_server_class_id.obj.conf. Editing one of these files directly or through the Administration interface changes the configuration of a virtual server class.

This chapter discusses server instructions in obj.conf, the use of OBJECT tags, the use of variables, the flow of control in obj.conf, the syntax rules for editing obj.conf, and a note about example directives.

NOTE For detailed information about the standard directives and predefined Server Application Functions (SAFs) that are used in the obj.conf file, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.
How the Server Handles Requests from Clients

This chapter has the following sections:

- How the Server Handles Requests from Clients
- Dynamic Reconfiguration
- Server Instructions in obj.conf
- Configuring HTTP Compression
- The Object and Client Tags
- Variables Defined in server.xml
- Flow of Control in obj.conf
- Changes in Function Flow
- Syntax Rules for Editing obj.conf
- About obj.conf Directive Examples

How the Server Handles Requests from Clients

Sun ONE Web Server is a web server that accepts and responds to Hypertext Transfer Protocol (HTTP) requests. Browsers such as Netscape™ Communicator communicate using several protocols including HTTP and FTP. The Sun ONE Web Server handles HTTP specifically.

For more information about the HTTP protocol, refer to Chapter 12, “Hypertext Transfer Protocol” and also the latest HTTP specification.

HTTP Basics

As a quick summary, the HTTP/1.1 protocol works as follows:

- The client (usually a browser) opens a connection to the server and sends a request.
- The server processes the request, generates a response, and closes the connection if it finds a Connection: Close header.

The request consists of a line indicating a method such as GET or POST, a Universal Resource Identifier (URI) indicating which resource is being requested, and an HTTP protocol version separated by spaces.
This is normally followed by a number of headers, a blank line indicating the end of the headers, and sometimes body data. Headers may provide various information about the request or the client body data. Headers are typically only sent for POST and PUT methods.

The example request shown below would be sent by a Netscape browser to request the server foo.com to send back the resource in /index.html. In this example, no body data is sent because the method is GET (the point of the request is to get some data, not to send it).

```
GET /index.html HTTP/1.0
User-agent: Mozilla
Accept: text/html, text/plain, image/jpeg, image/gif, */*
Host: foo.com
```

The server receives the request and processes it. It handles each request individually, although it may process many requests simultaneously. Each request is broken down into a series of steps that together make up the request-handling process.

The server generates a response that includes the HTTP protocol version, HTTP status code, and a reason phrase separated by spaces. This is normally followed by a number of headers. The end of the headers is indicated by a blank line. The body data of the response follows. A typical HTTP response might look like this:

```
HTTP/1.0 200 OK
Server: Sun-ONE-Web-Server/6.1
Content-type: text/html
Content-length: 83

<HTML>
<HEAD><TITLE>Hello World</TITLE></HEAD>
<BODY>Hello World</BODY>
</HTML>
```
The status code and reason phrase tell the client how the server handled the request. Normally the status code 200 is returned, indicating that the request was handled successfully and the body data contains the requested item. Other result codes indicate redirection to another server or the browser’s cache, or various types of HTTP errors such as 404 Not Found.

**NSAPI Filters**

In previous versions of the Web Server, the NSAPI API allowed multiple Server Application Functions (SAFs) to interact in request processing. For example, one SAF could be used to authenticate the client after which a second SAF would generate the content.

In addition to the existing NSAPI interfaces, Sun ONE Web Server introduces NSAPI filters that enable a function to intercept (and potentially modify) the content presented to or generated by another function.

For more information on NSAPI filters in Sun ONE Web Server 6.1, see Chapter 4, “Creating Custom Filters.”

Two new NSAPI stages, Input and Output, can be used to insert filters in obj.conf. The Input and Output stages are described later in this chapter.

**Steps in the Request-handling Process**

When the server first starts up it performs some initialization and then waits for an HTTP request from a client (such as a browser). When it receives a request, it first selects a virtual server. For details about how the virtual server is determined, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference Guide.

After the virtual server is selected, the obj.conf file for the virtual server class specifies how the request is handled in the following steps:

1. **AuthTrans** (authorization translation)
   - Verify any authorization information (such as name and password) sent in the request.

2. **NameTrans** (name translation)
   - Translate the logical URI into a local file system path.
3. **PathCheck** (path checking)
   
   Check the local file system path for validity and check that the requestor has access privileges to the requested resource on the file system.

4. **ObjectType** (object typing)
   
   Determine the MIME-type (Multi-purpose Internet Mail Encoding) of the requested resource (for example, text/html, image/gif, and so on).

5. **Input** (prepare to read input)
   
   Select filters that will process incoming request data read by the Service step.

6. **Output** (prepare to send output)
   
   Select filters that will process outgoing response data generated by the Service step.

7. **Service** (generate the response)
   
   Generate and return the response to the client.

8. **AddLog** (adding log entries)
   
   Add entries to log file(s).

9. **Error** (service)
   
   This step is executed only if an error occurs in the previous steps. If an error occurs, the server logs an error message and aborts the process.

---

### Directives for Handling Requests

The file `obj.conf` contains a series of instructions, known as directives, that tell the Sun ONE Web Server what to do at each stage in the request-handling process. Each directive invokes a Server Application Function (SAF) with one or more arguments. Each directive applies to a specific stage in the request-handling process. The stages are **AuthTrans**, **NameTrans**, **PathCheck**, **ObjectType**, **Input**, **Output**, **Service**, and **AddLog**.

For example, the following directive applies during the **NameTrans** stage. It calls the `document-root` function with the `root` argument set to `D://Sun/WebServer61/Server1/docs`. (The `document-root` function translates the `http://server_name/` part of the URL to the document root, which in this example is `D://Sun/WebServer61/Server1/docs`.)

```
NameTrans fn="document-root" root="D:/Sun/WebServer61/Server1/docs"
```
Dynamic Reconfiguration

You do not need to restart the server for changes to certain configuration files to take effect (for example, `obj.conf`, `mime.types`, `server.xml`, and virtual server-specific ACL files). All you need to do is apply the changes by clicking the Apply link and then clicking the Load Configuration Files button on the Apply Changes screen. If there are errors in installing the new configuration, the previous configuration is restored.

When you edit `obj.conf` and apply the changes, a new configuration is loaded into memory that contains all of the information from the dynamically configurable files.

Every new connection references the newest configuration. Once the last session referencing a configuration ends, the now unused old configuration is deleted.

Server Instructions in `obj.conf`

The `obj.conf` file contains directives that instruct the server how to handle requests received from clients such as browsers. These directives appear inside `OBJECT` tags.

Each directive calls a function, indicating when to call it and specifying arguments for it.

The syntax of each directive is:

```
Directive fn=func-name name1="value1"...nameN="valueN"
```

For example:

```
NameTrans fn="document-root" root="D:/Sun/WebServer61/Server1/docs"
```

Directive indicates when this instruction is executed during the request-handling process. The value is one of `AuthTrans`, `NameTrans`, `PathCheck`, `ObjectType`, `Service`, `AddLog`, and `Error`.

The value of the `fn` argument is the name of the SAF to execute. All directives must supply a value for the `fn` parameter; if there’s no function, the instruction won’t do anything.
The remaining parameters are the arguments needed by the function, and they vary from function to function.

Sun ONE Web Server is shipped with a set of built-in Server Application Functions (SAFs) that you can use to create and modify directives in obj.conf. For more information about these predefined SAFs, see the Sun ONE Web Server 6.1 Administrator's Configuration File Reference. You can also define new SAFs, as discussed in Chapter 3, “Creating Custom SAFs.”

The magnus.conf file contains Init directive SAFs that initialize the server. For more information, see Chapter 2, “SAFs in the magnus.conf File.”

Summary of the Directives

Following are the categories of server directives and a description of what each does. Each category corresponds to a stage in the request-handling process. The section “Flow of Control in obj.conf” on page 37 explains exactly how the server decides which directive or directives to execute in each stage.

**NOTE** For detailed information about the standard directives and predefined Server Application Functions (SAFs) that are used in the obj.conf file, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

- **AuthTrans**
  
  Verifies any authorization information (normally sent in the Authorization header) provided in the HTTP request and translates it into a user and/or a group. Server access control occurs in two stages. AuthTrans verifies the authenticity of the user. Later, PathCheck tests the user’s access privileges for the requested resource.

  AuthTrans fn=basic-auth userfn=ntauth auth-type=basic userdb=none

  This example calls the basic-auth function, which calls a custom function (in this case ntauth, to verify authorization information sent by the client. The Authorization header is sent as part of the basic server authorization scheme.

- **NameTrans**
  
  Translates the URL specified in the request from a logical URL to a physical file system path for the requested resource. This may also result in redirection to another site. For example:
NameTrans fn="document-root"
root="D:/Sun/WebServer61/Server1/docs"

This example calls the document-root function with a root argument of D:/Sun/WebServer61/Server1/docs. The function document-root function translates the http://server_name/ part of the requested URL to the document root, which in this case is D:/Sun/WebServer61/Server1/docs. Thus a request for http://server-name/doc1.html is translated to D:/Sun/WebServer61/Server1/docs/doc1.html.

- PathCheck

Performs tests on the physical path determined by the NameTrans step. In general, these tests determine whether the path is valid and whether the client is allowed to access the requested resource. For example:

PathCheck fn="find-index" index-names="index.html,home.html"

This example calls the find-index function with an index-names argument of index.html,home.html. If the requested URL is a directory, this function instructs the server to look for a file called either index.html or home.html in the requested directory.

- ObjectType

Determines the MIME (Multi-purpose Internet Mail Encoding) type of the requested resource. The MIME type has attributes type (which indicates content type), encoding, and language. The MIME type is sent in the headers of the response to the client. The MIME type also helps determine which Service directive the server should execute.

The resulting type may be:

- A common document type such as text/html or image/gif (for example, the file name extension .gif translates to the MIME type image/gif).
- An internal server type. Internal types always begin with magnus-internal.

For example:

ObjectType fn="type-by-extension"

This example calls the type-by-extension function, which causes the server to determine the MIME type according to the requested resource’s file extension.

- Input
Selects filters that will process incoming request data read by the Service step. The Input directive allows you to invoke the insert-filter SAF in order to install filters that process incoming data. All Input directives are executed when the server or a plugin first attempts to read entity body data from the client. The Input directives are executed at most once per request. For example:

```
Input fn="insert-filter" filter="http-decompression"
```

This directive instructs the insert-filter function to add a filter named http-decompression to the filter stack, which would decompress incoming HTTP request data before passing it to the Service step.

- **Output**

Selects filters that will process outgoing response data generated by the Service step. The Output directive allows you to invoke the insert-filter SAF to install filters that process outgoing data. All Output directives are executed when the server or a plugin first attempts to write entity body data from the client. The Output directives are executed at most once per request. For example:

```
Output fn="insert-filter" filter="http-compression"
```

This directive instructs the insert-filter function to add a filter named http-compression to the filter stack, which would compress outgoing HTTP response data generated by the Service step.

- **Service**

Generates and sends the response to the client. This involves setting the HTTP result status, setting up response headers (such as Content-Type and Content-Length), and generating and sending the response data. The default response is to invoke the send-file function to send the contents of the requested file along with the appropriate header files to the client.

The default Service directive is:

```
Service method="(GET|HEAD|POST)" type="*~magnus-internal/*" fn="send-file"
```

This directive instructs the server to call the send-file function in response to any request whose method is GET, HEAD, or POST, and whose type does not begin with magnus-internal/. (Note the use of the special characters ~ to mean "does not match.")

Another example is:
Service method=" (GET|HEAD) " type="magnus-internal/imagemap" fn="imagemap"

In this case, if the method of the request is either GET or HEAD, and the type of the requested resource is "magnus-internal/imagemap," the function imagemap is called.

- **AddLog**

  Adds an entry to a log file to record information about the transaction. For example:

  AddLog fn="flex-log" name="access"

  This example calls the flex-log function to log information about the current request in the log file named access.

- **Error**

  Handles an HTTP error. This directive is invoked if a previous directive results in an error. Typically the server handles an error by sending a custom HTML document to the user describing the problem and possible solutions.

  For example:

  Error fn="send-error" reason="Unauthorized" path="D:/Sun/WebServer61/Server1/errors/unauthorized.html"

  In this example, the server sends the file in D:/Sun/WebServer61/Server1/errors/unauthorized.html whenever a client requests a resource that it is not authorized to access.

### Configuring HTTP Compression

When compression is enabled in the server, an entry gets added to the obj.conf file. A sample entry is shown below:

Output fn="insert-filter" filter="http-compression" type="text/\*"

Depending on the options specified, this line might also contain these options:

vary="on" compression-level="9"

To restrict compression to documents of only a particular type, or to exclude browsers that don’t work well with compressed content, you would need to edit the obj.conf file, as discussed below.
The option that appears as:

type="text/*"

restricts compression to documents that have a MIME type of text/* (for example, text/ascii, text/css, text/html, and so on). This can be modified to compress only certain types of documents. If you want to compress only HTML documents, for example, you would change the option to:

type="text/html"

Alternatively, you can specifically exclude browsers that are known to misbehave when they receive compressed content (but still request it anyway) by using the <Client> tag as follows:

```
<Client match="none"
   browser="*MSIE [1-3]*"
   browser="*MSIE [1-5]*Mac*"
   browser="Mozilla/[1-4]*Nav*">
   Output fn="insert-filter" filter="http-compression" type="text/*"
</Client>
```

This restricts compression to browsers that are not any of the following:

- Internet Explorer for Windows earlier than version 4
- Internet Explorer for Macintosh earlier than version 6
- Netscape Navigator/Communicator earlier than version 6

Internet Explorer on Windows earlier than version 4 may request compressed data at times, but does not correctly support it. Internet Explorer on Macintosh earlier than version 6 does the same. Netscape Communicator version 4.x requests compression, but only correctly handles compressed HTML. It will not correctly handle linked CSS or JavaScript from the compressed HTML, so administrators often simply prevent their servers from sending any compressed content to that browser (or earlier).

For more information about the <Client> tag, see the “The Client Tag” on page 34.
The Object and Client Tags

This section discusses the use of <Object> and <Client> tags in the file obj.conf.

<Object> tags group directives that apply to requests for particular resources, while <Client> tags group directives that apply to requests received from specific clients.

These tags are described in the following topics:

- The Object Tag
- The Client Tag

The Object Tag

Directives in the obj.conf file are grouped into objects that begin with an <Object> tag and end with an </Object> tag. The default object provides instructions to the server about how to process requests by default. Each new object modifies the default object’s behavior.

An Object tag may have a name attribute or a ppath attribute. Either parameter may be a wildcard pattern. For example:

<Object name="cgi">
- or -

<Object ppath="/usr/sun/webserver61/server1/docs/private/**">

The server always starts handling a request by processing the directives in the default object. However, the server switches to processing directives in another object after the NameTrans stage of the default object if either of the following conditions is true:

- The successful NameTrans directive specifies a name argument.
- The physical path name that results from the NameTrans stage matches the ppath attribute of another object.

When the server has been alerted to use an object other than the default object, it processes the directives in the other object before processing the directives in the default object. For some steps in the process, the server stops processing directives in that particular stage (such as the Service stage) as soon as one is successfully executed, whereas for other stages the server processes all directives in that stage, including the ones in the default object as well as those in the additional object. For more details, see “Flow of Control in obj.conf” on page 37.
Objects that Use the name Attribute

If a NameTrans directive in the default object specifies a name argument, the server switches to processing the directives in the object of that name before processing the remaining directives in the default object.

For example, the following NameTrans directive in the default object assigns the name cgi to any request whose URL starts with http://server_name/cgi/:

```xml
<Object name="default">
  NameTrans fn="pfx2dir" from="/cgi"
  dir="D:/sun/webserver61/server1/docs/m ycgi" name="cgi"
  ...
</Object>
```

When that NameTrans directive is executed, the server starts processing directives in the object named cgi:

```xml
<Object name="cgi">
  more directives...
</Object>
```

Objects that Use the ppath Attribute

When the server finishes processing the NameTrans directives in the default object, the logical URL of the request will have been converted to a physical path name. If this physical path name matches the ppath attribute of another object in obj.conf, the server switches to processing the directives in that object before processing the remaining ones in the default object.

For example, the following NameTrans directive translates the http://server_name/ part of the requested URL to D:/Sun/WebServer61/Server1/docs/ (which is the document root directory):

```xml
<Object name="default">
  NameTrans fn="document-root"
  root="D:/Sun/WebServer61/Server1/docs"
  ...
</Object>
```
The URL http://server_name/internalplan1.html would be translated to D:/Sun/WebServer61/Server1/docs/internalplan1.html. However, suppose that obj.conf contains the following additional object:

```html
<Object ppath="*internal*">
  more directives...
</Object>
```

In this case, the partial path *internal* matches the path D:/Sun/WebServer61/Server1/docs/internalplan1.html. So now the server starts processing the directives in this object before processing the remaining directives in the default object.

## The Client Tag

The `<Client>` tag is used to limit execution of a set of directives to requests received from specific clients. Directives listed between the `<Client>` and `</Client>` tags are executed only when information in the client request matches the parameter values specified.

### Client Tag Parameters

The following table lists the `<Client>` tag parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>browser</td>
<td>User-agent string sent by a browser to the Web Server</td>
</tr>
<tr>
<td>chunked</td>
<td>Boolean value set by a client requesting chunked encoding</td>
</tr>
<tr>
<td>code</td>
<td>HTTP response code</td>
</tr>
<tr>
<td>dns</td>
<td>DNS name of the client</td>
</tr>
<tr>
<td>internal</td>
<td>Boolean value indicating internally generated request</td>
</tr>
<tr>
<td>ip</td>
<td>IP address of the client</td>
</tr>
<tr>
<td>keep-alive</td>
<td>Boolean value indicating the client has requested a keep-alive connection</td>
</tr>
<tr>
<td>keysize</td>
<td>Key size used in an SSL transaction</td>
</tr>
</tbody>
</table>
The <Client> tag parameters provide greater control over when and if directives are executed. In the following example, use of the odds parameter gives a request a 25% chance of being redirected:

```xml
<Client odds="25%">
  NameTrans fn="redirect" from="/Pogues"
  url-prefix="http://pogues.example.com"
</Client>
```

One or more wildcard patterns can be used to specify Client tag parameter values.
Wildcards can also be used to exclude clients that match the parameter value specified in the `<Client>` tag. In the following example, the `<Client>` tag and the `AddLog` directive are combined to direct the Web Server to log access requests from all clients except those from the specified subnet:

```
<Client ip="~192.85.250.*">
  AddLog fn="flex-log" name="access"
</Client>
```

Using the ~ wildcard negates the expression, so the Web Server excludes clients from the specified subnet.

You can also create a negative match by setting the match parameter of the `Client` tag to none. In the following example, access requests from the specified subnet are excluded, as are all requests to the virtual server `www.sunone.com`:

```
<Client match="none" ip="192.85.250.*" urlhost="www.sunone.com">
  AddLog fn="flex-log" name="access"
</Client>
```

For more information about wildcard patterns, see Chapter 9, “Using Wildcard Patterns.”
Variables Defined in server.xml

You can define variables in the server.xml file and reference them in an obj.conf file. For example, the following server.xml code defines and uses a variable called docroot:

```xml
<!DOCTYPE SERVER SYSTEM "server.dtd" [
  <!ATTLIST VARS
docroot CDATA #IMPLIED>
>]
...
  <VS id="a.com" connections="ls1" urlhosts="a.com"
      mime="mime1" aclids="std">
    <property name="docroot" value="/opt/SUNWwbsvr/docs"/>
  </VS>
...
```

You can reference the variable in obj.conf as follows:

```
NameTrans fn=document-root root="$docroot"
```

Using this docroot variable saves you from having to define document roots for virtual server classes in the obj.conf files. It also allows you to define different document roots for different virtual servers within the same virtual server class.

**NOTE**  
Variable substitution is allowed only in an obj.conf file. It is not allowed in any other Sun ONE Web Server configuration files. Any variable referenced in an obj.conf file must be defined in the server.xml file.

For more information about defining variables, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

Flow of Control in obj.conf

Before the server can process a request, it must direct the request to the correct virtual server. For details about how the virtual server is determined, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

After the virtual server is determined, the server executes the obj.conf file for the virtual server class to which the virtual server belongs. This section discusses how the server decides which directives to execute in obj.conf.
Flow of Control in obj.conf

AuthTrans

When the server receives a request, it executes the AuthTrans directives in the default object to check that the client is authorized to access the server.

If there is more than one AuthTrans directive, the server executes them all (unless one of them results in an error). If an error occurs, the server skips all other directives except for Error directives.

NameTrans

Next, the server executes a NameTrans directive in the default object to map the logical URL of the requested resource to a physical path name on the server's file system. The server looks at each NameTrans directive in the default object in turn, until it finds one that can be applied.

If there is more than one NameTrans directive in the default object, the server considers each directive until one succeeds.

The NameTrans section in the default object must contain exactly one directive that invokes the document-root function. This function translates the http://server_name/part of the requested URL to a physical directory that has been designated as the server's document root. For example:

NameTrans fn="document-root" root="D:/Sun/WebServer61/server1/docs"

The directive that invokes document-root must be the last directive in the NameTrans section so that it is executed if no other NameTrans directive is applicable.

The pfx2dir (prefix to directory) function is used to set up additional mappings between URLs and directories. For example, the following directive translates the URL http://server_name/cgi/ into the directory path name D:/Sun/WebServer61/server1/docs/mycgi/:

NameTrans fn="pfx2dir" from="/cgi" dir="D:/Sun/WebServer61/server1/docs/mycgi"

Notice that if this directive appeared after the one that calls document-root, it would never be executed, with the result that the resultant directory path name would be D:/Sun/WebServer61/server1/docs/cgi/ (not mycgi). This illustrates why the directive that invokes document-root must be the last one in the NameTrans section.
How and When the Server Processes Other Objects

As a result of executing a NameTrans directive, the server might start processing directives in another object. This happens if the NameTrans directive that was successfully executed specifies a name or generates a partial path that matches the name or ppath attribute of another object.

If the successful NameTrans directive assigns a name by specifying a name argument, the server starts processing directives in the named object (defined with the OBJECT tag) before processing directives in the default object for the rest of the request-handling process.

For example, the following NameTrans directive in the default object assigns the name cgi to any request whose URL starts with http://server_name/cgi/.

```xml
<Object name="default">
    ...
    NameTrans fn="pfx2dir" from="/cgi"
    dir="D:/Sun/WebServer61/server1/docs/mycgi" name="cgi"
    ...
</Object>
```

When that NameTrans directive is executed, the server starts processing directives in the object named cgi:

```xml
<Object name="cgi">
    more directives...
</Object>
```

When a NameTrans directive has been successfully executed, there will be a physical path name associated with the requested resource. If the resultant path name matches the ppath (partial path) attribute of another object, the server starts processing directives in the other object before processing directives in the default object for the rest of the request-handling process.
For example, suppose obj.conf contains an object as follows:

```xml
<Object ppath="*internal*">
  more directives...
</Object>
```

Now suppose the successful NameTrans directive translates the requested URL to the path name D:/Sun/WebServer61/server1/docs/internalplan1.html. In this case, the partial path *internal* matches the path D:/Sun/WebServer61/server1/docs/internalplan1.html. So now the server would start processing the directives in this object before processing the remaining directives in the default object.

### PathCheck

After converting the logical URL of the requested resource to a physical path name in the NameTrans step, the server executes PathCheck directives to verify that the client is allowed to access the requested resource.

If there is more than one PathCheck directive, the server executes all of the directives in the order in which they appear, unless one of the directives denies access. If access is denied, the server switches to executing directives in the Error section.

If the NameTrans directive assigned a name or generated a physical path name that matches the name or ppath attribute of another object, the server first applies the PathCheck directives in the matching object before applying the directives in the default object.

### ObjectType

Assuming that the PathCheck directives all approve access, the server next executes the ObjectType directives to determine the MIME type of the request. The MIME type has three attributes: type, encoding, and language. When the server sends the response to the client, the type, language, and encoding values are transmitted in the headers of the response. The type also frequently helps the server to determine which Service directive to execute to generate the response to the client.
If there is more than one ObjectType directive, the server applies all of the directives in the order in which they appear. However, once a directive sets an attribute of the MIME type, further attempts to set the same attribute are ignored. The reason that all ObjectType directives are applied is that one directive may set one attribute, for example type, while another directive sets a different attribute, such as language.

As with the PathCheck directives, if another object has been matched to the request as a result of the NameTrans step, the server executes the ObjectType directives in the matching object before executing the ObjectType directives in the default object.

### Setting the Type By File Extension

Usually the default way the server figures out the MIME type is by calling the type-by-extension function. This function instructs the server to look up the MIME type according to the requested resource’s file extension in the MIME types table. This table was created during virtual server initialization by the MIME types file (which is usually called mime.types).

For example, the entry in the MIME types table for the extensions .html and .htm is usually:

```
type=text/html  exts=htm,html
```

which says that all files with the extension .htm or .html are text files formatted as HTML, and the type is text/html.

Note that if you make changes to the MIME types file, you must reconfigure the server before those changes can take effect.

For more information about MIME types, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference Guide.

### Forcing the Type

If no previous ObjectType directive has set the type, and the server does not find a matching file extension in the MIME types table, the type still has no value even after type-by-expression has been executed. Usually if the server does not recognize the file extension, it is a good idea to force the type to be text/plain, so that the content of the resource is treated as plain text. There are also other situations where you might want to set the type regardless of the file extension, such as forcing all resources in the designated CGI directory to have the MIME type magnus-internal/cgi.

The function that forces the type is force-type.
For example, the following directives first instruct the server to look in the MIME types table for the MIME type, then if the type attribute has not been set (that is, the file extension was not found in the MIME types table), set the type attribute to text/plain.

```plaintext
ObjectType fn="type-by-extension"
ObjectType fn="force-type" type="text/plain"
```

If the server receives a request for a file abc.dogs, it looks in the MIME types table, does not find a mapping for the extension .dogs, and consequently does not set the type attribute. Since the type attribute has not already been set, the second directive is successful, forcing the type attribute to text/plain.

The following example illustrates another use of force-type. In this example, the type is forced to magnus-internal/cgi before the server gets a chance to look in the MIME types table. In this case, all requests for resources in http://server_name/cgi/ are translated into requests for resources in the directory D:/Sun/WebServer61/server1/docs/mycgi/. Since a name is assigned to the request, the server processes ObjectType directives in the object named cgi before processing the ones in the default object. This object has one ObjectType directive, which forces the type to be magnus-internal/cgi.

```plaintext
NameTrans fn="pfx2dir" from="/cgi"
dir="D:/Sun/WebServer61/server1/docs/mycgi" name="cgi"
<Object name="cgi">
    ObjectType fn="force-type" type="magnus-internal/cgi"
    Service fn="send-cgi"
</Object>
```

The server continues processing all ObjectType directives including those in the default object, but since the type attribute has already been set, no other directive can set it to another value.
Input

The Input directive selects filters that will process incoming request data read by the Service step. It allows you to invoke the insert-filter SAF in order to install filters that process incoming data.

The Input directives are executed at most once per request.

You can define the appropriate position of a specific filter within the filter stack. For example, filters that translate content from XML to HTML are placed higher in the filter stack than filters that compress data for transmission. You can use the filter_create function to define the filter’s position in the filter stack, and init-filter-order to override the defined position.

When two or more filters are defined to occupy the same position in the filter stack, filters that were inserted later will appear higher than filters that were inserted earlier. That is, the order of Input fn="insert-filter" and Output fn="insert-filter" directives in obj.conf becomes important.

For more information, see “Chapter 4, “Creating Custom Filters.”

Output

The Output directive selects filters that will process outgoing response data generated by the Service step. The Output directive allows you to invoke the insert-filter SAF to install filters that process outgoing data. All Output directives are executed when the server or a plugin first attempts to write entity body data from the client.

The Output directives are executed at most once per request.

You can define the appropriate position of a specific filter within the filter stack. For example, filters that translate content from XML to HTML are placed higher in the filter stack than filters that compress data for transmission. You can use the filter_create function to define the filter’s position in the filter stack, init-filter-order to override the defined position.

When two or more filters are defined to occupy the same position in the filter stack, filters that were inserted later will appear higher than filters that were inserted earlier. That is, the order of Input fn="insert-filter" and Output fn="insert-filter" directives in obj.conf becomes important.

For more information, see Chapter 4, “Creating Custom Filters.”
Service

Next, the server needs to execute a Service directive to generate the response to send to the client. The server looks at each Service directive in turn, to find the first one that matches the type, method and query string. If a Service directive does not specify type, method, or query string, then the unspecified attribute matches anything.

If there is more than one Service directive, the server applies the first one that matches the conditions of the request, and ignores all remaining Service directives.

As with the PathCheck and ObjectType directives, if another object has been matched to the request as a result of the NameTrans step, the server considers the Service directives in the matching object before considering the ones in the default object. If the server successfully executes a Service directive in the matching object, it will not get around to executing the Service directives in the default object, since it only executes one Service directive.

Service Examples

For an example of how Service directives work, consider what happens when the server receives a request for the URL D:/server_name/jos.html. In this case, all directives executed by the server are in the default object.

- The following NameTrans directive translates the requested URL to D:/Sun/WebServer61/server1/docs/jos.html:
  
  NameTrans fn="document-root"
  root="D:/Sun/WebServer61/server1/docs"

- Assume that the PathCheck directives all succeed.

- The following ObjectType directive tells the server to look up the resource’s MIME type in the MIME types table:
  
  ObjectType fn="type-by-extension"

- The server finds the following entry in the MIME types table, which sets the type attribute to text/html:
  
  type=text/html exts=htm, html

- The server invokes the following Service directive. The value of the type parameter matches anything that does not begin with magnus-internal/. (For a list of all wildcard patterns, see Chapter 9, “Using Wildcard Patterns.”) This directive sends the requested file, jos.html, to the client.
Here is an example that involves using another object:

- The following NameTrans directive assigns the name personnel to the request.

```xml
NameTrans fn=assign-name name=personnel from=/personnel
```

- As a result of the name assignment, the server switches to processing the directives in the object named personnel. This object is defined as:

```xml
<Object name="personnel">
  Service fn="index-simple"
</Object>
```

- The personnel object has no PathCheck or ObjectType directives, so the server processes the PathCheck and ObjectType directives in the default object. Let's assume that all PathCheck and ObjectType directives succeed.
- When processing Service directives, the server starts by considering the Service directive in the personnel object, which is:

```xml
Service fn="index-simple"
```
- The server executes this Service directive, which calls the index-simple function.

Since a Service directive has now been executed, the server does not process any other Service directives. (However, if the matching object had not had a Service directive that was executed, the server would continue looking at Service directives in the default object.)
Default Service Directive

There is usually a Service directive that does the default task (sends a file) if no other Service directive matches a request sent by a browser. This default directive should come last in the list of Service directives in the default object, to ensure it only gets called if no other Service directives have succeeded. The default Service directive is usually:

```
Service method="(GET|HEAD|POST)" type="*~magnus-internal/*"
  fn="send-file"
```

This directive matches requests whose method is GET, HEAD, or POST, which covers nearly virtually all requests sent by browsers. The value of the type argument uses special pattern-matching characters. For complete information about the special pattern-matching characters, see Chapter 9, “Using Wildcard Patterns.”

The characters “*~” mean “anything that doesn’t match the following characters,” so the expression *~magnus-internal/* means “anything that doesn’t match magnus-internal/.” An asterisk by itself matches anything, so the whole expression *~magnus-internal/* matches anything that does not begin with magnus-internal/.

So if the server has not already executed a Service directive when it reaches this directive, it executes the directive so long as the request method is GET, HEAD or POST, and the value of the type attribute does not begin with magnus-internal/. The invoked function is send-file, which simply sends the contents of the requested file to the client.

AddLog

After the server generate the response and sends it to the client, it executes AddLog directives to add entries to the log files.

All AddLog directives are executed. The server can add entries to multiple log files.

Depending on which log files are used and which format they use, the Init section in magnus.conf may need to have directives that initialize the logs. For example, if one of the AddLog directives calls flex-log, which uses the extended log format, the Init section must contain a directive that invokes flex-init to initialize the flexible logging system.
For more information about initializing logs, see the discussion of the functions `flex-init` and `init-clf` in Chapter 2, “SAFs in the magnus.conf File.”

For more information about `flex-log`, see information about predefined SAFs in the `obj.conf` file in the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

**Error**

If an error occurs during the request-handling process, such as if a PathCheck or AuthTrans directive denies access to the requested resource, or the requested resource does not exist, the server immediately stops executing all other directives and immediately starts executing the `Error` directives.

**Changes in Function Flow**

There are times when the function flow changes from the normal request-handling process. This happens during internal redirects, restarts, and URI translation functions.

**Internal Redirects**

An example of an internal redirect is a servlet include or forward. In this case, because there is no exposed NSAPI function to handle an internal redirect, when an internal redirect occurs, the request structure is copied into `rq->orig_rq`. For more information on the request data structure, see “Request” on page 255.

**Restarts**

A restart occurs when a `REQ_RESTART` is returned from a PathCheck or Service function. For example, when a CGI is redirected using a relative path.

On a restart, much of the request is cleared. Some elements of the HTTP request (`rq->reqpb`), the server’s “working” variables (`rq->vars`), and response headers (`rq->srvhdrs`) are cleared. The method, protocol, and clf-request variables from `rq->reqpb` are saved. The saved variables are put back into the data structure. The
new URI is inserted (and if there is a query string in the new URI, that too is inserted) into $rq->reqpb$. The parameter $rq->rq_attr.req_restarted$ is set to 1.

For more information on the request data structure, see “Request” on page 255, and for more information on the $rq$ parameter, see “rq (request)” on page 82.

**URI Translation**

At times it is necessary to find the physical path for a URI without actually running a request. The function `request_translate_uri` does this. A new request structure is created and run through the AuthTrans and NameTrans stages to get the physical path. Thereafter, the new request is freed.

**Syntax Rules for Editing obj.conf**

Several rules are important in the obj.conf file. Be very careful when editing this file. Simple mistakes can make the server fail to start or operate correctly.

**CAUTION** Do not remove any directives from any obj.conf file that are present in the obj.conf file that exists when you first install Sun ONE Web Server. The server may not function properly.

**Order of Directives**

The order of directives is important, since the server executes them in the order they appear in obj.conf. The outcome of some directives affect the execution of other directives.

For PathCheck directives, the order within the PathCheck section is not so important, since the server executes all PathCheck directives. However, the order within the ObjectType section is very important, because if an ObjectType directive sets an attribute value, no other ObjectType directive can change that value. For example, if the default ObjectType directives were listed in the following order (which is the wrong way around), every request would have its type value set to text/plain, and the server would never have a chance to set the type according to the extension of the requested resource.
Similarly, the order of directives in the Service section is very important. The server executes the first Service directive that matches the current request and does not execute any others.

**Parameters**

The number and names of parameters depends on the function. The order of parameters on the line is not important.

**Case Sensitivity**

Items in the obj.conf file are case-sensitive including function names, parameter names, many parameter values, and path names.

**Separators**

The C language allows function names to be composed only of letters, digits, and underscores. You may use the hyphen (-) character in the configuration file in place of underscore (_) for your C code function names. This is only true for function names.

**Quotes**

Quotes (”) are only required around value strings when there is a space in the string. Otherwise they are optional. Each open-quote must be matched by a close-quote.

**Spaces**

- Spaces are not allowed at the beginning of a line except when continuing the previous line.
• Spaces are not allowed before or after the equal (=) sign that separates the name and value.
• Spaces are not allowed at the end of a line or on a blank line.

Line Continuation
A long line may be continued on the next line by beginning the next line with a space or tab.

Path Names
Always use forward slashes (/) rather than backslashes (\) in path names under Windows. Backslash escapes the next character.

Comments
Comments begin with a pound (#) sign. If you manually add comments to obj.conf, then use the Server Manager interface to make changes to your server, the Server Manager will wipe out your comments when it updates obj.conf.

About obj.conf Directive Examples
Every line in the obj.conf file begins with one of the following keywords:

AuthTrans
NameTrans
PathCheck
ObjectType
Input
Output
Service
AddLog
Error
<Object
</Object>

If any line of any example begins with a different word in the manual, the line is wrapping in a way that it does not in the actual file. In some cases this is due to line length limitations imposed by the PDF and HTML formats of the manuals.
For example, the following directive is all on one line in the actual obj.conf file:

```
NameTrans fn="pfx2dir" from="/cgi"
dir="D:/Sun/WebServer61/server1/docs/mycgi" name="cgi"
```
When the Sun ONE Web Server starts up, it looks in a file called `magnus.conf` in the `server-id/config` directory to establish a set of global variable settings that affect the server's behavior and configuration. Sun ONE Web Server executes all of the directives defined in `magnus.conf`. The order of the directives is not important.

**NOTE** When you edit the `magnus.conf` file, you must restart the server for the changes to take effect.

This chapter lists the `Init` SAFs that can be specified in `magnus.conf` in Sun ONE Web Server 6.1. For information about the other, non-SAF directives in `magnus.conf`, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

The `Init` directives initialize the server (for example, they load and initialize additional modules and plugins, and initialize log files).

The `Init` directives are SAFs, like `obj.conf` directives, and have SAF syntax rather than the simpler variable value syntax of other `magnus.conf` directives.

They are located in `magnus.conf` because, like other `magnus.conf` directives, they are executed only once at server startup.

Each `Init` directive has an optional `LateInit` parameter. For the UNIX platform, if `LateInit` is set to `yes`, the function is executed by the child process after it is forked from the parent. If `LateInit` is set to `no` or is not provided, the function is executed by the parent process before the fork. When the server is started up by user root but runs as another user, any activities that must be performed as the user root (such as writing to a root-owned file) must be done before the fork. Functions that create threads, with the exception of `thread-pool-init`, should execute after the fork (that is, the relevant `Init` directive should have `LateInit=yes` set).
For all platforms, any function that requires access to a fully parsed configuration should have `LateInit=yes` set on its `Init` directive.

Upon failure, `Init`-class functions return `REQ_ABORTED`. The server logs the error according to the instructions in the `Error` directives in `obj.conf`, and terminates. Any other result code is considered a success.

**Syntax**

`Init` functions have the following syntax:

```
Init fn=function param1="value1" ...paramN="valueN"
```

Directives have the following syntax:

```
directive value
```

The following `Init`-class functions and their parameters are described in detail in this chapter:

- `cindex-init` changes the default characteristics for fancy indexing.
- `define-perf-bucket` creates a performance bucket.
- `dns-cache-init` configures DNS caching.
- `flex-init` initializes the flexible logging system.
- `flex-rotate-init` enables rotation for flexible logs.
- `init-cgi` changes the default settings for CGI programs.
- `init-clf` initializes the Common Log subsystem.
- `init-dav` initializes the WebDAV subsystem.
- `init-filter-order` controls the position of specific filters within filter stacks.
- `init-j2ee` initializes the Java subsystem.
- `init-uhome` loads user home directory information.
- `load-modules` loads shared libraries into the server.
- `nt-console-init` enables the Windows console, which is the command-line shell that displays standard output and error streams.
- `perf-init` enables system performance measurement via performance buckets.
- `pool-init` configures pooled memory allocation.
- **register-http-method** lets you extend the HTTP protocol by registering new HTTP methods.
- **stats-init** enables reporting of performance statistics in XML format.
- **thread-pool-init** configures an additional thread pool.

## cindex-init

Applicable in Init-class directives.

The function `cindex-init` sets the default settings for common indexing. Common indexing (also known as fancy indexing) is performed by the `Service` function `index-common`. Indexing occurs when the requested URL translates to a directory that does not contain an index file or home page, or no index file or home page has been specified.

In common (fancy) indexing, the directory list shows the name, last modified date, size, and description for each indexed file or directory.

### Parameters

The following table describes parameters for the `cindex-init` function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| opts      | (Optional) String of letters specifying the options to activate. Currently there is only one possible option:  

   - `s` tells the server to scan each HTML file in the directory being indexed for the contents of the HTML `<TITLE>` tag to display in the description field. The `<TITLE>` tag must be within the first 255 characters of the file. This option is off by default.  

   The search for `<TITLE>` is not case-sensitive. |
Table 2-1  cindex-init parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>widths</td>
<td>(Optional) Specifies the width for each column in the indexing display. The string is a comma-separated list of numbers that specify the column widths in characters for name, last-modified date, size, and description, respectively. The default values for the widths parameter are 22, 18, 8, 33. The final three values (corresponding to last-modified date, size, and description, respectively) can each be set to 0 to turn the display for that column off. The name column cannot be turned off. The minimum size of a column (if the value is nonzero) is specified by the length of its title. For example, the minimum size of the date column is 5 (the length of “Date” plus one space). If you set a nonzero value for a column that is less than the length of its title, the width defaults to the minimum required to display the title.</td>
</tr>
<tr>
<td>timezone</td>
<td>(Optional) Indicates whether the last-modified time is shown in local time or in Greenwich Mean Time. The values are GMT or local. The default is local.</td>
</tr>
<tr>
<td>format</td>
<td>(Optional) Parameter determines the format of the last modified date display. It uses the format specification for the UNIX function strftime(). The default is %d-%b-%Y %H:%M.</td>
</tr>
<tr>
<td>ignore</td>
<td>(Optional) Specifies a wildcard pattern for file names the server should ignore while indexing. File names starting with a period (.) are always ignored. The default is to only ignore file names starting with a period (.).</td>
</tr>
<tr>
<td>icon-uri</td>
<td>(Optional) Specifies the URI prefix the index-common function uses when generating URLs for file icons (.gif files). By default, it is /mc-icons/. If icon-uri is different from the default, the pfx2dir function in the NameTrans directive must be changed so that the server can find these icons.</td>
</tr>
</tbody>
</table>
define-perf-bucket

Applicable in Init-class directives.

The define-perf-bucket function creates a performance bucket, which you can use to measure the performance of SAFs in obj.conf (for more information about predefined SAFs that are used in obj.conf, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference).

For more information about performance buckets, see the Sun ONE Web Server 6.1 Performance Tuning, Sizing, and Scaling Guide.

Parameters
The following table describes parameters for the define-perf-bucket function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name for the bucket (for example, cgi-bucket).</td>
</tr>
<tr>
<td>description</td>
<td>Description of what the bucket measures (for example, CGI Stats).</td>
</tr>
</tbody>
</table>

Example

```
Init fn=cindex-init widths=50,1,1,0
Init fn=cindex-init ignore=*private*
Init fn=cindex-init widths=22,0,0,50
```

Example

```
Init fn="define-perf-bucket" name="cgi-bucket" description="CGI Stats"
```
dns-cache-init

Applicable in Init-class directives.

The dns-cache-init function specifies that DNS lookups should be cached when DNS lookups are enabled. If DNS lookups are cached, then when the server gets a client’s host name information, it stores that information in the DNS cache. If the server needs information about the client in the future, the information is available in the DNS cache.

You may specify the size of the DNS cache and the time it takes before a cache entry becomes invalid. The DNS cache can contain 32 to 32768 entries; the default value is 1024 entries. Values for the time it takes for a cache entry to expire (specified in seconds) can range from 1 second to 1 year; the default value is 1200 seconds (20 minutes).

Parameters
The following table describes parameters for the dns-cache-init function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache-size</td>
<td>(Optional) Specifies how many entries are contained in the cache. Acceptable values are 32 to 32768; the default value is 1024.</td>
</tr>
<tr>
<td>expire</td>
<td>(Optional) Specifies how long (in seconds) it takes for a cache entry to expire. Acceptable values are 1 to 31536000 (1 year); the default is 1200 seconds (20 minutes).</td>
</tr>
</tbody>
</table>

Example

Init fn="dns-cache-init" cache-size="2140" expire="600"
flex-init

Applicable in Init-class directives.

The flex-init function opens the named log file to be used for flexible logging and establishes a record format for it. The log format is recorded in the first line of the log file. You cannot change the log format while the log file is in use by the server.

The flex-log function (applicable in AddLog-class directives) writes entries into the log file during the AddLog stage of the request-handling process.

The log file stays open until the server is shut down or restarted (at which time all logs are closed and reopened).

| NOTE | If the server has AddLog-stage directives that call flex-log, the flexible log file must be initialized by flex-init during server initialization.  

For more information about flex-log, see information about predefined SAFs in the obj.conf file in the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

You may specify multiple log file names in the same flex-init function call. Then use multiple AddLog directives with the flex-log function to log transactions to each log file.

The flex-init function may be called more than once. Each new log file name and format will be added to the list of log files.

If you move, remove, or change the currently active log file without shutting down or restarting the server, client accesses might not be recorded. To save or backup the currently active log file, you need to rename the file and then restart the server. The server first looks for the log file by name, and if it doesn’t find it, creates a new one (the renamed original log file is left for you to use).

For information on rotating log files, see flex-rotate-init.

The flex-init function has three parameters: one that names the log file, one that specifies the format of each record in that file, and one that specifies the logging mode.

Parameters
The following table describes parameters for the flex-init function.
More on Log Format

The flex-init function recognizes anything contained between percent signs (%) as the name portion of a name-value pair stored in a parameter block in the server. (The one exception to this rule is the %SYSDATE% component, which delivers the current system date.) %SYSDATE% is formatted using the time format %d/%b/%Y:%H:%M:%S plus the offset from GMT.

(See Chapter 3, “Creating Custom SAFs” for more information about parameter blocks, and Chapter 7, “NSAPI Function Reference” for functions that manipulate pblocks.)

Table 2-4 flex-init parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| logFileName       | Name of the parameter is the name of the log file. The value of the parameter specifies either the full path to the log file or a file name relative to the server’s logs directory. For example: access="/usr/netscape/server4/https-servername/logs/access" mylogfile = "log1"
                      | You will use the log file name later, as a parameter to the flex-log function (applicable in AddLog-class directives). |
| buffer-size       | Specifies the size of the global log buffer. The default is 8192. See the third flex-init example below. |
| buffers-per-file  | Specifies the number of buffers for a given log file. The default value is determined by the server. Access log entries can be logged in strict chronological order by using a single buffer per log file. To accomplish this, add buffers-per-file="1" to the Init fn="flex-log-init" line in magnus.conf. This ensures that requests are logged in chronological order. Note that this approach will result in decreased performance when the server is under heavy load. |
| format .logFileName | Specifies the format of each log entry in the log file. For information about the format, see the “More on Log Format” section below. |
Any additional text is treated as literal text, so you can add to the line to make it more readable. Typical components of the formatting parameter are listed in the following table Table 2-5. Certain components might contain spaces, so they should be bounded by escaped quotes (\`).

If no format parameter is specified for a log file, the common log format is used:

```
"%Ses->client.ip% - %Req->vars.auth-user% [%SYSDATE%]
 "%Req->reqpb.clf-request%" %Req->srvhdrs.clf-status%
%Req->srvhdrs.content-length%"
```

You can now log cookies by logging the `Req->headers.cookie.name` component.

In the following table, the components that are enclosed in escaped double quotes (\`) are the ones that could potentially resolve to values that have white spaces.

**Table 2-5** Typical Components of flex-init Formatting

<table>
<thead>
<tr>
<th>Flex-log Option</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client host name (unless iponly is specified in flex-log or DNS name is not available) or IP address</td>
<td><code>%Ses-&gt;client.ip%</code></td>
</tr>
<tr>
<td>Client DNS name</td>
<td><code>%Ses-&gt;client.dns%</code></td>
</tr>
<tr>
<td>System date</td>
<td><code>%SYSDATE%</code></td>
</tr>
<tr>
<td>Full HTTP request line</td>
<td>&quot;%Req-&gt;reqpb clf-request%&quot;</td>
</tr>
<tr>
<td>Status</td>
<td><code>%Req-&gt;srvhdrs clf-status%</code></td>
</tr>
<tr>
<td>Response content length</td>
<td><code>%Req-&gt;srvhdrs content-length%</code></td>
</tr>
<tr>
<td>Response content type</td>
<td><code>%Req-&gt;srvhdrs content-type%</code></td>
</tr>
<tr>
<td>Referer header</td>
<td>&quot;%Req-&gt;headersreferer%&quot;</td>
</tr>
<tr>
<td>User-agent header</td>
<td>&quot;%Req-&gt;headers.user-agent%&quot;</td>
</tr>
<tr>
<td>HTTP method</td>
<td><code>%Req-&gt;reqpb.method%</code></td>
</tr>
<tr>
<td>HTTP URI</td>
<td><code>%Req-&gt;reqpb.uri%</code></td>
</tr>
<tr>
<td>HTTP query string</td>
<td><code>%Req-&gt;reqpb.query%</code></td>
</tr>
<tr>
<td>HTTP protocol version</td>
<td><code>%Req-&gt;reqpb.protocol%</code></td>
</tr>
<tr>
<td>Accept header</td>
<td><code>%Req-&gt;headers.accept%</code></td>
</tr>
<tr>
<td>Date header</td>
<td><code>%Req-&gt;headers.date%</code></td>
</tr>
</tbody>
</table>
Table 2-5  Typical Components of flex-init Formatting

<table>
<thead>
<tr>
<th>Flex-log Option</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>If-Modified-Since header</td>
<td>%Req-&gt;headers.if-modified-since%</td>
</tr>
<tr>
<td>Authorization header</td>
<td>%Req-&gt;headers.authorization%</td>
</tr>
<tr>
<td>Any header value</td>
<td>%Req-&gt;headers.headername%</td>
</tr>
<tr>
<td>Name of authorized user</td>
<td>%Req-&gt;vars.auth-user%</td>
</tr>
<tr>
<td>Value of a cookie</td>
<td>%Req-&gt;headers.cookie.name%</td>
</tr>
<tr>
<td>Value of any variable in Req-&gt;vars</td>
<td>%Req-&gt;vars.varname%</td>
</tr>
<tr>
<td>Virtual server ID</td>
<td>%vsid%</td>
</tr>
<tr>
<td>Duration</td>
<td>%duration%</td>
</tr>
</tbody>
</table>

Table 2-5: Typical Components of flex-init Formatting

- Records the time in microseconds the server spent handling the request. Statistics must be enabled for the server instance before %duration% can be used. For information about enabling statistics, see the Sun ONE Web Server 6.1 Administrator’s Guide.

Examples
The first example below initializes flexible logging into the file /usr/sun/webserver61/server1/https-servername/logs/access.

<table>
<thead>
<tr>
<th>Init fn=flex-init</th>
</tr>
</thead>
<tbody>
<tr>
<td>access=&quot;/usr/sun/webserver61/server1/https-servername/logs/access&quot;</td>
</tr>
<tr>
<td>format.access=&quot;%Ses-&gt;client.ip% - %Req-&gt;vars.auth-user% [%SYSDATE%] &quot;%Req-&gt;reqpb.clf-request%&quot; %Req-&gt;srvhdrs.clf-status% %Req-&gt;srvhdrs.content-length%&quot;</td>
</tr>
</tbody>
</table>

62  Sun ONE Web Server 6.1  •  NSAPI Programmer’s Guide  •  April 2004
This will record the following items:

- IP or host name, followed by the three characters “ - “
- User name, followed by the two characters “[“
- System date, followed by the two characters “] “
- Full HTTP request in quotes, followed by a single space
- HTTP result status in quotes, followed by a single space
- Content length

This is the default format, which corresponds to the Common Log Format (CLF).

It is advisable that the first six elements of any log always be in exactly this format, because a number of log analyzers expect that as output.

The second example initializes flexible logging into the file

```
Init fn=flex-init
extended="/usr/sun/webserver61/server1/https-servername/logs/extended" format.extended="%Ses->client.ip% - %Req->vars.auth-user%
[%SYSDATE%] \"%Req->reqpb.clf-request\" %Req->srvhdrs.clf-status%
%Req->srvhdrs.content-length% %Req->headers.referer%
\"%Req->headers.user-agent\" %Req->reqpb.method% %Req->reqpb.uri%
%Req->reqpb.query% %Req->reqpb.protocol\"
```

The third example shows how logging can be tuned to prevent request handling threads from making blocking calls when writing to log files, instead delegating these calls to the log flush thread.

Doubling the size of the `buffer-size` and `num-buffers` parameters from their defaults and lowering the value of the `LogFlushInterval magnus.conf` directive to 4 seconds (see Chapter 2, “SAFs in the magnus.conf File”) frees the request-handling threads to quickly write the log data.
flex-rotate-init

Applicable in Init-class directives.

The `flex-rotate-init` function configures log rotation for all log files on the server, including error logs and the common-log, flex-log, and record-useragent AddLog SAFs. Call this function in the Init section of `magnus.conf` before calling `flex-init`. The `flex-rotate-init` function allows you to specify a time interval for rotating log files. At the specified time interval, the server moves the log file to a file whose name indicates the time of moving. The log functions in the AddLog stage in `obj.conf` then start logging entries in a new log file. The server does not need to be shut down while the log files are being rotated.

**NOTE** The server keeps all rotated log files forever, so you will need to clean them up as necessary to free disk space.

By default, log rotation is disabled.

Parameters

The following table describes parameters for the `flex-rotate-init` function.
Table 2-6  flex-rotate-init parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotate-start</td>
<td>Indicates the time to start rotation. This value is a four-digit string indicating the time in 24-hour format. For example, 0900 indicates 9 a.m., while 1800 indicates 9 p.m.</td>
</tr>
<tr>
<td>rotate-interval</td>
<td>Indicates the number of minutes to elapse between each log rotation.</td>
</tr>
<tr>
<td>rotate-access</td>
<td>(Optional) Determines whether common-log, flex-log, and record-useragent logs are rotated (AddLog SAFs). Values are yes (the default), and no.</td>
</tr>
<tr>
<td>rotate-error</td>
<td>(Optional) Determines whether error logs are rotated. Values are yes (the default), and no.</td>
</tr>
<tr>
<td>rotate-callback</td>
<td>(Optional) Specifies the file name of a user-supplied program to execute following log file rotation. The program is passed the post-rotation name of the rotated log file as its parameter.</td>
</tr>
</tbody>
</table>

Example

This example enables log rotation, starting at midnight and occurring every hour.

```
Init fn=flex-rotate-init rotate-start=2400 rotate-interval=60
```

See Also

flex-init

init-cgi

Applicable in Init-class directives.

The init-cgi function performs certain initialization tasks for CGI execution. Two options are provided: timeout of the execution of the CGI script, and establishment of environment variables.
Parameters
The following table describes parameters for the init-cgi function.

Table 2-7  init-cgi parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeout</td>
<td>(Optional) Specifies how many seconds the server waits for CGI output. If the CGI script has not delivered any output in that many seconds, the server terminates the script. The default is 300 seconds.</td>
</tr>
<tr>
<td>cgistub-path</td>
<td>(Optional) Specifies the path to the CGI stub binary. If not specified, Sun ONE Web Server looks in the following directories in the following order, relative to the server instance's config directory: ../private/Cgistub, then ../../bin/https/bin/Cgistub. Use the first directory to house an suid Cgistub (that is, a Cgistub owned by root that has the set-user-ID-on-exec bit set). Use the second directory to house a non-suid Cgistub. The second directory is the location used by Sun ONE Web Server 4.x servers. If present, the ../private directory must be owned by the server user and have permissions d??x------. This prevents other users (for example, users with shell accounts or CGI access) from using Cgistub to set their uid. For information about installing an suid Cgistub, see the Sun ONE Web Server 6.1 Programmer’s Guide.</td>
</tr>
<tr>
<td>env-variable</td>
<td>(Optional) Specifies the name and value for an environment variable that the server places into the environment for the CGI. You can set any number of environment variables in a single init-cgi function.</td>
</tr>
</tbody>
</table>

Example

```
Init fn=init-cgi LD_LIBRARY_PATH=/usr/lib;/usr/local/lib
```
init-clf

Applicable in Init-class directives.

The init-clf function opens the named log files to be used for common logging. The common-log function writes entries into the log files during the AddLog stage of the request-handling process. The log files stay open until the server is shut down (at which time the log files are closed) or restarted (at which time the log files are closed and reopened).

| NOTE | If the server has an AddLog-stage directive that calls common-log, common log files must be initialized by init-clf during initialization. |

| NOTE | This function should only be called once. If it is called again, the new call will replace log file names from all previous calls. |

If you move, remove, or change the log file without shutting down or restarting the server, client accesses might not be recorded. To save or backup a log file, you need to rename the file (and for UNIX, send the -HUP signal), and then restart the server. The server first looks for the log file by name, and if it doesn’t find it, creates a new one (the renamed original log file is left for you to use).

For information on rotating log files, see flex-rotate-init.

**Parameters**

The following table describes parameters for the init-clf function.
Applicable in Init-class directives.

The init-dav function performs initialization tasks to load the WebDAV plugin.

**Parameters**

This function requires a LateInit=yes parameter.
Example

```
Init fn="load-modules" shlib="/s1ws6.1/lib/libdavplugin.so"
func="init-dav,ntrans-dav,service-dav"
shlib_flags="(global|now)"
Init fn="init-dav" LateInit=yes
```

Example

```
Init fn=init-cgi LD_LIBRARY_PATH=/usr/lib;/usr/local/lib
```

**init-filter-order**

Applicable in Init-class directives.

The `init-filter-order` Init SAF can be used to control the position of specific filters within filter stacks. For example, `init-filter-order` can be used to ensure that a filter that converts outgoing XML to XHTML is inserted above a filter that converts outgoing XHTML to HTML.

Filters that appear higher in the filter stack are given an earlier opportunity to process outgoing data, and filters that appear lower in the filter stack are given an earlier opportunity to process incoming data.

The appropriate position of a specific filter within the filter stack is defined by the filter developer. For example, filters that translate content from XML to HTML are placed higher in the filter stack than filters that compress data for transmission. Filter developers use the `filter_create` function to define the filter's position in the filter stack. `init-filter-order` can be used to override the position defined by the filter developer.

When two or more filters are defined to occupy the same position in the filter stack, filters that were inserted later will appear higher than filters that were inserted earlier. That is, the order of Input `fn="insert-filter"` and Output `fn="insert-filter"` directives in obj.conf becomes important. For example, consider two filters, `xhtml-to-html` and `xml-to-xhtml`, which convert XHTML to HTML and XML to XHTML, respectively. Since both filters transform data from
one format to another, they may be defined to occupy the same position in the filter stack. To transform XML documents to XHTML and then to HTML before sending the data to the client, Output fn="insert-filter" directives in obj.conf would appear in the following order:

Output fn="insert-filter" filter="xhtml-to-html"
Output fn="insert-filter" filter="xml-to-xhtml"

In general, administrators should use the order of Input fn="insert-filter" and Output fn="insert-filter" directives in obj.conf to control the position of filters in the filter stack. init-filter-order should only be used to address specific filter interoperability problems.

| NOTE | The load-module SAFs that create the filters should be called before init-filter-order attempts to order them. |

Parameters
The following table describes parameters for the init-filter-order function.

| Table 2-9 | init-filter-order parameters |
| Parameter | Description |
| filters | Comma-separated list of filters in the order they should appear within a filter stack, listed from highest to lowest. |

Example

```
Init fn="init-filter-order"
filters="xml-to-xhtml,xhtml-to-html,http-compression"
```

init-j2ee

Applicable in Init-class directives.
The init-j2ee function initializes the Java subsystem.

Parameters
This function requires a LateInit=yes parameter.
**init-uhome**

Applicable in Init-class directives.

**UNIX Only.** The `init-uhome` function loads information about the system's user home directories into internal hash tables. This increases memory usage slightly, but improves performance for servers that have a lot of traffic to home directories.

**Parameters**

The following table describes parameters for the `init-uhome` function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pwfile</td>
<td>(Optional) Specifies the full file system path to a file other than <code>/etc/passwd</code>. If not provided, the default UNIX path (<code>/etc/passwd</code>) is used.</td>
</tr>
</tbody>
</table>

**Examples**

```
Init fn=init-uhome

Init fn=init-uhome pwfile=/etc/passwd-http
```
load-modules

Applicable in Init-class directives.

The load-modules function loads a shared library or dynamic-link library (DLL) into the server code. Specified functions from the library can then be executed from any subsequent directives. Use this function to load new plugins or SAFs.

If you define your own SAFs, you get the server to load them by using the load-modules function and specifying the shared library or DLL to load.

Parameters
The following table describes parameters for the load-modules function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shlib</td>
<td>Specifies either the full path to the shared library or DLL, or a file name relative to the server configuration directory.</td>
</tr>
<tr>
<td>funcs</td>
<td>Comma-separated list of the names of the functions in the shared library or DLL to be made available for use by other Init directives or by Service directives in obj.conf. The list should not contain any spaces. The dash (-) character may be used in place of the underscore (_) character in function names.</td>
</tr>
<tr>
<td>NativeThread</td>
<td>(Optional) Specifies which threading model to use.</td>
</tr>
<tr>
<td></td>
<td>no causes the routines in the library to use user-level threading;</td>
</tr>
<tr>
<td></td>
<td>yes enables kernel-level threading. The default is yes.</td>
</tr>
<tr>
<td>pool</td>
<td>Name of a custom thread pool, as specified in thread-pool-init.</td>
</tr>
</tbody>
</table>

Examples

```
Init fn=load-modules shlib="C:/mysrvfns/corpfn.dll"
funcs="moveit"

Init fn=load-modules shlib="/mysrvfns/corpfn.so"
funcs="myinit,myservice"
Init fn=myinit
```
nt-console-init

Applicable in Init-class directives.

The nt-console-init function enables the Windows console, which is the command-line shell that displays standard output and error streams.

Parameters
The following table describes parameters for the nt-console-init function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stderr</td>
<td>Directs error messages to the Windows console. The required and only value is console.</td>
</tr>
<tr>
<td>stdout</td>
<td>Directs output to the Windows console. The required and only value is console.</td>
</tr>
</tbody>
</table>

Example

Init fn="nt-console-init" stdout=console stderr=console

perf-init

Applicable in Init-class directives.

The perf-init function enables system performance measurement via performance buckets.

For more information about performance buckets, see the Sun ONE Web Server 6.1 Performance Tuning, Sizing, and Scaling Guide.

Parameters
The following table describes parameters for the perf-init function.
pool-init

Table 2-13  perf-init parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disable</td>
<td>Flag to disable the use of system performance measurement via performance buckets. Should have a value of true or false. Default value is true.</td>
</tr>
</tbody>
</table>

Example

```
Init fn=perf-init disable=false
```

See Also
define-perf-bucket

cell-init

Applicable in Init-class directives.

The pool-init function changes the default values of pooled memory settings. The size of the free block list may be changed or pooled memory may be entirely disabled.

Memory allocation pools allow the server to run significantly faster. If you are programming with the NSAPI, note that MALLOC, REALLOC, CALLOC, STRDUP, and FREE work slightly differently if pooled memory is disabled. If pooling is enabled, the server automatically cleans up all memory allocated by these routines when each request completes. In most cases, this will improve performance and prevent memory leaks. If pooling is disabled, all memory is global and there is no clean-up.

If you want persistent memory allocation, add the prefix PERM_ to the name of each routine (PERM_MALLOC, PERM_REALLOC, PERM_CALLOC, PERM_STRDUP, and PERM_FREE).
Parameters
The following table describes parameters for the **pool-init** function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>free-size</td>
<td>(Optional) Maximum size in bytes of free block list. May not be greater than 1048576.</td>
</tr>
<tr>
<td>disable</td>
<td>(Optional) Flag to disable the use of pooled memory. Should have a value of <code>true</code> or <code>false</code>. Default value is <code>false</code>.</td>
</tr>
</tbody>
</table>

Example

```
Init fn=pool-init disable=true
```

**register-http-method**

Applicable in **Init**-class directives.

This function lets you extend the HTTP protocol by registering new HTTP methods. (You do not need to register the default HTTP methods.)

Upon accepting a connection, the server checks if the method it received is known to it. If the server does not recognize the method, it returns a “501 Method Not Implemented” error message.

Parameters
The following table describes parameters for the **register-http-method** function.
Example
The following example shows the use of `register-http-method` and a Service function for one of the methods.

```
Init fn="register-http-method" methods="MY_METHOD1,MY_METHOD2"
Service fn="MyHandler" method="MY_METHOD1"
```

## stats-init

Applicable in `Init`-class directives.

The `stats-init` function enables reporting of performance statistics in XML format. The actual report is generated by the `stats-xml` function in `obj.conf`.

### Parameters

The following table describes parameters for the `stats-init` function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>update-interval</code></td>
<td>Period in seconds between statistics updates within the server. Set higher for better performance, lower for more frequent updates. The minimum value is 1; the default is 5.</td>
</tr>
<tr>
<td><code>virtual-servers</code></td>
<td>Maximum number of virtual servers for which statistics are tracked. This number should be set higher than the number of virtual servers configured. Smaller numbers result in lower memory usage. The minimum value is 1; the default is 1000.</td>
</tr>
<tr>
<td><code>profiling</code></td>
<td>Enables NSAPI performance profiling using buckets if set to <code>yes</code>. This can also be enabled through the <code>perf-init</code> Init SAF. The default is <code>no</code>, which results in slightly better server performance.</td>
</tr>
</tbody>
</table>
thread-pool-init

Applicable in Init-class directives.

The thread-pool-init function creates a new pool of user threads. A pool must be declared before it is used. To tell a plugin to use the new pool, specify the pool parameter when loading the plugin with the Init-class function load-modules.

One reason to create a custom thread pool would be if a plugin is not thread-aware, in which case you can set the maximum number of threads in the pool to 1.

The older parameter NativeThread=yes always engages one default native pool, called NativePool.

The native pool on UNIX is normally not engaged, as all threads are OS-level threads. Using native pools on UNIX may introduce a small performance overhead, as they’ll require an additional context switch; however, they can be used to localize the jvm.stickyAttach effect or for other purposes, such as resource control and management, or to emulate single-threaded behavior for plugins.

On Windows, the default native pool is always being used and Sun ONE Web Server uses fibers (user-scheduled threads) for initial request processing. Using custom additional pools on Windows introduces no additional overhead.

In addition, native thread pool parameters can be added to the magnus.conf file for convenience. For more information, see “Native Thread Pools” in the chapter “Syntax and Use of magnus.conf” in the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

Parameters

The following table describes parameters for the thread-pool-init function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the thread pool.</td>
</tr>
</tbody>
</table>
Table 2-17  thread-pool-init parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxthreads</td>
<td>Maximum number of threads in the pool.</td>
</tr>
<tr>
<td>minthreads</td>
<td>Minimum number of threads in the pool.</td>
</tr>
<tr>
<td>queueSize</td>
<td>Size of the queue for the pool. If all threads in the pool are busy, further request-handling threads that want to get a thread from the pool will wait in the pool queue. The number of request-handling threads that can wait in the queue is limited by the queue size. If the queue is full, the next request-handling thread that comes to the queue is turned away, with the result that the request is turned down, but the request-handling thread remains free to handle another request instead of becoming locked up in the queue.</td>
</tr>
<tr>
<td>stackSize</td>
<td>Stack size of each thread in the native (kernel) thread pool.</td>
</tr>
</tbody>
</table>

Example

```
Init fn=thread-pool-init name="my-custom-pool" maxthreads=5
  minthreads=1 queueSize=200
Init fn=load-modules shlib="C:/mydir/myplugin.dll"
  funcs="tracker" pool="my-custom-pool"
```

See Also

`load-modules`
This chapter describes how to write your own NSAPI plugins that define custom Server Application Functions (SAFs). Creating plugins allows you to modify or extend the Sun ONE Web Server's built-in functionality. For example, you can modify the server to handle user authorization in a special way or generate dynamic HTML pages based on information in a database.

This chapter has the following sections:

- Future Compatibility Issues
- The SAF Interface
- SAF Parameters
- Result Codes
- Creating and Using Custom SAFs
- Overview of NSAPI C Functions
- Required Behavior of SAFs for Each Directive
- CGI to NSAPI Conversion

Before writing custom SAFs, you should familiarize yourself with the request-handling process, as described in general in “Steps in the Request-handling Process” on page 24, and in greater detail in the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference. Also, before writing a custom SAF, check to see if a built-in SAF already accomplishes the tasks you have in mind.

See Chapter 2, “SAFs in the magnus.conf File” for a list of the predefined Init SAFs. For information about predefined SAFs used in the obj.conf file, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.
Future Compatibility Issues

For a complete list of the NSAPI routines for implementing custom SAFs, see Chapter 7, “NSAPI Function Reference.”

Future Compatibility Issues

The NSAPI interface may change in a future version of Sun ONE Web Server. To keep your custom plugins upgradeable, do the following:

- Make sure plugin users know how to edit the configuration files (such as magnus.conf and obj.conf) manually. The plugin installation software should not be used to edit these configuration files.
- Keep the source code so you can recompile the plugin.

The SAF Interface

All SAFs (custom and built-in) have the same C interface regardless of the request-handling step for which they are written. They are small functions designed for a specific purpose within a specific request-response step. They receive parameters from the directive that invokes them in the obj.conf file, from the server, and from previous SAFs.

Here is the C interface for a SAF:

```c
int function(pblock *pb, Session *sn, Request *rq);
```

The next section discusses the parameters in detail.

The SAF returns a result code that indicates whether and how it succeeded. The server uses the result code from each function to determine how to proceed with processing the request. See “Result Codes” on page 83 for details of the result codes.

SAF Parameters

This section discusses the SAF parameters in detail. The parameters are:

- **pb (parameter block)** -- contains the parameters from the directive that invokes the SAF in the obj.conf file.
- **sn (session)** -- contains information relating to a single TCP/IP session.
Chapter 3 Creating Custom SAFs

• **rq (request)** -- contains information relating to the current request.

**pb (parameter block)**

The **pb** parameter is a pointer to a **pblock** data structure that contains values specified by the directive that invokes the SAF. A **pblock** data structure contains a series of name-value pairs.

For example, a directive that invokes the **basic-ncsa** function might look like:

```
AuthTrans fn=basic-ncsa auth-type=basic
dbm=/sun/server61/userdb/rs
```

In this case, the **pb** parameter passed to **basic-ncsa** contains name-value pairs that correspond to **auth-type=basic** and **dbm=/Sun/WebServer61/server1/userdb/rs**.

NSAPI provides a set of functions for working with **pblock** data structures. For example, **pblock_findval()** returns the value for a given name in a **pblock**. See “Parameter Block Manipulation Routines” on page 92 for a summary of the most commonly used functions for working with parameter blocks.

**sn (session)**

The **sn** parameter is a pointer to a **session** data structure. This parameter contains variables related to an entire session (that is, the time between the opening and closing of the TCP/IP connection between the client and the server). The same **sn** pointer is passed to each SAF called within each request for an entire session. The following list describes the most important fields in this data structure (see Chapter 7, “NSAPI Function Reference” for information about NSAPI routines for manipulating the **session** data structure).

• **sn->client**

  Pointer to a **pblock** containing information about the client such as its IP address, DNS name, or certificate. If the client does not have a DNS name or if it cannot be found, it will be set to **-none**.

• **sn->csd**
Platform-independent client socket descriptor. You will pass this to the routines for reading from and writing to the client.

rq (request)

The rq parameter is a pointer to a request data structure. This parameter contains variables related to the current request, such as the request headers, URI, and local file system path. The same request pointer is passed to each SAF called in the request-response process for an HTTP request.

The following list describes the most important fields in this data structure (see Chapter 7, “NSAPI Function Reference” for information about NSAPI routines for manipulating the request data structure).

- rq->vars
  Pointer to a pblock containing the server’s “working” variables. This includes anything not specifically found in the following three pblocks. The contents of this pblock vary depending on the specific request and the type of SAF. For example, an AuthTrans SAF may insert an auth-user parameter into rq->vars which can be used subsequently by a PathCheck SAF.

- rq->reqpb
  Pointer to a pblock containing elements of the HTTP request. This includes the HTTP method (GET, POST, and so on), the URI, the protocol (normally HTTP/1.0), and the query string. This pblock does not normally change throughout the request-response process.

- rq->headers
  Pointer to a pblock containing all of the request headers (such as User-Agent, If-Modified-Since, and so on) received from the client in the HTTP request. See Chapter 12, “Hypertext Transfer Protocol” for more information about request headers. This pblock does not normally change throughout the request-response process.

- rq->srvhdrs
  Pointer to a pblock containing the response headers (such as Server, Date, Content-Type, Content-Length, and so on) to be sent to the client in the HTTP response. See Chapter 12, “Hypertext Transfer Protocol” for more information about response headers.
The \textit{rq} parameter is the primary mechanism for passing along information throughout the request-response process. On input to a SAF, \textit{rq} contains whatever values were inserted or modified by previously executed SAFs. On output, \textit{rq} contains any modifications or additional information inserted by the SAF. Some SAFs depend on the existence of specific information provided at an earlier step in the process. For example, a \texttt{PathCheck} SAF retrieves values in \texttt{rq->vars} that were previously inserted by an \texttt{AuthTrans} SAF.

**Result Codes**

Upon completion, a SAF returns a result code. The result code indicates what the server should do next. The result codes are:

- **REQ_PROCEED**
  Indicates that the SAF achieved its objective. For some request-response steps (\texttt{AuthTrans, NameTrans, Service, and Error}), this tells the server to proceed to the next request-response step, skipping any other SAFs in the current step. For the other request-response steps (\texttt{PathCheck, ObjectType, and AddLog}), the server proceeds to the next SAF in the current step.

- **REQ_NOACTION**
  Indicates that the SAF took no action. The server continues with the next SAF in the current server step.

- **REQ_ABORTED**
  Indicates that an error occurred and an HTTP response should be sent to the client to indicate the cause of the error. A SAF returning \texttt{REQ_ABORTED} should also set the HTTP response status code. If the server finds an \texttt{Error} directive matching the status code or reason phrase, it executes the SAF specified. If not, the server sends a default HTTP response with the status code and reason phrase plus a short HTML page reflecting the status code and reason phrase for the user. The server then goes to the first \texttt{AddLog} directive.

- **REQ_EXIT**
  Indicates the connection to the client was lost. This should be returned when the SAF fails in reading or writing to the client. The server then goes to the first \texttt{AddLog} directive.
Creating and Using Custom SAFs

Custom SAFs are functions in shared libraries that are loaded and called by the server. Follow these steps to create a custom SAF:

1. **Write the Source Code** using the NSAPI functions. Each SAF is written for a specific directive.

2. **Compile and Link** the source code to create a shared library (.so, .sl, or .dll) file.

3. **Load and Initialize the SAF** by editing the `magnus.conf` file to:
   - Load the shared library file containing your custom SAF(s)
   - Initialize the SAF if necessary

4. **Instruct the Server to Call the SAFs** by editing `obj.conf` to call your custom SAF(s) at the appropriate time.

5. **Restart the Server.

6. **Test the SAF** by accessing your server from a browser with a URL that triggers your function.

The following sections describe these steps in greater detail.

**Write the Source Code**

Write your custom SAFs using NSAPI functions. For a summary of some of the most commonly used NSAPI functions, see “Overview of NSAPI C Functions” on page 92. For information about available routines, see Chapter 7, “NSAPI Function Reference.”

For examples of custom SAFs, see `nsapi/examples/` in the server root directory, and also see Chapter 5, “Examples of Custom SAFs and Filters.”

The signature for all SAFs is:

```c
int function(pblock *pb, Session *sn, Request *rq);
```

For more details on the parameters, see “SAF Parameters” on page 80.

The Sun ONE Web Server runs as a multi-threaded single process. On UNIX platforms there are actually two processes (a parent and a child), for historical reasons. The parent process performs some initialization and forks the child process. The child process performs further initialization and handles all of the HTTP requests.
Keep the following in mind when writing your SAF:

- Write thread-safe code
- Blocking may affect performance
- Write small functions with parameters and configure them in obj.conf
- Carefully check and handle all errors (and log them so you can determine the source of problems and fix them)

If necessary, write an initialization function that performs initialization tasks required by your new SAFs. The initialization function has the same signature as other SAFs:

```c
int function(pblock *pb, Session *sn, Request *rq);
```

SAFs expect to be able to obtain certain types of information from their parameters. In most cases, parameter block (pblock) data structures provide the fundamental storage mechanism for these parameters. A pblock maintains its data as a collection of name-value pairs. For a summary of the most commonly used functions for working with pblock structures, see “Parameter Block Manipulation Routines” on page 92.

When defining a SAF, you do not specifically state which directive it is written for. However, each SAF must be written for a specific directive (such as AuthTrans, Service, and so on). Each directive expects its SAFs to behave in particular ways, and your SAF must conform to the expectations of the directive for which it was written. For details of what each directive expects of its SAFs, see “Required Behavior of SAFs for Each Directive” on page 96.

## Compile and Link

Compile and link your code with the native compiler for the target platform. For UNIX, use the `gmake` command. For Windows, use the `nmake` command. For Windows, use Microsoft Visual C++ 6.0 or newer. You must have an import list that specifies all global variables and functions to access from the server binary. Use the correct compiler and linker flags for your platform. Refer to the example Makefile in the `server_root/plugins/nsapi/examples` directory.

Adhere to the following guidelines for compiling and linking.

### Include Directory and nsapi.h File

Add the `server_root/plugins/include` (UNIX) or `server_root\plugins\include` (Windows) directory to your makefile to include the `nsapi.h` file.
Libraries

Add the `server_root/bin/https/lib` (UNIX) or `server_root\bin\https\bin` (Windows) library directory to your linker command.

The following table lists the library that you need to link to.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td><code>ns-httpd40.dll</code> (in addition to the standard Windows libraries)</td>
</tr>
<tr>
<td>HP-UX</td>
<td><code>libns-httpd40.sl</code></td>
</tr>
<tr>
<td>All other UNIX platforms</td>
<td><code>libns-httpd40.so</code></td>
</tr>
</tbody>
</table>

Linker Commands and Options for Generating a Shared Object

To generate a shared library, use the commands and options listed in the following table.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris™ Operating System (SPARC® Platform Edition)</td>
<td><code>ld -G</code> or <code>cc -G</code></td>
</tr>
<tr>
<td>Windows</td>
<td><code>link -LD</code></td>
</tr>
<tr>
<td>HP-UX</td>
<td><code>cc +Z -b -Wl,+s -Wl,-B,symbolic</code></td>
</tr>
<tr>
<td>AIX</td>
<td><code>cc -p 0 -berok -blibpath:$(LD_RPATH)</code></td>
</tr>
<tr>
<td>Compaq</td>
<td><code>cc -shared</code></td>
</tr>
<tr>
<td>Linux</td>
<td><code>gcc -shared</code></td>
</tr>
<tr>
<td>IRIX</td>
<td><code>cc -shared</code></td>
</tr>
</tbody>
</table>
Additional Linker Flags

Use the linker flags in the following table to specify which directories should be searched for shared objects during runtime to resolve symbols.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris SPARC</td>
<td>-R dir:dir</td>
</tr>
<tr>
<td>Windows</td>
<td>(no flags, but the ns-httpd40.dll file must be in the system PATH variable)</td>
</tr>
<tr>
<td>HP-UX</td>
<td>-Wl,+b,dir,dir</td>
</tr>
<tr>
<td>AIX</td>
<td>-blibpath:dir:dir</td>
</tr>
<tr>
<td>Compaq</td>
<td>-rpath dir:dir</td>
</tr>
<tr>
<td>Linux</td>
<td>-Wl,-rpath,dir:dir</td>
</tr>
<tr>
<td>IRIX</td>
<td>-Wl,-rpath,dir:dir</td>
</tr>
</tbody>
</table>

On UNIX, you can also set the library search path using the `LD_LIBRARY_PATH` environment variable, which must be set when you start the server.

Compiler Flags

The following table lists the flags and defines you need to use for compilation of your source code.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris SPARC</td>
<td>-DXP_UNIX -D_REENTRANT -KPIC -DSOLARIS</td>
</tr>
<tr>
<td>Windows</td>
<td>-DXP_WIN32 -DWIN32 /MD</td>
</tr>
<tr>
<td>HP-UX</td>
<td>-DXP_UNIX -D_REENTRANT -DHPUX</td>
</tr>
<tr>
<td>AIX</td>
<td>-DXP_UNIX -D_REENTRANT -DAIX $ (DEBUG)</td>
</tr>
<tr>
<td>Compaq</td>
<td>-DXP_UNIX -KPIC</td>
</tr>
<tr>
<td>Linux</td>
<td>-DLINUX -D_REENTRANT -fPIC</td>
</tr>
<tr>
<td>IRIX</td>
<td>-o32 -exceptions -DXP_UNIX -KPIC</td>
</tr>
<tr>
<td>All platforms</td>
<td>-MCC_HTTPD -NET_SSL</td>
</tr>
</tbody>
</table>
The following table lists the optional flags and defines you can use.

<table>
<thead>
<tr>
<th>Flag/Define</th>
<th>Platforms</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-DSPAPI20</td>
<td>All</td>
<td>Needed for the proxy utilities function include file putil.h</td>
</tr>
</tbody>
</table>

### Compiling 3.x Plugins on AIX

For AIX only, plugins built for 3.x versions of the server must be relinked to work with 4.x and 6.x versions. The files you need, which are in the `server_root/plugins/nsapi/examples/` directory, are as follows:

- The `Makefile` file has the `-G` option instead of the old `-bM:SRE -berok -brtl -bnoentry` options.

- A script, `relink_36plugin`, modifies a plugin built for 3.x versions of the server to work with 4.x and 6.x versions. The script's comments explain its use.

Sun ONE Web Server 4.x and 6.x versions are built on AIX 4.2, which natively supports runtime-linking. Because of this, NSAPI plugins, which reference symbols in the `ns-httpd` main executable, must be built with the `-G` option, which specifies that symbols must be resolved at runtime.

Previous versions of Sun ONE Web Server, however, were built on AIX 4.1, which did not support native runtime-linking. Sun ONE Web Server had specific additional software to enable plugins. No special runtime-linking directives were required to build plugins. Because of this, plugins that have been built for previous server versions on AIX will not work with Sun ONE Web Server 4.x and 6.x versions as they are.

However, they can easily be relinked to work with Sun ONE Web Server 4.x and 6.x versions. The `relink_36plugin` script relinks existing plugins. Only the existing plugin itself is required for the script; original source and `.o` files are not needed. More specific comments are in the script itself. Since all AIX versions from 4.2 onward natively support runtime-linking, no plugins for Sun ONE Web Server versions 4.x and later will need to be relinked.
Load and Initialize the SAF

For each shared library (plugin) containing custom SAFs to be loaded into the Sun ONE Web Server, add an Init directive that invokes the load-modules SAF to magnus.conf.

The syntax for a directive that calls load-modules is:

```
Init fn=load-modules shlib=[path]sharedlibname funcs="SAF1,...,SAFn"
```

- `shlib` is the local file system path to the shared library (plugin).
- `funcs` is a comma-separated list of function names to be loaded from the shared library. Function names are case-sensitive. You may use dash a (-) in place of an underscore (_) in function names. There should be no spaces in the function name list.

If the new SAFs require initialization, be sure that the initialization function is included in the `funcs` list.

For example, if you created a shared library `animations.so` that defines two SAFs `do_small_anim()` and `do_big_anim()` and also defines the initialization function `init_my_animations`, you would add the following directive to load the plugin:

```
Init fn=load-modules shlib=animations.so 
funcs="do_small_anim,do_big_anim,init_my_animations"
```

If necessary, also add an Init directive that calls the initialization function for the newly loaded plugin. For example, if you defined the function `init_my_new_SAF()` to perform an operation on the maxAnimLoop parameter, you would add a directive such as the following to magnus.conf:

```
Init fn=init_my_animations maxAnimLoop=5
```

Instruct the Server to Call the SAFs

Next, add directives to obj.conf to instruct the server to call each custom SAF at the appropriate time. The syntax for directives is:

```
Directive fn=function-name [name1="value1"]...[nameN="valueN"]
```

- `Directive` is one of the server directives, such as AuthTrans, Service, and so on.
• *function-name* is the name of the SAF to execute.

• *nameN*="*valueN*" are the names and values of parameters which are passed to the SAF.

Depending on what your new SAF does, you might need to add just one directive to *obj.conf*, or you might need to add more than one directive to provide complete instructions for invoking the new SAF.

For example, if you define a new *AuthTrans* or *PathCheck* SAF, you could just add an appropriate directive in the default object. However, if you define a new *Service* SAF to be invoked only when the requested resource is in a particular directory or has a new kind of file extension, you would need to take extra steps.

If your new *Service* SAF is to be invoked only when the requested resource has a new kind of file extension, you might need to add an entry to the MIME types file so that the *type* value gets set properly during the *ObjectType* stage. Then you could add a *Service* directive to the default object that specifies the desired *type* value.

If your new *Service* SAF is to be invoked only when the requested resource is in a particular directory, you might need to define a *NameTrans* directive that generates a *name* or *ppath* value that matches another object, and then in the new object you could invoke the new *Service* function.

For example, suppose your plugin defines two new SAFs, *do_small_anim()* and *do_big_anim()* which both take *speed* parameters. These functions run animations. All files to be treated as small animations reside in the directory D:/Sun/WebServer61/server1/docs/animations/small, while all files to be treated as full-screen animations reside in the directory D:/Sun/WebServer61/server1/docs/animations/fullscreen.

To ensure that the new animation functions are invoked whenever a client sends a request for either a small or full-screen animation, you would add *NameTrans* directives to the default object to translate the appropriate URLs to the corresponding path names and also assign a name to the request.

```
NameTrans fn=pfx2dir from="/animations/small"  
dir="D:/Sun/WebServer61/server1/docs/animations/small"  
name="small_anim"
NameTrans fn=pfx2dir from="/animations/fullscreen"  
dir="D:/Sun/WebServer61/server1/docs/animations/fullscreen"  
name="fullscreen_anim"
```
You also need to define objects that contain the Service directives that run the animations and specify the speed parameter.

```
<Object name="small_anim">
  Service fn=do_small_anim speed=40
</Object>
<Object name="fullscreen_anim">
  Service fn=do_big_anim speed=20
</Object>
```

### Restart the Server

After modifying obj.conf, you need to restart the server. A restart is required for all plugins that implement SAFs and/or filters.

### Test the SAF

Test your SAF by accessing your server from a browser with a URL that triggers your function. For example, if your new SAF is triggered by requests to resources in `http://server-name/animations/small`, try requesting a valid resource that starts with that URI.

You should disable caching in your browser so that the server is sure to be accessed. In Netscape Navigator you may hold the shift key while clicking the Reload button to ensure that the cache is not used. (Note that the shift-reload trick does not always force the client to fetch images from source if the images are already in the cache.)

You may also wish to disable the server cache using the cache-init SAF. Examine the access log and error log to help with debugging.
Overview of NSAPI C Functions

NSAPI provides a set of C functions that are used to implement SAFs. They serve several purposes. They provide platform independence across Sun ONE Web Server operating system and hardware platforms. They provide improved performance. They are thread-safe which is a requirement for SAFs. They prevent memory leaks. And they provide functionality necessary for implementing SAFs. You should always use these NSAPI routines when defining new SAFs.

This section provides an overview of the function categories available and some of the more commonly used routines. All of the public routines are detailed in Chapter 7, “NSAPI Function Reference.”

The main categories of NSAPI functions are:

- Parameter Block Manipulation Routines
- Protocol Utilities for Service SAFs
- Memory Management
- File I/O
- Network I/O
- Threads
- Utilities
- Virtual Server

Parameter Block Manipulation Routines

The parameter block manipulation functions provide routines for locating, adding, and removing entries in a pblock data structure:

- pblock_findval returns the value for a given name in a pblock.
- pblock_nvinsert adds a new name-value entry to a pblock.
- pblock_remove removes a pblock entry by name from a pblock. The entry is not disposed. Use param_free to free the memory used by the entry.
- param_free frees the memory for the given pblock entry.
- pblock_pblock2str creates a new string containing all of the name-value pairs from a pblock in the form “name=value name=value.” This can be a useful function for debugging.
Protocol Utilities for Service SAFs

Protocol utilities provide functionality necessary to implement Service SAFs:

- `request_header` returns the value for a given request header name, reading the headers if necessary. This function must be used when requesting entries from the browser header pblock (`rq->headers`).
- `protocol_status` sets the HTTP response status code and reason phrase.
- `protocol_start_response` sends the HTTP response and all HTTP headers to the browser.

Memory Management

Memory management routines provide fast, platform-independent versions of the standard memory management routines. They also prevent memory leaks by allocating from a temporary memory (called “pooled” memory) for each request, and then disposing the entire pool after each request. There are wrappers for standard memory routines for using permanent memory. To disable pooled memory for debugging, see the built-in SAF `pool-init` in Chapter 2, “SAFs in the magnus.conf File.”

- `MALLOC`
- `FREE`
- `PERM_STRDUP`
- `REALLOC`
- `CALLOC`
- `PERM_MALLOC`
- `PERM_FREE`
- `PERM_STRDUP`
- `PERM_REALLOC`
- `PERM_CALLOC`
File I/O

The file I/O functions provide platform-independent, thread-safe file I/O routines.

- `system_fopenRO` opens a file for read-only access.
- `system_fopenRW` opens a file for read-write access, creating the file if necessary.
- `system_fopenWA` opens a file for write-append access, creating the file if necessary.
- `system_fclose` closes a file.
- `system_fread` reads from a file.
- `system_fwrite` writes to a file.
- `system_fwrite_atomic` locks the given file before writing to it. This avoids interference between simultaneous writes by multiple threads.

Network I/O

Network I/O functions provide platform-independent, thread-safe network I/O routines. These routines work with SSL when it’s enabled.

- `netbuf_grab` reads from a network buffer’s socket into the network buffer.
- `netbuf_getc` gets a character from a network buffer.
- `net_flush` flushes buffered data.
- `net_read` reads bytes from a specified socket into a specified buffer.
- `net_sendfile` sends the contents of a specified file to a specified a socket.
- `net_write` writes to the network socket.

Threads

Thread functions include functions for creating your own threads that are compatible with the server’s threads. There are also routines for critical sections and condition variables.

- `systhread_start` creates a new thread.
- `systhread_sleep` puts a thread to sleep for a given time.
• crit_init  creates a new critical section variable.
• crit_enter  gains ownership of a critical section.
• crit_exit  surrenders ownership of a critical section.
• crit_terminate  disposes of a critical section variable.
• condvar_init  creates a new condition variable.
• condvar_notify  awakens any threads blocked on a condition variable.
• condvar_wait  blocks on a condition variable.
• condvar_terminate  disposes of a condition variable.
• prepare_nsapi_thread  allows threads that are not created by the server to act like server-created threads.

Utilities

Utility functions include platform-independent, thread-safe versions of many standard library functions (such as string manipulation), as well as new utilities useful for NSAPI.

• daemon_atrestart  (UNIX only) registers a user function to be called when the server is sent a restart signal (HUP) or at shutdown.
• condvar_init  gets the next line (up to a LF or CRLF) from a buffer.
• util_hostname  gets the local host name as a fully qualified domain name.
• util_later_than  compares two dates.
• util_sprintf  is the same as the standard library routine sprintf().
• util_strftime  is the same as the standard library routine strftime().
• util_uri_escape  converts the special characters in a string into URI-escaped format.
• util_uri_unescape  converts the URI-escaped characters in a string back into special characters.

NOTE  You cannot use an embedded null in a string, because NSAPI functions assume that a null is the end of the string. Therefore, passing unicode-encoded content through an NSAPI plugin doesn’t work.
Virtual Server

The virtual server functions provide routines for retrieving information about virtual servers.

- `request_get_vs` finds the virtual server to which a request is directed.
- `vs_alloc_slot` allocates a new slot for storing a pointer to data specific to a certain virtual server.
- `vs_get_data` finds the value of a pointer to data for a given virtual server and slot.
- `vs_get_default_httpd_object` obtains a pointer to the default (or root) object from the virtual server's virtual server class configuration.
- `vs_get_doc_root` finds the document root for a virtual server.
- `vs_get_httpd_objset` obtains a pointer to the virtual server class configuration for a given virtual server.
- `vs_get_id` finds the ID of a virtual server.
- `vs_get_mime_type` determines the MIME type that would be returned in the Content-Type: header for the given URI.
- `vs_lookup_config_var` finds the value of a configuration variable for a given virtual server.
- `vs_register_cb` allows a plugin to register functions that will receive notifications of virtual server initialization and destruction events.
- `vs_set_data` sets the value of a pointer to data for a given virtual server and slot.
- `vs_translate_uri` translates a URI as though it were part of a request for a specific virtual server.

Required Behavior of SAFs for Each Directive

When writing a new SAF, you should define it to do certain things, depending on which stage of the request-handling process will invoke it. For example, SAFs to be invoked during the Init stage must conform to different requirements than SAFs to be invoked during the Service stage.
The rq parameter is the primary mechanism for passing along information throughout the request-response process. On input to a SAF, rq contains whatever values were inserted or modified by previously executed SAFs. On output, rq contains any modifications or additional information inserted by the SAF. Some SAFs depend on the existence of specific information provided at an earlier step in the process. For example, a PathCheck SAF retrieves values in rq->vars that were previously inserted by an AuthTrans SAF.

This section outlines the expected behavior of SAFs used at each stage in the request-handling process.

- **Init SAFs**
- **AuthTrans SAFs**
- **NameTrans SAFs**
- **PathCheck SAFs**
- **ObjectType SAFs**
- **Input SAFs**
- **Output SAFs**
- **Service SAFs**
- **Error SAFs**
- **AddLog SAFs**

For more detailed information about these SAFs, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

**Init SAFs**

- **Purpose:** Initialize at startup.
- **Called at:** server startup and restart.
- rq and sn are NULL.
- Initialize any shared resources such as files and global variables.
- Can register callback function with daemon_atrestart() to clean up.
- On error, insert error parameter into pb describing the error and return REQ_ABORTED.
Required Behavior of SAFs for Each Directive

- If successful, return `REQ_PROCEED`.

**AuthTrans SAFs**

- **Purpose:** Verify any authorization information. Only basic authorization is currently defined in the HTTP/1.0 specification.

  - Check for `Authorization` header in `rq->headers` that contains the authorization type and uu-encoded user and password information. If header was not sent, return `REQ_NOACTION`.

  - If header exists, check authenticity of user and password.

  - If authentic, create `auth-type`, `plus auth-user and/or auth-group parameter` in `rq->vars` to be used later by PathCheck SAFs.

  - Return `REQ_PROCEED` if the user was successfully authenticated, `REQ_NOACTION` otherwise.

**NameTrans SAFs**

- **Purpose:** Convert logical URI to physical path.

  - Perform operations on logical path (`ppath in rq->vars`) to convert it into a full local file system path.

  - Return `REQ_PROCEED` if `ppath in rq->vars` contains the full local file system path, or `REQ_NOACTION` if not.

  - To redirect the client to another site, change `ppath in rq->vars` to `/URL`. Add `url` to `rq->vars` with full URL (for example, `http://home.netscape.com/`). Return `REQ_PROCEED`.

**PathCheck SAFs**

- **Purpose:** Check path validity and user’s access rights.

  - Check `auth-type`, `auth-user, and/or auth-group in rq->vars`.

  - Return `REQ_PROCEED` if user (and group) is authorized for this area (`ppath in rq->vars`).
Required Behavior of SAFs for Each Directive

- If not authorized, insert `WWW-Authenticate` to `rq->srvhdrs` with a value such as: `Basic; Realm="Our private area"`. Call `protocol_status()` to set HTTP response status to `PROTOCOL_UNAUTHORIZED`. Return `REQ_ABORTED`.

ObjectType SAFs

- Purpose: Determine `content-type` of data.
- If `content-type` in `rq->srvhdrs` already exists, return `REQ_NOACTION`.
- Determine the MIME type and create `content-type` in `rq->srvhdrs`.
- Return `REQ_PROCEED` if `content-type` is created, `REQ_NOACTION` otherwise.

Input SAFs

- Purpose: Insert filters that process incoming (client-to-server) data.
- Input SAFs are executed when a plugin or the server first attempts to read entity body data from the client.
- Input SAFs are executed at most once per request.
- Return `REQ_PROCEED` to indicate success, or `REQ_NOACTION` to indicate it performed no action.

Output SAFs

- Purpose: Insert filters that process outgoing (server-to-client) data.
- Output SAFs are executed when a plugin or the server first attempts to write entity body data from the client.
- Output SAFs are executed at most once per request.
- Return `REQ_PROCEED` to indicate success, or `REQ_NOACTION` to indicate it performed no action.
Service SAFs

- Purpose: Generate and send the response to the client.
- A Service SAF is only called if each of the optional parameters type, method, and query specified in the directive in obj.conf match the request.
- Remove existing content-type from rq->srvhdrs. Insert correct content-type in rq->srvhdrs.
- Create any other headers in rq->srvhdrs.
- Call protocol_status to set HTTP response status.
- Call protocol_start_response to send HTTP response and headers.
- Generate and send data to the client using net_write.
- Return REQ_PROCEED if successful, REQ_EXIT on write error, REQ_ABORTED on other failures.

Error SAFs

- Purpose: Respond to an HTTP status error condition.
- The Error SAF is only called if each of the optional parameters code and reason specified in the directive in obj.conf match the current error.
- Error SAFs do the same as Service SAFs, but only in response to an HTTP status error condition.

AddLog SAFs

- Purpose: Log the transaction to a log file.
- AddLog SAFs can use any data available in pb, sn, or rq to log this transaction.
- Return REQ_PROCEED.
You may have a need to convert a CGI variable into an SAF using NSAPI. Since the CGI environment variables are not available to NSAPI, you’ll retrieve them from the NSAPI parameter blocks. The table below indicates how each CGI environment variable can be obtained in NSAPI.

Keep in mind that your code must be thread-safe under NSAPI. You should use NSAPI functions that are thread-safe. Also, you should use the NSAPI memory management and other routines for speed and platform independence.

<table>
<thead>
<tr>
<th>CGI getenv()</th>
<th>NSAPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTH_TYPE</td>
<td>pblock_findval(&quot;auth-type&quot;, rq-&gt;vars);</td>
</tr>
<tr>
<td>AUTH_USER</td>
<td>pblock_findval(&quot;auth-user&quot;, rq-&gt;vars);</td>
</tr>
<tr>
<td>CONTENT_LENGTH</td>
<td>pblock_findval(&quot;content-length&quot;, rq-&gt;headers);</td>
</tr>
<tr>
<td>CONTENT_TYPE</td>
<td>pblock_findval(&quot;content-type&quot;, rq-&gt;headers);</td>
</tr>
<tr>
<td>GATEWAY_INTERFACE</td>
<td>&quot;CGI/1.1&quot;</td>
</tr>
<tr>
<td>HTTP_*</td>
<td>pblock_findval( &quot;*&quot;, rq-&gt;headers);</td>
</tr>
<tr>
<td></td>
<td>(* is lowercase; dash replaces underscore)</td>
</tr>
<tr>
<td>PATH_INFO</td>
<td>pblock_findval(&quot;path-info&quot;, rq-&gt;vars);</td>
</tr>
<tr>
<td>PATH_TRANSLATED</td>
<td>pblock_findval(&quot;path-translated&quot;, rq-&gt;vars);</td>
</tr>
<tr>
<td>QUERY_STRING</td>
<td>pblock_findval(&quot;query&quot;, rq-&gt;reqpb);</td>
</tr>
<tr>
<td></td>
<td>(GET only; POST puts query string in body data)</td>
</tr>
<tr>
<td>REMOTE_ADDR</td>
<td>pblock_findval(&quot;ip&quot;, sn-&gt;client);</td>
</tr>
<tr>
<td>REMOTE_HOST</td>
<td>session_dns(sn) ? session_dns(sn) : pblock_findval(&quot;ip&quot;, sn-&gt;client);</td>
</tr>
<tr>
<td>REMOTE_IDENT</td>
<td>pblock_findval( &quot;from&quot;, rq-&gt;headers);</td>
</tr>
<tr>
<td></td>
<td>(not usually available)</td>
</tr>
<tr>
<td>REMOTE_USER</td>
<td>pblock_findval(&quot;auth-user&quot;, rq-&gt;vars);</td>
</tr>
<tr>
<td>REQUEST_METHOD</td>
<td>pblock_findval(&quot;method&quot;, req-&gt;reqpb);</td>
</tr>
<tr>
<td>SCRIPT_NAME</td>
<td>pblock_findval(&quot;uri&quot;, req-&gt;reqpb);</td>
</tr>
<tr>
<td>SERVER_NAME</td>
<td>char *util_hostname();</td>
</tr>
<tr>
<td>SERVER_PORT</td>
<td>conf_getglobals()-&gt;Vport;</td>
</tr>
<tr>
<td></td>
<td>(as a string)</td>
</tr>
<tr>
<td>SERVER_PROTOCOL</td>
<td>pblock_findval(&quot;protocol&quot;, rq-&gt;reqpb);</td>
</tr>
</tbody>
</table>
Table 3-6  Parameter Blocks for CGI Variables

<table>
<thead>
<tr>
<th>CGI getenv()</th>
<th>NSAPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVER_SOFTWARE</td>
<td>MAGNUS_VERSION_STRING</td>
</tr>
<tr>
<td>Sun ONE-specific:</td>
<td></td>
</tr>
<tr>
<td>CLIENT_CERT</td>
<td>pblock_findval(&quot;auth-cert&quot;, rq-&gt;vars)</td>
</tr>
<tr>
<td>HOST</td>
<td>char *session_maxdns(sn);</td>
</tr>
<tr>
<td>HTTPS</td>
<td>security_active ? &quot;ON&quot; : &quot;OFF&quot;;</td>
</tr>
<tr>
<td>HTTPS_KEYSIZE</td>
<td>pblock_findval(&quot;keysize&quot;, sn-&gt;client);</td>
</tr>
<tr>
<td>HTTPS_SECRETKEYSIZE</td>
<td>pblock_findval(&quot;secret-keysize&quot;, sn-&gt;client);</td>
</tr>
<tr>
<td>QUERY</td>
<td>pblock_findval( query&quot;, rq-&gt;reqpb);</td>
</tr>
<tr>
<td>SERVER_URL</td>
<td>http_uri2url_dynamic(&quot;&quot;,&quot;, sn, rq);</td>
</tr>
</tbody>
</table>
This chapter describes how to create custom filters that can be used to intercept and possibly modify the content presented to or generated by another function.

This chapter has the following sections:

- Future Compatibility Issues
- The NSAPI Filter Interface
- Filter Methods
- Position of Filters in the Filter Stack
- Filters that Alter Content-Length
- Creating and Using Custom Filters
- Overview of NSAPI Functions for Filter Development

Future Compatibility Issues

The NSAPI interface may change in a future version of Sun ONE Web Server. To keep your custom plugins upgradeable, do the following:

- Make sure plugin users know how to edit the configuration files (such as `magnus.conf` and `obj.conf`) manually. The plugin installation software should not be used to edit these configuration files.
- Keep the source code so you can recompile the plugin.
The NSAPI Filter Interface

Sun ONE Web Server 6.1 extends NSAPI by introducing a new filter interface that complements the existing Server Application Function (SAF) interface. Filters make it possible to intercept and possibly modify data sent to and from the server. The server communicates with a filter by calling the filter’s filter methods. Each filter implements one or more filter methods. A filter method is a C function that performs a specific operation, such as processing data sent by the server.

Filter Methods

This section describes the filter methods that a filter can implement. To create a filter, a filter developer implements one or more of these methods. This section describes the following filter methods:

- `insert`
- `remove`
- `flush`
- `read`
- `write`
- `writev`
- `sendfile`

For more information about these methods, see Chapter 7, "NSAPI Function Reference."

C Prototypes for Filter Methods

Following is a list of C prototypes for the filter methods:

```c
int insert(FilterLayer *layer, pblock *pb);
void remove(FilterLayer *layer);
int flush(FilterLayer *layer);
int read(FilterLayer *layer, void *buf, int amount, int timeout);
int write(FilterLayer *layer, const void *buf, int amount);
int writev(FilterLayer *layer, const struct iovec *iov, int iov_size);
int sendfile(FilterLayer *layer, sendfiledata *sfd);
```
The `layer` parameter is a pointer to a `FilterLayer` data structure, which contains variables related to a particular instance of a filter. Following is a list of the most important fields in the `FilterLayer` data structure:

- `context->sn`: Contains information relating to a single TCP/IP session (the same `sn` pointer that’s passed to SAFs).
- `context->rq`: Contains information relating to the current request (the same `rq` pointer that’s passed to SAFs).
- `context->data`: Pointer to filter-specific data.
- `lower`: A platform-independent socket descriptor used to communicate with the next filter in the stack.

The meaning of the `context->data` field is defined by the filter developer. Filters that must maintain state information across filter method calls can use `context->data` to store that information.

For more information about `FilterLayer`, see “`FilterLayer`” on page 258.

**insert**

The `insert` filter method is called when an SAF such as `insert-filter` calls the `filter_insert` function to request that a specific filter be inserted into the filter stack. Each filter must implement the `insert` filter method.

When `insert` is called, the filter can determine whether it should be inserted into the filter stack. For example, the filter could inspect the `Content-Type` header in the `rq->srvhdrspblock` to determine whether it is interested in the type of data that will be transmitted. If the filter should not be inserted, the `insert` filter method should indicate this by returning `REQ_NOACTION`.

If the filter should be inserted, the `insert` filter method provides an opportunity to initialize this particular instance of the filter. For example, the `insert` method could allocate a buffer with `MALLOC` and store a pointer to that buffer in `layer->context->data`.

The filter is not part of the filter stack until after `insert` returns. As a result, the `insert` method should not attempt to read from, write to, or otherwise interact with the filter stack.

**See Also**

`insert` in “NSAPI Function Reference”
Filter Methods

remove

The *remove* filter method is called when a filter stack is destroyed (that is, when the corresponding socket descriptor is closed), when the server finishes processing the request the filter was associated with, or when an SAF such as `remove-filter` calls the *filter_remove* function. The *remove* filter method is optional.

The *remove* method can be used to clean up any data the filter allocated in *insert* and to pass any buffered data to the next filter by calling `net_write(layer->lower, ...)`. The *remove* method is optional.

See Also

*remove* in “NSAPI Function Reference”

flush

The *flush* filter method is called when a filter or SAF calls the *net_flush* function. The *flush* method should pass any buffered data to the next filter by calling `net_write(layer->lower, ...)`. The *flush* method is optional, but it should be implemented by any filter that buffers outgoing data.

See Also

*flush* in “NSAPI Function Reference”

read

The *read* filter method is called when a filter or SAF calls the *net_read* function. Filters that are interested in incoming data (data sent from a client to the server) implement the *read* filter method.

Typically, the *read* method will attempt to obtain data from the next filter by calling `net_read(layer->lower, ...)`. The *read* method may then modify the received data before returning it to its caller.

See Also

*read* in “NSAPI Function Reference”
write

The write filter method is called when a filter or SAF calls the net_write function. Filters that are interested in outgoing data (data sent from the server to a client) implement the write filter method.

Typically, the write method will pass data to the next filter by calling net_write(layer->lower, ...). The write method may modify the data before calling net_write. For example, the http-compression filter compresses data before passing it on to the next filter.

If a filter implements the write filter method but does not pass the data to the next layer before returning to its caller (that is, if the filter buffers outgoing data), the filter should also implement the flush method.

See Also
write in “NSAPI Function Reference”

writev

The writev filter method performs the same function as the write filter method, but the format of its parameters is different. It is not necessary to implement the writev filter method; if a filter implements the write filter method but not the writev filter method, the server uses the write method instead of the writev method. A filter should not implement the writev method unless it also implements the write method.

Under some circumstances, the server may run slightly faster when filters that implement the write filter method also implement the writev filter method.

See Also
writev in “NSAPI Function Reference”

sendfile

The sendfile filter method performs a function similar to the writev filter method, but it sends a file directly instead of first copying the contents of the file into a buffer. It is not necessary to implement the sendfile filter method; if a filter implements the write filter method but not the sendfile filter method, the server will use the write method instead of the sendfile method. A filter should not implement the sendfile method unless it also implements the write method.
Position of Filters in the Filter Stack

Under some circumstances, the server may run slightly faster when filters that implement the write filter method also implement the sendfile filter method.

See Also
sendfile in “NSAPI Function Reference”

Position of Filters in the Filter Stack

All data sent to the server (such as the result of an HTML form) or sent from the server (such as the output of a JSP page) is passed through a set of filters known as a filter stack. The server creates a separate filter stack for each connection. While processing a request, individual filters can be inserted into and removed from the stack.

Different types of filters occupy different positions within a filter stack. Filters that deal with application-level content (such filters that translates a page from XHTML to HTML) occupy a higher position than filters that deal with protocol-level issues (such as filters that format HTTP responses). When two or more filters are defined to occupy the same position in the filter stack, filters that were inserted later will appear higher than filters that were inserted earlier.

Filters positioned higher in the filter stack are given an earlier opportunity to process outgoing data, while filters positioned lower in the stack are given an earlier opportunity to process incoming data. For example, in the following figure, the xml-to-xhtml filter is given an earlier opportunity to process outgoing data than the xhtml-to-html filter.
When you create a filter with the `filter_create` function, you specify what position your filter should occupy in the stack. You can also use the `init-filter-order` Init SAF to control the position of specific filters within filter stacks. For example, `init-filter-order` can be used to ensure that a filter that converts outgoing XML to XHTML is inserted above a filter that converts outgoing XHTML to HTML.

For more information, see “filter_create” on page 167 and “init-filter-order” on page 69.
Filters that Alter Content-Length

Filters that Alter Content-Length

Filters that can alter the length of an incoming request body or outgoing response body must take special steps to ensure interoperability with other filters and SAFs.

Filters that process incoming data are referred to as input filters. If an input filter can alter the length of the incoming request body (for example, if a filter decompresses incoming data) and there is a `Content-Length` header in the `rq->headers` `pblock`, the filter's `insert` filter method should remove the `Content-Length` header and replace it with a `Transfer-encoding: identity` header as follows:

```
pb_param *pp;

pp = pblock_remove("content-length",
layer->context->rq->headers);
if (pp != NULL) {
    param_free(pp);
    pblock_nvinsert("transfer-encoding", "identity",
layer->context->rq->headers);
}
```

Because some SAFs expect a `Content-Length` header when a request body is present, before calling the first `Service` SAF the server will insert all relevant filters, read the entire request body, and compute the length of the request body after it has been passed through all input filters. However, by default, the server will read at most 8192 bytes of request body data. If the request body exceeds 8192 bytes after being passed through the relevant input filters, the request will be cancelled. For more information, see the description of `ChunkedRequestBufferSize` in the "Syntax and Use of magnus.conf" chapter in the Sun ONE Web Server 6.1 Administrator's Configuration File Reference.

Filters that process outgoing data are referred to as output filters. If an output filter can alter the length of the outgoing response body (for example, if the filter compresses outgoing data), the filter's `insert` filter method should remove the `Content-Length` header from `rq->srvhdrs` as follows:
Creating and Using Custom Filters

Custom filters are defined in shared libraries that are loaded and called by the server. The general steps for creating a custom filter are as follows:

1. **Write the Source Code** using the NSAPI functions.
2. **Compile and Link** the source code to create a shared library (.so, .sl, or .dll) file.
3. **Load and Initialize the Filter** by editing the `magnus.conf` file.
4. **Instruct the Server to Insert the Filter** by editing the `obj.conf` file to insert your custom filter(s) at the appropriate time.
5. **Restart the Server**.
6. **Test the Filter** by accessing your server from a browser with a URL that triggers your filter.

These steps are described in greater detail in the following sections.

**Write the Source Code**

Write your custom filter methods using NSAPI functions. For a summary of the NSAPI functions specific to filter development, see “Overview of NSAPI Functions for Filter Development” on page 114. For a summary of general purpose NSAPI functions, see “Overview of NSAPI Functions for Filter Development” on page 114. Each filter method must be implemented as a separate function. See “Filter Methods” on page 104 for the filter method prototypes.

The filter must be created by a call to `filter_create`. Typically, each plugin defines an `nsapi_module_init` function that is used to call `filter_create` and perform any other initialization tasks. See `nsapi_module_init` and `filter_create` for more information.
Filter methods are invoked whenever the server or an SAF calls certain NSAPI functions such as `net_write` or `filter_insert`. As a result, filter methods can be invoked from any thread and should only block using NSAPI functions (for example, `crit_enter` and `net_read`). If a filter method blocks using other functions (for example, the Windows `WaitForMultipleObjects` and `ReadFile` functions), the server may hang. Also, shared objects that define filters should be loaded with the `NativeThread="no"` flag, as described in “Load and Initialize the Filter” on page 112.

If a filter method must block using a non-NSAPI function, `KernelThreads 1` should be set in `magnus.conf`. For more information about `KernelThreads`, see the description in the chapter "Syntax and Use of magnus.conf" in the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

Keep the following in mind when writing your filter:

- Write thread-safe code
- IO should only be performed using the NSAPI functions documented in “File I/O” on page 94 and “Network I/O” on page 94
- Thread synchronization should only be performed using NSAPI functions documented in “Threads” on page 94
- Blocking may affect performance.
- Carefully check and handle all errors

For examples of custom filters, see `server_root/plugins/nsapi/examples` and also Chapter 5, “Examples of Custom SAFs and Filters.”

Compile and Link

Filters are compiled and linked in the same way as SAFs. See “Compile and Link” on page 85 in the “Creating Custom SAFs” chapter for more information.

Load and Initialize the Filter

For each shared library (plugin) containing custom SAFs to be loaded into the Sun ONE Web Server, add an `Init` directive that invokes the `load-modules` SAF to `magnus.conf`. The syntax for a directive that loads a filter plugin is:

```
Init fn=load-modules shlib=[path]sharedlibname NativeThread="no"
```

- `shlib` is the local file system path to the shared library (plugin).
NativeThread indicates whether the plugin requires native threads. Filters should be written to run on any type of thread (see “Write the Source Code” on page 111).

When the server encounters such a directive, it calls the plugin’s nsapi_module_init function to initialize the filter.

Instruct the Server to Insert the Filter

Add an Input or Output directive to obj.conf to instruct the server to insert your filter into the filter stack. The format of the directive is as follows:

```
Directive fn=insert-filter filter="filter-name" [name1="value1"]...
   [nameN="valueN"]
```

- **Directive** is Input or Output.
- **filter-name** is the name of the filter, as passed to filter_create, to insert.
- **nameN="valueN"** are the names and values of parameters that are passed to the filter’s insert filter method.

Filters that process incoming data should be inserted using an Input directive. Filters that process outgoing data should be inserted using an Output directive.

To ensure that your filter is inserted whenever a client sends a request, add the Input or Output directive to the default object. For example, the following portion of obj.conf instructs the server to insert a filter named example-replace and pass it two parameters, from and to:

```
<Object name="default">
   Output fn=insert-filter
       filter="example-replace"
       from="Old String"
       to="New String"
   ...
</Object>
```
Restart the Server

For the server to load your plugin, you must restart the server. A restart is required for all plugins that implement SAFs and/or filters.

Test the Filter

Test your SAF by accessing your server from a browser. You should disable caching in your browser so that the server is sure to be accessed. In Netscape Navigator, you can hold the shift key while clicking the Reload button to ensure that the cache is not used. (Note that the shift-reload trick does not always force the client to fetch images from source if the images are already in the cache.) Examine the access and error logs to help with debugging.

Overview of NSAPI Functions for Filter Development

NSAPI provides a set of C functions that are used to implement SAFs and filters. This section lists the functions that are specific to the development of filters. All of the public routines are described in detail in Chapter 7, “NSAPI Function Reference.”

The NSAPI functions specific to the development of filters are:

- `filter_create` creates a new filter
- `filter_insert` inserts the specified filter into a filter stack
- `filter_remove` removes the specified filter from a filter stack
- `filter_name` returns the name of the specified filter
- `filter_find` finds an existing filter given a filter name
- `filter_layer` returns the layer in a filter stack that corresponds to the specified filter
This chapter provides examples of custom Sever Application Functions (SAFs) and filters for each directive in the request-response process. You may wish to use these examples as the basis for implementing your own custom SAFs and filters. For more information about creating your own custom SAFs, see Chapter 3, “Creating Custom SAFs.” For more information about creating your own filters, see Chapter 4, “Creating Custom Filters.”

Before writing custom SAFs, you should be familiar with the request-response process and the role of the configuration file `obj.conf` (this file is discussed in the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference).

Before writing your own SAF, check to see if an existing SAF serves your purpose. The predefined SAFs are discussed in the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

For a list of the NSAPI functions for creating new SAFs, see Chapter 7, “NSAPI Function Reference.”

This chapter has the following sections:

- Examples in the Build
- AuthTrans Example
- NameTrans Example
- PathCheck Example
- ObjectType Example
- Output Example
- Service Example
Examples in the Build

- AddLog Example
- Quality of Service Example

Examples in the Build

The nsapi/examples/ or plugins/nsapi/examples subdirectory within the server installation directory contains examples of source code for SAFs.

You can use the example.mak makefile in the same directory to compile the examples and create a library containing the functions in all of the example files.

To test an example, load the examples shared library into the Sun ONE Web Server by adding the following directive in the Init section of magnus.conf:

```
Init fn=load-modules shlib=examples.so/dll
  funcs=function1,function2,function3
```

The `funcs` parameter specifies the functions to load from the shared library.

If the example uses an initialization function, be sure to specify the initialization function in the `funcs` argument to load-modules, and also add an Init directive to call the initialization function.

For example, the PathCheck example implements the restrict-by-acf function, which is initialized by the acf-init function. The following directive loads both these functions:

```
Init fn=load-modules yourlibrary  funcs=acf-init,restrict-by-acf
```

The following directive calls the acf-init function during server initialization:

```
Init fn=acf-init file=extra-arg
```

To invoke the new SAF at the appropriate step in the response handling process, add an appropriate directive in the object to which it applies, for example:

```
PathCheck fn=restrict-by-acf
```

After adding new Init directives to magnus.conf, you always need to restart the Sun ONE Web Server to load the changes, since Init directives are only applied during server initialization.
AuthTrans Example

This simple example of an AuthTrans function demonstrates how to use your own custom ways of verifying that the user name and password that a remote client provided is accurate. This program uses a hard-coded table of user names and passwords and checks a given user’s password against the one in the static data array. The userdb parameter is not used in this function.

AuthTrans directives work in conjunction with PathCheck directives. Generally, an AuthTrans function checks if the user name and password associated with the request are acceptable, but it does not allow or deny access to the request; it leaves that to a PathCheck function.

AuthTrans functions get the user name and password from the headers associated with the request. When a client initially makes a request, the user name and password are unknown so the AuthTrans function and PathCheck function work together to reject the request, since they can’t validate the user name and password. When the client receives the rejection, the usual response is for it to present a dialog box asking the user for their user name and password, and then the client submits the request again, this time including the user name and password in the headers.

In this example, the hardcoded-auth function, which is invoked during the AuthTrans step, checks if the user name and password correspond to an entry in the hard-coded table of users and passwords.

Installing the Example

To install the function on the Sun ONE Web Server, add the following Init directive to magnus.conf to load the compiled function:

```
Init fn=load-modules shlib=yourlibrary funcs=hardcoded-auth
```

Inside the default object in obj.conf, add the following AuthTrans directive:

```
AuthTrans fn=basic-auth auth-type="basic" userfn=hardcoded-auth userdb=unused
```

Note that this function does not actually enforce authorization requirements, it only takes given information and tells the server if it’s correct or not. The PathCheck function require-auth performs the enforcement, so add the following PathCheck directive as well:
PathCheck fn=require-auth realm="test realm" auth-type="basic"

Source Code

The source code for this example is in the auth.c file in the nsapi/examples/ or plugins/nsapi/examples subdirectory of the server root directory.

#include "nsapi.h"

typedef struct {
    char *name;
    char *pw;
} user_s;

static user_s user_set[] = {
    {"joe", "shmoe"},
    {"suzy", "creamcheese"},
    {NULL, NULL}
};

#include "frame/log.h"

#ifdef __cplusplus
extern "C"
#endif

NSAPI_PUBLIC int hardcoded_auth(pblock *param, Session *sn, Request *rq) {
    /* Parameters given to us by auth-basic */
    char *pwfile = pblock_findval("userdb", param);
    char *user = pblock_findval("user", param);
    char *pw = pblock_findval("pw", param);
    /* Temp variables */
    register int x;

    for(x = 0; user_set[x].name != NULL; ++x) {
        /* If this isn't the user we want, keep going */
        if(strcmp(user, user_set[x].name) != 0) continue;

        /* Verify password */
        if(strcmp(pw, user_set[x].pw)) {
            log_error(LOG_SECURITY, "hardcoded-auth", sn, rq,
                    "user %s entered wrong password", user);
            /* This will cause the enforcement function to ask */
The `ntrans.c` file in the `nsapi/examples/` or `plugins/nsapi/examples` subdirectory of the server root directory contains source code for two example NameTrans functions:

- **explicit_pathinfo**
  
  This example allows the use of explicit extra path information in a URL.

- **https_redirect**
  
  This example redirects the URL if the client is a particular version of Netscape Navigator.

This section discusses the first example. Look at the source code in `ntrans.c` for the second example.

---

**NOTE**

A NameTrans function is used primarily to convert the logical URL in `ppath` in `rq->vars` to a physical path name. However, the example discussed here, `explicit_pathinfo`, does not translate the URL into a physical path name; it changes the value of the requested URL. See the second example, `https_redirect`, in `ntrans.c` for an example of a NameTrans function that converts the value of `ppath` in `rq->vars` from a URL to a physical path name.

The `explicit_pathinfo` example allows URLs to explicitly include extra path information for use by a CGI program. The extra path information is delimited from the main URL by a specified separator, such as a comma.
For example:

http://server-name/cgi/marketing,/jan/releases/hardware

In this case, the URL of the requested resource (which would be a CGI program) is http://server-name/cgi/marketing, and the extra path information to give to the CGI program is /jan/releases/hardware.

When choosing a separator, be sure to pick a character that will never be used as part of the real URL.

The explicit_pathinfo function reads the URL, strips out everything following the comma, and puts it in the path-info field of the vars field in the request object (rq->vars). CGI programs can access this information through the PATH_INFO environment variable.

One side effect of explicit_pathinfo is that the SCRIPT_NAME CGI environment variable has the separator character tacked onto the end.

NameTrans directives usually return REQ_PROCEED when they change the path, so that the server does not process any more NameTrans directives. However, in this case we want name translation to continue after we have extracted the path info, since we have not yet translated the URL to a physical path name.

Installing the Example

To install the function on the Sun ONE Web Server, add the following Init directive to magnus.conf to load the compiled function:

Init fn=load-modules shlib=yourlibrary funcs=explicit-pathinfo

Inside the default object in obj.conf, add the following NameTrans directive:

NameTrans fn=explicit-pathinfo separator="",

This NameTrans directive should appear before other NameTrans directives in the default object.

Source Code

This example is in the ntrans.c file in the nsapi/examples/ or plugins/nsapi/examples subdirectory of the server root directory.

#include "nsapi.h"
#include <string.h> /* strchr */
#include "frame/log.h" /* log_error */
#ifdef __cplusplus
extern "C"
#endif

NSAPI_PUBLIC int explicit_pathinfo(pblock *pb, Session *sn, Request *rq)
{
    /* Parameter: The character to split the path by */
    char *sep = pblock_findval("separator", pb);

    /* Server variables */
    char *ppath = pblock_findval("ppath", rq->vars);

    /* Temp var */
    char *t;

    /* Verify correct usage */
    if(!sep) {
        log_error(LOG_MISCONFIG, "explicit-pathinfo", sn, rq,
                  "missing parameter (need root)");
        /* When we abort, the default status code is 500 Server
         * Error */
        return REQ_ABORTED;
    }

    /* Check for separator. If not there, don't do anything */
    t = strchr(ppath, sep[0]);
    if(!t)
        return REQ_NOACTION;

    /* Truncate path at the separator */
    *t++ = '\0';

    /* Assign path information */
    pblock_nvinsert("path-info", t, rq->vars);

    /* Normally NameTrans functions return REQ_PROCEED when they
     * change the path. However, we want name translation to
     * continue after we're done. */
    return REQ_NOACTION;
}

#include "base/util.h"    /* is_mozilla */
#include "frame/protocol.h"   /* protocol_status */
#include "base/shexp.h"     /* shexp_cmp */

#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC int https_redirect(pblock *pb, Session *sn, Request *rq)
{
    /* Server Variable */
    char *ppath = pblock_findval("ppath", rq->vars);
    /* Parameters */
    char *from = pblock_findval("from", pb);
    char *url = pblock_findval("url", pb);
    char *alt = pblock_findval("alt", pb);
    /* Work vars */
    char *ua;
    /* Check usage */
    if((!from) || (!url)) {
        log_error(LOG_MISCONFIG, "https-redirect", sn, rq,
                    "missing parameter (need from, url)");
        return REQ_ABORTED;
    }
    /* Use wildcard match to see if this path is one we should redirect */
    if(shexp_cmp(ppath, from) != 0)
        return REQ_NOACTION;   /* no match */
    /* Sigh. The only way to check for SSL capability is to check UA */
    if(request_header("user-agent", &ua, sn, rq) == REQ_ABORTED)
        return REQ_ABORTED;
    /* The is_mozilla function checks for Mozilla version 0.96 or greater */
    if(util_is_mozilla(ua, "0", "96")) {
        /* Set the return code to 302 Redirect */
        protocol_status(sn, rq, PROTOCOL_REDIRECT, NULL);
        /* The error handling functions use this to set the Location: */
        pblock_nvinsert("url", url, rq->vars);
        return REQ_ABORTED;
    }
    /* No match. Old client. */
    /* If there is an alternate document specified, use it. */
    if(alt) {
        pb_param *pp = pblock_find("ppath", rq->vars);
        /* Trash the old value */
        FREE(pp->value);
        /* We must dup it because the library will later free this pblock */
        pp->value = STRDUP(alt);
        */
PathCheck Example

The example in this section demonstrates how to implement a custom SAF for performing path checks. This example simply checks if the requesting host is on a list of allowed hosts.

The `Init` function `acf-init` loads a file containing a list of allowable IP addresses with one IP address per line. The `PathCheck` function `restrict_by_acf` gets the IP address of the host that is making the request and checks if it is on the list. If the host is on the list, it is allowed access; otherwise, access is denied.

For simplicity, the stdio library is used to scan the IP addresses from the file.

Installing the Example

To load the shared object containing your functions, add the following line in the `Init` section of the `magnus.conf` file:

```plaintext
Init fn=load-modules yourlibrary funcs=acf-init,restrict-by-acf
```

To call `acf-init` to read the list of allowable hosts, add the following line to the `Init` section in `magnus.conf`. (This line must come after the one that loads the library containing `acf-init`).

```plaintext
Init fn=acf-init file=fileContainingHostsList
```

To execute your custom SAF during the request-response process for some object, add the following line to that object in the `obj.conf` file:

```plaintext
PathCheck fn=restrict-by-acf
```
Source Code

The source code for this example is in pcheck.c in the nsapi/examples/ or plugins/nsapi/examples subdirectory within the server root directory.

```c
#include "nsapi.h"

static char **hosts = NULL;

#include <stdio.h>
#include "base/daemon.h"
#include "base/util.h" /* util_sprintf */
#include "frame/log.h" /* log_error */
#include "frame/protocol.h" /* protocol_status */

#define MAX_ACF_LINE 256

/* The longest line we'll allow in an access control file */

int acf_init(pblock *pb, Session *sn, Request *rq)
{
    register int x;
    for(x = 0; hosts[x]; ++x)
        FREE(hosts[x]);
    FREE(hosts);
    hosts = NULL;
}
```

```c
#include "nsapi.h"

static char **hosts = NULL;

#include <stdio.h>
#include "base/daemon.h"
#include "base/util.h" /* util_sprintf */
#include "frame/log.h" /* log_error */
#include "frame/protocol.h" /* protocol_status */

#define MAX_ACF_LINE 256

/* The longest line we'll allow in an access control file */

int acf_init(pblock *pb, Session *sn, Request *rq)
{
    register int x;
    for(x = 0; hosts[x]; ++x)
        FREE(hosts[x]);
    FREE(hosts);
    hosts = NULL;
}
```
/* Check usage. Note that Init functions have special 
error logging */
if(!acf_file) {
    util_sprintf(err, "missing parameter to acf_init
    (need file)");
    pbblock_nvinsert("error", err, pb);
    return REQ_ABORTED;
}

f = fopen(acf_file, "r");
/* Did we open it? */
if(!f) {
    util_sprintf(err, "can't open access control file %s (%s)",
        acf_file, system_errmsg());
    pbblock_nvinsert("error", err, pb);
    return REQ_ABORTED;
}

/* Initialize hosts array */
um_hosts = 0;
hosts = (char **) MALLOC(1 * sizeof(char *));
hosts[0] = NULL;
while(fgets(buf, MAX_ACF_LINE, f)) {
    /* Blast linefeed that stdio helpfully leaves on there */
    buf[strlen(buf) - 1] = '\0';
    hosts = (char **) REALLOC(hosts, (num_hosts + 2) *
        sizeof(char *));
    hosts[num_hosts++] = STRDUP(buf);
    hosts[num_hosts] = NULL;
}

fclose(f);
/* At restart, free hosts array */
daemon_atrestart(acf_free, NULL);
return REQ_PROCEED

#ifdef __cplusplus
extern "C"
#endif

NSAPI_PUBLIC int restrict_by_acf(pblock *pb, Session *sn, Request *
rq)
{
    /* No parameters */
/* Working variables */
char *remip = pblock_findval("ip", sn->client);
register int x;
if(!hosts) {
    log_error(LOG_MISCONFIG, "restrict-by-acf", sn, rq,
        "restrict-by-acf called without call to acf-init");
    /* When we abort, the default status code is 500 Server
       Error */
    return REQ_ABORTED;
}
for(x = 0; hosts[x] != NULL; ++x) {
    /* If they're on the list, they're allowed */
    if(!strcmp(remip, hosts[x]))
        return REQ_NOACTION;
}
/* Set response code to forbidden and return an error. */
protocol_status(sn, rq, PROTOCOL_FORBIDDEN, NULL);
return REQ_ABORTED;

ObjectType Example

The example in this section demonstrates how to implement html2shtml, a custom
SAF that instructs the server to treat a .html file as a .shtml file if a .shtml version
of the requested file exists.

A well-behaved ObjectType function checks if the content type is already set, and
if so, does nothing except return REQ_NOACTION.

    if(pblock_findval("content-type", rq->srvhdrs))
        return REQ_NOACTION;

The primary task an ObjectType directive needs to perform is to set the content
type (if it is not already set). This example sets it to
magnus-internal/parsed-html in the following lines:
The `html2shtml` function looks at the requested file name. If it ends with `.html`, the function looks for a file with the same base name, but with the extension `.shtml` instead. If it finds one, it uses that path and informs the server that the file is parsed HTML instead of regular HTML. Note that this requires an extra `stat` call for every HTML file accessed.

### Installing the Example

To load the shared object containing your function, add the following line in the `Init` section of the `magnus.conf` file:

```
Init fn=load-modules shlib=yourlibrary funcs=html2shtml
```

To execute the custom SAF during the request-response process for some object, add the following line to that object in the `obj.conf` file:

```
ObjectType fn=html2shtml
```

### Source Code

The source code for this example is in `otype.c` in the `nsapi/examples/` or `plugins/nsapi/examples` subdirectory within the server root directory.

```c
#include "nsapi.h"
#include <string.h> /* strncpy */
#include "base/util.h"

#ifdef __cplusplus
extern "C"
#endif

NSAPI_PUBLIC int html2shtml(pblock *pb, Session *sn, Request *rq)
{
    /* No parameters */

    /* Work variables */
    pb_param *path = pblock_find("path", rq->srvhdrs);
```
struct stat finfo;
char *npath;
int baselen;

/* If the type has already been set, don't do anything */
if(pblock_findval("content-type", rq->srvhdrs))
    return REQ_NOACTION;

/* If path does not end in .html, let normal object types do
 * their job */
baselen = strlen(path->value) - 5;
if(strcasecmp(&path->value[baselen], ".html") != 0)
    return REQ_NOACTION;

/* 1 = Room to convert html to shtml */
npath = (char *) MALLOC((baselen + 5) + 1 + 1);
strncpy(npath, path->value, baselen);
strcpy(&npath[baselen], ".shtml");

/* If it's not there, don't do anything */
if(stat(npath, &finfo) == -1) {
    FREE(npath);
    return REQ_NOACTION;
}
/* Got it, do the switch */
FREE(path->value);
path->value = npath;

/* The server caches the stat() of the current path. Update it. */
(void) request_stat_path(NULL, rq);
pblock_nvinsert("content-type", "magnus-internal/parsed-html",
    rq->srvhdrs);
return REQ_PROCEED;
Output Example

This section describes an example NSAPI filter named example-replace, which examines outgoing data and substitutes one string for another. It shows how you can create a filter that intercepts and modifies outgoing data.

Installing the Example

To load the filter, add the following line in the Init section of the magnus.conf file:

```
Init fn="load-modules" shlib="<path>/replace.ext"
NativeThread="no"
```

To execute the filter during the request-response process for some object, add the following line to that object in the obj.conf file:

```
Output fn="insert-filter" type="text/*" filter="example-replace"
from="iPlanet" to="Sun ONE"
```

Source Code

The source code for this example is in the replace.c file in the plugins/nsapi/examples subdirectory of the server root directory.

```
#ifdef XP_WIN32
#define NSAPI_PUBLIC __declspec(dllexport)
#else /* !XP_WIN32 */
#define NSAPI_PUBLIC
#endif /* !XP_WIN32 */

/*
 * nsapi.h declares the NSAPI interface.
 */
#include "nsapi.h"

/* -------------------ExampleReplaceData------------------------- */
/* 
* ExampleReplaceData will be used to store information between 
* filter method invocations. Each instance of the example-replace 
* filter will have its own ExampleReplaceData object. 
*/

typedef struct ExampleReplaceData ExampleReplaceData;

struct ExampleReplaceData {
    char *from; /* the string to replace */
    int fromlen; /* length of "from" */
    char *to; /* the string to replace "from" with */
    int tolen; /* length of "to" */
    int matched; /* number of "from" chars matched */
};

/* ------------------ example_replace_insert ------------------------ */

/* 
* example_replace_insert implements the example-replace filter's 
* insert method. The insert filter method is called before the 
* server adds the filter to the filter stack. 
*/

#ifdef __cplusplus
extern "C"
#endif
int example_replace_insert(FilterLayer *layer, pblock *pb) {
    const char *from;
    const char *to;
    ExampleReplaceData *data;

    /* 
    * Look for the string to replace, "from", and the string to 
    * replace it with, "to". Both values are required. 
    */
    from = pblock_findval("from", pb);
    to = pblock_findval("to", pb);
    if (from == NULL || to == NULL || strlen(from) < 1) {
        log_error(LOG_MISCONFIG, "example-replace-insert",
            layer->context->sn, layer->context->rq,
            "missing parameter (need from and to)");
        return REQ_ABORTED; /* error preparing for insertion */
    }
/*
 * Allocate an ExampleReplaceData object that will store
 * configuration and state information.
 */
data = (ExampleReplaceData *)MALLOC(sizeof(ExampleReplaceData));
if (data == NULL)
    return REQ_ABORTED; /* error preparing for insertion */

/* Initialize the ExampleReplaceData */
data->from = STRDUP(from);
data->fromlen = strlen(from);
data->to = STRDUP(to);
data->tolen = strlen(to);
data->matched = 0;

/* Check for out of memory errors */
if (data->from == NULL || data->to == NULL) {
    FREE(data->from);
    FREE(data->to);
    FREE(data);
    return REQ_ABORTED; /* error preparing for insertion */
}

/* Store a pointer to the ExampleReplaceData object in the
 * FilterLayer. This information can then be accessed from other
 * filter methods. */
layer->context->data = data;

/* Remove the Content-length: header if we might change the
 * body length */
if (data->tolen != data->fromlen) {
    pb_param *pp;
    pp = pblock_remove("content-length",
             layer->context->rq->srvhdrs);
    if (pp)
        param_free(pp);
}

return REQ_PROCEED; /* insert filter */
}

/* ----------------- example_replace_remove ----------------- */
/*
* example_replace_remove implements the example-replace filter's
* remove method. The remove filter method is called before the
* server removes the filter from the filter stack.
*/

#ifdef __cplusplus
extern "C"
#endif

void example_replace_remove(FilterLayer *layer)
{
    ExampleReplaceData *data;

    /* Access the ExampleReplaceData we allocated in
       example_replace_insert */
    data = (ExampleReplaceData *)layer->context->data;

    /* Send any partial "from" match */
    if (data->matched > 0)
        net_write(layer->lower, data->from, data->matched);

    /* Destroy the ExampleReplaceData object */
    FREE(data->from);
    FREE(data->to);
    FREE(data);
}

/* -------------- example_replace_write ------------------------- */

/*
* example_replace_write implements the example-replace filter's
* write method. The write filter method is called when there is data
* to be sent to the client.
*/

#ifdef __cplusplus
extern "C"
#endif

int example_replace_write(FilterLayer *layer, const void *buf, int amount)
{
    ExampleReplaceData *data;
    const char *buffer;
    int consumed;
    int i;
    int unsent;
int rv;

/* Access the ExampleReplaceData we allocated in example_replace_insert */
data = (ExampleReplaceData *)layer->context->data;

/* Check for "from" matches in the caller's buffer */
buffer = (const char *)buf;
consumed = 0;
for (i = 0; i < amount; i++) {
    /* Check whether this character matches */
    if (buffer[i] == data->from[data->matched]) {
        /* Matched a(nother) character */
data->matched++;
        /* If we've now matched all of "from"... */
        if (data->matched == data->fromlen) {
            /* Send any data that preceded the match */
            unsent = i + 1 - consumed - data->matched;
            if (unsent > 0) {
                rv = net_write(layer->lower, &buffer[consumed], unsent);
                if (rv != unsent)
                    return IO_ERROR;
            }
            /* Send "to" in place of "from" */
r
            if (rv != data->tolen)
                return IO_ERROR;
        }
        /* Send "to" in place of "from" */
r
        rv = net_write(layer->lower, data->to, data->tolen);
        if (rv != data->tolen)
            return IO_ERROR;
        /* We've handled up to and including buffer[i] */
        consumed = i + 1;
        /* Start looking for the next "from" match from scratch */
data->matched = 0;
    } else if (data->matched > 0) {
        /* This match didn't pan out, we need to backtrack */
        int j;
        int backtrack = data->matched;
data->matched = 0;
        /* Check for other potential "from" matches */
        * preceding buffer[i] */
    } else if (data->matched > 0) {
        /* This match didn't pan out, we need to backtrack */
        int j;
        int backtrack = data->matched;
data->matched = 0;
        /* Check for other potential "from" matches */
        * preceding buffer[i] */
    }
for (j = 1; j < backtrack; j++) {
    /* Check whether this character matches */
    if (data->from[j] == data->from[data->matched]) {
        /* Matched a(nother) character */
        data->matched++;
    } else if (data->matched > 0) {
        /* This match didn't pan out, we need to */
        /* backtrack */
        j -= data->matched;
        data->matched = 0;
    }
}

    /* If the failed (partial) match begins before the */
    /* buffer... */
    unsent = backtrack - data->matched;
    if (unsent > i) {
        /* Send the failed (partial) match */
        rv = net_write(layer->lower, data->from, unsent);
        if (rv != unsent)
            return IO_ERROR;
        /* We've handled up to, but not including, */
        /* buffer[i] */
        consumed = i;
    }

    /* We're not done with buffer[i] yet */
    i--;
}

    /* Send any data we know won't be part of a future */
    /* "from" match */
    unsent = amount - consumed - data->matched;
    if (unsent > 0) {
        rv = net_write(layer->lower, &buffer[consumed], unsent);
        if (rv != unsent)
            return IO_ERROR;
    }

    return amount;
}

    /* ---------------- nsapi_module_init --------------------------- */
/*
* This is the module initialization entry point for this NSAPI
* plugin. The server calls this entry point in response to the
* Init fn="load-modules" line in magnus.conf.
*/

NSAPI_PUBLIC nsapi_module_init(pblock *pb, Session *sn, Request *rq)
{
    FilterMethods methods = FILTER_METHODS_INITIALIZER;
    const Filter *filter;

    /*
    * Create the example-replace filter. The example-replace filter
    * has order FILTER_CONTENT_TRANSLATION, meaning it transforms
    * content (entity body data) from one form to another. The
    * example-replace filter implements the write filter method,
    * meaning it is interested in outgoing data.
    */
    methods.insert = &example_replace_insert;
    methods.remove = &example_replace_remove;
    methods.write = &example_replace_write;
    filter = filter_create("example-replace",
                           FILTER_CONTENT_TRANSLATION,
                           &methods);
    if (filter == NULL) {
        pblock_nvinsert("error", system_errmsg(), pb);
        return REQ_ABORTED; /* error initializing plugin */
    }

    return REQ_PROCEED; /* success */
}

Service Example

This section discusses a very simple Service function called simple_service. All
this function does is send a message in response to a client request. The message is
initialized by the init_simple_service function during server initialization.

For a more complex example, see the file service.c in the examples directory,
which is discussed in “More Complex Service Example” on page 138.
Installing the Example

To load the shared object containing your functions, add the following line in the Init section of the magnus.conf file:

```
Init fn=load-modules shlib=yourlibrary
  funcs=simple-service-init,simple-service
```

To call the simple-service-init function to initialize the message representing the generated output, add the following line to the Init section in magnus.conf. (This line must come after the one that loads the library containing simple-service-init.)

```
Init fn=simple-service-init
  generated-output="<H1>Generated output msg</H1>"
```

To execute the custom SAF during the request-response process for some object, add the following line to that object in the obj.conf file:

Service type="text/html" fn=simple-service

The type="text/html" argument indicates that this function is invoked during the Service stage only if the content-type has been set to text/html.

Source Code

```c
#include <nsapi.h>

static char *simple_msg = "default customized content";

/* This is the initialization function. 
   * It gets the value of the generated-output parameter 
   * specified in the Init directive in magnus.conf 
   */
NSAPI_PUBLIC int init-simple-service(pblock *pb, Session *sn, Request *rq)
{
  /* Get the message from the parameter in the directive in
```
```c
* magnus.conf

/*
  simple_msg = pblock_findval("generated-output", pb);
  return REQ_PROCEED;
}

/* This is the customized Service SAF.
  * It sends the "generated-output" message to the client.
  */
NSAPI_PUBLIC int simple-service(pblock *pb, Session *sn, Request *rq)
{
  int return_value;
  char msg_length[8];

  /* Use the protocol_status function to set the status of the
  * response before calling protocol_start_response.
  */
  protocol_status(sn, rq, PROTOCOL_OK, NULL);
  /* Although we would expect the ObjectType stage to
  * set the content-type, set it here just to be
  * completely sure that it gets set to text/html.
  */
  param_free(pblock_remove("content-type", rq->srvhdrs));
  pblock_nvinsert("content-type", "text/html", rq->srvhdrs);

  /* If you want to use keepalive, need to set content-length header.
   * The util_itoa function converts a specified integer to a
   * string, and returns the length of the string. Use this
   * function to create a textual representation of a number.
   */
  util_itoa(strlen(simple_msg), msg_length);
  pblock_nvinsert("content-length", msg_length, rq->srvhdrs);

  /* Send the headers to the client*/
  return_value = protocol_start_response(sn, rq);
  if (return_value == REQ_NOACTION) {
    /* HTTP HEAD instead of GET */
    return REQ_PROCEED;
  }

  /* Write the output using net_write*/
  return_value = net_write(sn->csd, simple_msg, strlen(simple_msg));
  if (return_value == IO_ERROR) {
    return REQ_EXIT;
  }
```

return REQ_PROCEED;
}

More Complex Service Example

The `send-images` function is a custom SAF that replaces the `doit.cgi` demonstration available on the iPlanet home pages. When a file is accessed as `/dir1/dir2/something.picgroup`, the `send-images` function checks if the file is being accessed by a Mozilla/1.1 browser. If not, it sends a short error message. The file `something.picgroup` contains a list of lines, each of which specifies a file name followed by a `content-type` (for example, `one.gif image/gif`).

To load the shared object containing your function, add the following line at the beginning of the `magnus.conf` file:

```
Init fn=load-modules shlib=yourlibrary funcs=send-images
```

Also, add the following line to the `mime.types` file:

```
type=magnus-internal/picgroup exts=picgroup
```

To execute the custom SAF during the request-response process for some object, add the following line to that object in the `obj.conf` file (`send-images` takes an optional parameter, `delay`, which is not used for this example):

```
Service method=(GET|HEAD) type=magnus-internal/picgroup fn=send-images
```

The source code is in `service.c` in the `nsapi/examples/` or `plugins/nsapi/examples` subdirectory within the server root directory.

AddLog Example

The example in this section demonstrates how to implement `brief-log`, a custom SAF for logging only three items of information about a request: the IP address, the method, and the URI (for example, `198.93.95.99 GET /jocelyn/dogs/homesneeded.html`).
Installing the Example

To load the shared object containing your functions, add the following line in the Init section of the magnus.conf file:

```
Init fn=load-modules shlib=yourlibrary funcs=brief-init,brief-log
```

To call brief-init to open the log file, add the following line to the Init section in magnus.conf. (This line must come after the one that loads the library containing brief-init.)

```
Init fn=brief-init file=/tmp/brief.log
```

To execute your custom SAF during the AddLog stage for some object, add the following line to that object in the obj.conf file:

```
AddLog fn=brief-log
```

Source Code

The source code is in addlog.c is in the nsapi/examples/ or plugins/nsapi/examples subdirectory within the server root directory.

```
#include "nsapi.h"
#include "base/daemon.h" /* daemon_atrestart */
#include "base/file.h"   /* system_fopenWA, system_fclose */
#include "base/util.h"   /* sprintf */

/* File descriptor to be shared between the processes */
static SYS_FILE logfd = SYS_ERROR_FD;

#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC void brief_terminate(void *parameter)
{
    system_fclose(logfd);
    logfd = SYS_ERROR_FD;
}

#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC int brief_init(pblock *pb, Session *sn, Request *rq)
```
AddLog Example

```c
{
    /* Parameter */
    char *fn = pblock_findval("file", pb);

    if(!fn) {
        pblock_nvinsert("error", "brief-init: please supply a file name", pb);
        return REQ_ABORTED;
    }

    logfd = system_fopenWA(fn);
    if(logfd == SYS_ERROR_FD) {
        pblock_nvinsert("error", "brief-init: please supply a file name", pb);
        return REQ_ABORTED;
    }

    /* Close log file when server is restarted */
    daemon_atrestart(brief_terminate, NULL);
    return REQ_PROCEED;
}

#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC int brief_log(pblock *pb, Session *sn, Request *rq)
{
    /* No parameters */

    /* Server data */
    char *method = pblock_findval("method", rq->reqpb);
    char *uri = pblock_findval("uri", rq->reqpb);
    char *ip = pblock_findval("ip", sn->client);

    /* Temp vars */
    char *logmsg;
    int len;

    logmsg = (char *)
        MALLOC(strlen(ip) + 1 + strlen(method) + 1 + strlen(uri) + 1 + 1);
    len = util_sprintf(logmsg, "%s %s %s\n", ip, method, uri);
    /* The atomic version uses locking to prevent interference */
    system_fwrite_atomic(logfd, logmsg, len);
    FREE(logmsg);

    return REQ_PROCEED;
}
```
Quality of Service Example

The code for the qos-handler (AuthTrans) and qos-error (Error) SAFs is provided as an example in case you want to define your own SAFs for quality of service handling.

For more information about predefined SAFs, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

Installing the Example

Inside the default object in obj.conf, add the following AuthTrans and Error directives:

```
AuthTrans fn=qos-handler
...
Error fn=qos-error code=503
```

Source Code

The source code for this example is in the qos.c file in the plugins/nsapi/examples subdirectory of the server root directory.

```
#include "nspr.h"
#include "base/pblock.h"
#include "frame/log.h"
#include "frame/http.h"

/*-----------------------------------
 decode : internal function used for parsing of QOS values in pblock
 -----------------------------------*/

void decode(const char* val, PRInt32* var, pblock* pb)
{
    char* pbval;
    if ( (!var) || (!val) || (!pb) )
        return;
    pbval = pblock_findval(val, pb);
    if (!pbval)
```

```
return;

    *var = atoi(pbval);
}

/*---------------------------------------------
qos_error_sample
This function is meant to be an error handler for an HTTP 503 error
code, which is returned by qos_handler when QOS limits are exceeded
and enforced.
This sample function just prints out a message about which
limits were exceeded.
---------------------------------------------
*/

NSAPI_PUBLIC int qos_error_sample(pblock *pb, Session *sn, Request *rq)
{
    char error[1024] = "";
    char* err_header = "<HTML><HEAD><TITLE>Unable to service
    request</TITLE></HEAD><BODY>";
    char* err_footer = "</BODY></HTML>";

    PRBool ours = PR_FALSE;

    PRInt32 vs_bw = 0, vs_bwlim = 0, vs_bw_ef = 0,
    vs_conn = 0, vs_connlim = 0, vs_conn_ef = 0,
    vsc_bw = 0, vsc_bwlim = 0, vsc_bw_ef = 0,
    vsc_conn = 0, vsc_connlim = 0, vsc_conn_ef = 0,
    srv_bw = 0, srv_bwlim = 0, srv_bw_ef = 0,
    srv_conn = 0, srv_connlim = 0, srv_conn_ef = 0;

    pblock* apb = rq->vars;

decode("vs_bandwidth", &vs_bw, apb);
decode("vs_connections", &vs_conn, apb);

decode("vs_bandwidth_limit", &vs_bwlim, apb);
decode("vs_bandwidth_enforced", &vs_bw_ef, apb);

decode("vs_connections_limit", &vs_connlim, apb);
decode("vs_connections_enforced", &vs_conn_ef, apb);

decode("vsclass_bandwidth", &vsc_bw, apb);
decode("vsclass_connections", &vsc_conn, apb);
decode("vsclass_bandwidth_limit", &vsc_bwlim, apb);
decode("vsclass_bandwidth_enforced", &vsc_bw_ef, apb);

decode("vsclass_connections_limit", &vsc_connlim, apb);
decode("vsclass_connections_enforced", &vsc_conn_ef, apb);

decode("server_bandwidth", &srv_bw, apb);
decode("server_connections", &srv_conn, apb);

decode("server_bandwidth_limit", &srv_bwlim, apb);
decode("server_bandwidth_enforced", &srv_bw_ef, apb);

decode("server_connections_limit", &srv_connlim, apb);
decode("server_connections_enforced", &srv_conn_ef, apb);

if ((vs_bwlim) && (vs_bw>vs_bwlim))
{
    /* VS bandwidth limit was exceeded, display it */
    ours = PR_TRUE;
    sprintf(error, "<P>Virtual server bandwidth limit of %d .
    Current VS bandwidth : %d . <P>",
            vs_bwlim, vs_bw);
}

if ((vs_connlim) && (vs_conn>vs_connlim))
{
    /* VS connection limit was exceeded, display it */
    ours = PR_TRUE;
    sprintf(error, "<P>Virtual server connection limit of %d .
    Current VS connections : %d . <P>",
            vs_connlim, vs_conn);
}

if ((vsc_bwlim) && (vsc_bw>vsc_bwlim))
{
    /* VSCLASS bandwidth limit was exceeded, display it */
    ours = PR_TRUE;
    sprintf(error, "<P>Virtual server class bandwidth limit of
    %d . Current VSCLASS bandwidth : %d . <P>",
            vsc_bwlim, vsc_bw);
}

if ((vsc_connlim) && (vsc_conn>vsc_connlim))
{
    /* VSCLASS connection limit was exceeded, display it */
    ours = PR_TRUE;
    sprintf(error, "<P>Virtual server class connection limit of
Quality of Service Example

```c
%d . Current VSCLASS connections : %d . <P>
   vsc_connlim, vsc_conn);
};

if ((srv_bwlim) && (srv_bw>srv_bwlim))
{
    /* SERVER bandwidth limit was exceeded, display it */
    ours = PR_TRUE;
    sprintf(error, "<P>Global bandwidth limit of %d . Current
   bandwidth : %d . <P>",
       srv_bwlim, srv_bw);
};

if ((srv_connlim) && (srv_conn>srv_connlim))
{
    /* SERVER connection limit was exceeded, display it */
    ours = PR_TRUE;
    sprintf(error, "<P>Global connection limit of %d . Current
   connections : %d . <P>",
       srv_connlim, srv_conn);
};

if (ours)
{
    /* this was really a QOS failure, therefore send the error
       page */
    pb_param *pp = pblock_remove ("content-type", rq->srvhdrs);
    if (pp != NULL)
        param_free (pp);

    pblock_nvinsert ("content-type", "text/html", rq->srvhdrs);

    protocol_start_response(sn, rq);
    net_write(sn->csd, err_header, strlen(err_header));
    net_write(sn->csd, error, strlen(error));
    net_write(sn->csd, err_footer, strlen(err_footer));
    return REQ_PROCEED;
} else
{
    /* this 503 didn't come from a QOS SAF failure, let someone
       else handle it */
    return REQ_PROCEED;
}
```
/*-----------------------------
qos_handler_sample
This is an NSAPI AuthTrans function.
It examines the QOS values in the request and compares them to the
QOS limits.
It does several things:
1) It will log errors if the QOS limits are exceeded.
2) It will return REQ_ABORTED with a 503 error code if the QOS limits
   are exceeded, and the QOS limits are set to be enforced. Otherwise
   it will return REQ_PROCEED.
----------------------------------
NSAPI_PUBLIC int qos_handler_sample(pblock *pb, Session *sn, Request
*rq)
{
    PRBool ok = PR_TRUE;

    PRInt32 vs_bw = 0, vs_bwlim = 0, vs_bw_ef = 0,
    vs_conn = 0, vs_connlim = 0, vs_conn_ef = 0,
    vsc_bw = 0, vsc_bwlim = 0, vsc_bw_ef = 0,
    vsc_conn = 0, vsc_connlim = 0, vsc_conn_ef = 0,
    srv_bw = 0, srv_bwlim = 0, srv_bw_ef = 0,
    srv_conn = 0, srv_connlim = 0, srv_conn_ef = 0;

    pblock* apb = rq->vars;

decode("vs_bandwidth", &vs_bw, apb);
decode("vs_connections", &vs_conn, apb);

    decode("vs_bandwidth_limit", &vs_bwlim, apb);
decode("vs_bandwidth_enforced", &vs_bw_ef, apb);

    decode("vs_connections_limit", &vs_connlim, apb);
decode("vs_connections_enforced", &vs_conn_ef, apb);

    decode("vsclass_bandwidth", &vsc_bw, apb);
decode("vsclass_connections", &vsc_conn, apb);

    decode("vsclass_bandwidth_limit", &vsc_bwlim, apb);
decode("vsclass_bandwidth_enforced", &vsc_bw_ef, apb);

    decode("vsclass_connections_limit", &vsc_connlim, apb);
decode("vsclass_connections_enforced", &vsc_conn_ef, apb);


```c
decode("server_bandwidth", &srv_bw, apb);
decode("server_connections", &srv_conn, apb);
decode("server_bandwidth_limit", &srv_bwlim, apb);
decode("server_bandwidth_enforced", &srv_bw_ef, apb);
decode("server_connections_limit", &srv_connlim, apb);
decode("server_connections_enforced", &srv_conn_ef, apb);

if ((vs_bwlim) && (vs_bw>vs_bwlim))
{
    /* bandwidth limit was exceeded, log it */
    ereport(LOG_FAILURE, "Virtual server bandwidth limit of %d exceeded. Current VS bandwidth : %d", &vs_bwlim, vs_bw);
    if (vs_bw_ef)
    {
        /* and enforce it */
        ok = PR_FALSE;
    };
}

if ((vs_connlim) && (vs_conn>vs_connlim))
{
    /* connection limit was exceeded, log it */
    ereport(LOG_FAILURE, "Virtual server connection limit of %d exceeded. Current VS connections : %d", &vs_connlim, vs_conn);
    if (vs_conn_ef)
    {
        /* and enforce it */
        ok = PR_FALSE;
    };
}

if ((vsc_bwlim) && (vsc_bw>vsc_bwlim))
{
    /* bandwidth limit was exceeded, log it */
    ereport(LOG_FAILURE, "Virtual server class bandwidth limit of %d exceeded. Current VSCLASS bandwidth : %d", &vsc_bwlim, vsc_bw);
    if (vsc_bw_ef)
    {
        /* and enforce it */
        ok = PR_FALSE;
    };
}
```
};

if ((vsc_connlim) && (vsc_conn>vsc_connlim)) {
    /* connection limit was exceeded, log it */
    ereport(LOG_FAILURE, "Virtual server class connection limit of %d exceeded. Current VSCLASS connections : %d", &vsc_connlim, vsc_conn);

    if (vsc_conn_ef)
        { /* and enforce it */
          ok = PR_FALSE;
        }
};

if ((srv_bwlim) && (srv_bw>srv_bwlim)) {
    /* bandwidth limit was exceeded, log it */
    ereport(LOG_FAILURE, "Global bandwidth limit of %d exceeded. Current global bandwidth : %d", &srv_bwlim, srv_bw);

    if (srv_bw_ef)
        { /* and enforce it */
          ok = PR_FALSE;
        }
};

if ((srv_connlim) && (srv_conn>srv_connlim)) {
    /* connection limit was exceeded, log it */
    ereport(LOG_FAILURE, "Global connection limit of %d exceeded. Current global connections : %d", &srv_connlim, srv_conn);

    if (srv_conn_ef)
        { /* and enforce it */
          ok = PR_FALSE;
        }
};

if (ok)
    { return REQ_PROCEED; }
else
{
    /* one of the limits was exceeded
    therefore, we set HTTP error 503 "server too busy" */
    protocol_status(sn, rq, PROTOCOLSERVICEUNAVAILABLE, NULL);
    return REQ_ABORTED;
}
}
Creating Custom Server-parsed HTML Tags

HTML files can contain tags that are executed on the server. For general information about server-parsed HTML tags, see the Sun ONE Web Server 6.1 Programmer’s Guide to Web Applications.

In Sun ONE Web Server 6.1, you can define your own server-side tags. For example, you could define the tag \texttt{HELLO} to invoke a function that prints "Hello World!" You could have the following code in your \texttt{hello.shtml} file:

```xml
<html>
  <head>
    <title>shtml custom tag example</title>
  </head>
  <body>
    <!--#HELLO-->
  </body>
</html>
```

When the browser displays this code, each occurrence of the \texttt{HELLO} tag calls the function.

The steps for defining a customized server-parsed tag are listed below, and described in this chapter:

1. **Define the Functions that Implement the Tag.**
   
   You must define the tag execution function. You must also define other functions that are called on tag loading and unloading, and on page loading and unloading.
Define the Functions that Implement the Tag

2. Write an Initialization Function.

   Write an initialization function that registers the tag using the `shtml_add_tag` function.

3. Load the New Tag into the Server.

Define the Functions that Implement the Tag

Define the functions that implement the tags in C, using NSAPI.

- Include the header `shtml_public.h`, which is in the directory `install_dir/include/shtml`.

- Link against the SHTML shared library. On Windows, `shtml.dll` is in `install_dir/bin`. On UNIX platforms, `libshtml.so` or `s1` is in `install_dir/lib`.

`ShtmlTagExecuteFunc` is the actual tag handler. It gets called with the usual NSAPI `pblock`, `Session`, and `Request` variables. In addition, it also gets passed the `TagUserData` created from the result of executing the tag loading and page loading functions (if defined) for that tag.

The signature for the tag execution function is:

```c
typedef int (*ShtmlTagExecuteFunc)(pblock*, Session*, Request*, TagUserData, TagUserData);
```

Write the body of the tag execution function to generate the output to replace the tag in the `.shtml` page. Do this in the usual NSAPI way, using the `net_write` NSAPI function, which writes a specified number of bytes to a specified socket from a specified buffer.

For more information about writing NSAPI plugins, see Chapter 3, “Creating Custom SAFs.”

For more information about `net_write` and other NSAPI functions, see Chapter 7, “NSAPI Function Reference.”

The tag execution function must return an int that indicates whether the server should proceed to the next instruction in `obj.conf`, which is one of:

- `REQ_PROCEED` -- the execution was successful
- `REQ_NOACTION` -- nothing happened
- `REQ_ABORTED` -- an error occurred
- `REQ_EXIT` -- the connection was lost
The other functions you must define for your tag are:

- **ShtmlTagInstanceLoad**
  
  This is called when a page containing the tag is parsed. It is not called if the page is retrieved from the browser's cache. It basically serves as a constructor, the result of which is cached and is passed into `ShtmlTagExecuteFunc` whenever the execution function is called.

- **ShtmlTagInstanceUnload**
  
  This is basically a destructor for cleaning up whatever was created in the `ShtmlTagInstanceLoad` function. It gets passed the result that was originally returned from the `ShtmlTagInstanceLoad` function.

- **ShtmlTagPageLoadFunc**
  
  This is called when a page containing the tag is executed, regardless of whether the page is still in the browser's cache. This provides a way to make information persistent between occurrences of the same tag on the same page.

- **ShtmlTagPageUnLoadFunc**
  
  This is called after a page containing the tag has executed. It provides a way to clean up any allocations done in a `ShtmlTagPageLoadFunc` and hence gets passed the result returned from the `ShtmlTagPageLoadFunc`.

The signatures for these functions are:

```c
#define TagUserData void*
typedef TagUserData (*ShtmlTagInstanceLoad)(
    const char* tag, pblock*, const char*, size_t);
typedef void (*ShtmlTagInstanceUnload)(TagUserData);
typedef int (*ShtmlTagExecuteFunc)(
    pblock*, Session*, Request*, TagUserData, TagUserData);
typedef TagUserData (*ShtmlTagPageLoadFunc)(
    pblock* pb, Session*, Request*);
typedef void (*ShtmlTagPageUnLoadFunc)(TagUserData);
```

Here is the code that implements the **HELLO** tag:

```c
/* mytag.c: NSAPI functions to implement #HELLO SSI calls */
#include "nsapi.h"
```
Define the Functions that Implement the Tag

```c
#include "shtml/shtml_public.h"

/* FUNCTION : mytag_con
 * DESCRIPTION: ShtmlTagInstanceLoad function */
#ifdef __cplusplus
extern "C"
#endif
TagUserData
mytag_con(const char* tag, pblock* pb, const char* c1, size_t t1)
{
    return NULL;
}

/* FUNCTION : mytag_des
 * DESCRIPTION: ShtmlTagInstanceUnload */
#ifdef __cplusplus
extern "C"
#endif
void
mytag_des(TagUserData v1)
{
}

/* FUNCTION : mytag_load
 * DESCRIPTION: ShtmlTagPageLoadFunc */
#ifdef __cplusplus
extern "C"
#endif
TagUserData
mytag_load(pblock *pb, Session *sn, Request *rq)
{
    return NULL;
}

/* FUNCTION : mytag_unload
 * DESCRIPTION: ShtmlTagPageUnloadFunc */
#ifdef __cplusplus
extern "C"
#endif
void
mytag_unload(TagUserData v2)
{
}
```
/* FUNCTION : mytag
   * DESCRIPTION: ShtmlTagExecuteFunc
   */
#endif
#endif
#endif
int
mytag(pblock* pb, Session* sn, Request* rq, TagUserData t1,
      TagUserData t2)
{
    char* buf;
    int length;
    char* client;
    buf = (char *) MALLOC(100*sizeof(char));
    length = util_sprintf(buf, "<h1>Hello World! </h1>", client);
    if (net_write(sn->csd, buf, length) == IO_ERROR)
    {
      FREE(buf);
      return REQ_ABORTED;
    }
    FREE(buf);
    return REQ_PROCEED;
} /* FUNCTION : mytag_init
   * DESCRIPTION: initialization function, calls shtml_add_tag() to
   * load new tag
   */
#endif
int
mytag_init(pblock* pb, Session* sn, Request* rq)
{
    int retVal = 0;
    // NOTE: ALL arguments are required in the shtml_add_tag() function
    retVal = shtml_add_tag("HELLO", mytag_con, mytag_des, mytag,
                           mytag_load, mytag_unload);
    return retVal;
} */ end mytag.c */
Write an Initialization Function

In the initialization function for the shared library that defines the new tag, register
the tag using the function shtml_add_tag. The signature is:

```c
NSAPI_PUBLIC int shtml_add_tag (  
    const char* tag,  
    ShtmlTagInstanceLoad ctor,  
    ShtmlTagInstanceUnload dtor,  
    ShtmlTagExecuteFunc execFn,  
    ShtmlTagPageLoadFunc pageLoadFn,  
    ShtmlTagPageUnLoadFunc pageUnLoadFn);  
```

Any of these arguments can return NULL except for the tag and execFn.

Load the New Tag into the Server

After creating the shared library that defines the new tag, you load the library into
the Sun ONE Web Server in the usual way for NSAPI plugins. That is, add the
following directives to the configuration file magnus.conf:

Add an Init directive whose fn parameter is load-modules and whose shlib
parameter is the shared library to load. For example, if you compiled your tag into
the shared object install_dir/hello.so, it would be:

```
Init funcs="mytag,mytag_init" shlib="install_dir/hello.so"  
fn="load-modules"
```

Add another Init directive whose fn parameter is the initialization function in the
shared library that uses shtml_add_tag to register the tag. For example:

```
Init fn="mytag_init"
```
This chapter lists all of the public C functions and macros of the Netscape Server Applications Programming Interface (NSAPI) in alphabetic order. These are the functions you use when writing your own Server Application Functions (SAFs).

See Chapter 2, “SAFs in the magnus.conf File” for a list of the predefined Init SAFs. For more information about the other predefined SAFs used in obj.conf, see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.

Each function provides the name, syntax, parameters, return value, a description of what the function does, and sometimes an example of its use and a list of related functions.

For more information on data structures, see Chapter 8, “Data Structure Reference,” and also look in the nsapi.h header file in the include directory in the build for Sun ONE Web Server 6.1.

NSAPI Functions (in Alphabetical Order)

For an alphabetical list of function names, see Appendix A, “Alphabetical List of NSAPI Functions and Macros.”
CALLOC

The **CALLOC** macro is a platform-independent substitute for the C library routine **calloc**. It allocates num*size bytes from the request's memory pool. If pooled memory has been disabled in the configuration file (with the pool-init built-in SAF), **PERM_CALLOC** and **CALLOC** both obtain their memory from the system heap.

**Syntax**

```c
void *CALLOC(int size)
```

**Returns**

A void pointer to a block of memory.

**Parameters**

`int size` is the size in bytes of each element.

**Example**

```c
char *name;
name = (char *) CALLOC(100);
```

**See Also**

FREE, REALLOC, STRDUP, PERM_MALLOC, PERM_FREE, PERM_REALLOC, PERM_STRDUP

cinfo_find

The **cinfo_find()** function uses the MIME types information to find the type, encoding, and/or language based on the extension(s) of the Universal Resource Identifier (URI) or local file name. Use this information to send headers `(rq->srvhdrs)` to the client indicating the `content-type`, `content-encoding`, and `content-language` of the data it will be receiving from the server.

The name used is everything after the last slash (/) or the whole string if no slash is found. File name extensions are not case-sensitive. The name may contain multiple extensions separated by period (.) to indicate type, encoding, or language. For example, the URI `a/b/filename.jp.txt.zip` could represent a Japanese language, text/plain type, zip encoded file.
Syntax

cinfo *cinfo_find(char *uri);

Returns
A pointer to a newly allocated cinfo structure if content info was found, or NULL if no content was found.

The cinfo structure that is allocated and returned contains pointers to the content-type, content-encoding, and content-language, if found. Each is a pointer into static data in the types database, or NULL if not found. Do not free these pointers. You should free the cinfo structure when you are done using it.

Parameters
char *uri is a Universal Resource Identifier (URI) or local file name. Multiple file name extensions should be separated by periods (.)

condvar_init

The condvar_init function is a critical-section function that initializes and returns a new condition variable associated with a specified critical-section variable. You can use the condition variable to prevent interference between two threads of execution.

Syntax

CONDVAR condvar_init(CRITICAL id);

Returns
A newly allocated condition variable (CONDVAR).

Parameters

CRITICAL id is a critical-section variable.

See Also
condvar_notify, condvar_terminate, condvar_wait, crit_init, crit_enter, crit_exit, crit_terminate
**condvar_notify**

The `condvar_notify` function is a critical-section function that awakens any threads that are blocked on the given critical-section variable. Use this function to awaken threads of execution of a given critical section. First, use `crit_enter` to gain ownership of the critical section. Then use the returned critical-section variable to call `condvar_notify` to awaken the threads. Finally, when `condvar_notify` returns, call `crit_exit` to surrender ownership of the critical section.

**Syntax**

```c
void condvar_notify(CONDVAR cv);
```

**Returns**

`void`

**Parameters**

`CONDVAR cv` is a condition variable.

**See Also**

`condvar_init`, `condvar_terminate`, `condvar_wait`, `crit_init`, `crit_enter`, `crit_exit`, `crit_terminate`

**condvar_terminate**

The `condvar_terminate` function is a critical-section function that frees a condition variable. Use this function to free a previously allocated condition variable.

**Warning**

Terminating a condition variable that is in use can lead to unpredictable results.

**Syntax**

```c
void condvar_terminate(CONDVAR cv);
```

**Returns**

`void`

**Parameters**

`CONDVAR cv` is a condition variable.
condvar_wait

The `condvar_wait` function is a critical-section function that blocks on a given condition variable. Use this function to wait for a critical section (specified by a condition variable argument) to become available. The calling thread is blocked until another thread calls `condvar_notify` with the same condition variable argument. The caller must have entered the critical section associated with this condition variable before calling `condvar_wait`.

**Syntax**

```c
void condvar_wait(CONDVAR cv);
```

**Returns**

`void`

**Parameters**

- `CONDVAR cv` is a condition variable.

**See Also**

- `condvar_init`, `condvar_terminate`, `condvar_notify`, `crit_init`, `crit_enter`, `crit_exit`, `crit_terminate`

crit_enter

The `crit_enter` function is a critical-section function that attempts to enter a critical section. Use this function to gain ownership of a critical section. If another thread already owns the section, the calling thread is blocked until the first thread surrenders ownership by calling `crit_exit`.

**Syntax**

```c
void crit_enter(CRITICAL crvar);
```

**Returns**

`void`

**Parameters**

- `CRITICAL crvar` is a critical-section variable.
crit_exit

The **crit_exit** function is a critical-section function that surrenders ownership of a critical section. Use this function to surrender ownership of a critical section. If another thread is blocked waiting for the section, the block will be removed and the waiting thread will be given ownership of the section.

**Syntax**

```c
void crit_exit(CRITICAL crvar);
```

**Returns**

`void`

**Parameters**

`CRITICAL crvar` is a critical-section variable.

**See Also**

crit_init, crit_enter, crit_terminate

---

crit_init

The **crit_init** function is a critical-section function that creates and returns a new critical-section variable (a variable of type `CRITICAL`). Use this function to obtain a new instance of a variable of type `CRITICAL` (a critical-section variable) to be used in managing the prevention of interference between two threads of execution. At the time of its creation, no thread owns the critical section.

**Warning**

Threads must not own or be waiting for the critical section when `crit_terminate` is called.

**Syntax**

```c
CRITICAL crit_init(void);
```

**Returns**

A newly allocated critical-section variable (`CRITICAL`).
Parameters
none

See Also
crit_enter, crit_exit, crit_terminate

crit_terminate
The crit_terminate function is a critical-section function that removes a previously allocated critical-section variable (a variable of type CRITICAL). Use this function to release a critical-section variable previously obtained by a call to crit_init.

Syntax
void crit_terminate(CRITICAL crvar);

Returns
void

Parameters
CRITICAL crvar is a critical-section variable.

See Also
crit_init, crit_enter, crit_exit

daemon_atrestart
The daemon_atrestart function lets you register a callback function named by fn to be used when the server terminates. Use this function when you need a callback function to deallocate resources allocated by an initialization function. The daemon_atrestart function is a generalization of the magnus_atrestart function.

The magnus.conf directives TerminateTimeout and ChildRestartCallback also affect the callback of NSAPI functions.

Syntax
void daemon_atrestart(void (*fn)(void *), void *data);
Returns
void

Parameters
void (* fn) (void *) is the callback function.

void *data is the parameter passed to the callback function when the server is restarted.

Example
/* Register the log_close function, passing it NULL */
 /* to close a log file when the server is */
 /* restarted or shutdown. */
daemon_atrestart(log_close, NULL);
NSAPI_PUBLIC void log_close(void *parameter)
{
system_fclose(global_logfd);
}

F

fc_open
The fc_open function returns a pointer to PRFileDesc that refers to an open file (fileName). The fileName must be the full path name of an existing file. The file is opened in read mode only. The application calling this function should not modify the currency of the file pointed to by the PRFileDesc * unless the DUP_FILE_DESC is also passed to this function. In other words, the application (at minimum) should not issue a read operation based on this pointer that would modify the currency for the PRFileDesc *. If such a read operation is required (that may change the currency for the PRFileDesc *), then the application should call this function with the argument DUP_FILE_DESC.

On a successful call to this function, a valid pointer to PRFileDesc is returned and the handle 'FcHdl' is properly initialized. The size information for the file is stored in the 'fileSize' member of the handle.

Syntax
PRFileDesc *fc_open(const char *fileName, FcHdl *hDl, PRUint32 flags,
Session *sn, Request *rq);
NSAPI Functions (in Alphabetical Order)

Returns
Point to PRFileDesc, or NULL on failure.

Parameters
const char *fileName is the full path name of the file to be opened.
Fchdl*hDl is a valid pointer to a structure of type Fchdl.
PRUint32 flags can be 0 or DUP_FILE_DESC.
Session *sn is a pointer to the session.
Request *rq is a pointer to the request.

fc_close
The fc_close function closes a file opened using fc_open. This function should only be called with files opened using fc_open.

Syntax
void fc_close(PRFileDesc *fd, Fchdl *hDl);

Returns
void

Parameters
PRFileDesc *fd is a valid pointer returned from a prior call to fc_open.
Fchdl *hDl is a valid pointer to a structure of type Fchdl. This pointer must have been initialized by a prior call to fc_open.

filebuf_buf2sd
The filebuf_buf2sd function sends a file buffer to a socket (descriptor) and returns the number of bytes sent.
Use this function to send the contents of an entire file to the client.

Syntax
int filebuf_buf2sd(filebuf *buf, SYS_NETFD sd);
NSAPI Functions (in Alphabetical Order)

Returns
The number of bytes sent to the socket if successful, or the constant IO_ERROR if the file buffer could not be sent.

Parameters
filebuf *buf is the file buffer that must already have been opened.

SYS_NETFD sd is the platform-independent socket descriptor. Normally this will be obtained from the csd (client socket descriptor) field of the sn (session) structure.

Example
if (filebuf_buf2sd(buf, sn->csd) == IO_ERROR)
    return(REQ_EXIT);

See Also
filebuf_close, filebuf_open, filebuf_open_nostat, filebuf_getc

filebuf_close
The filebuf_close function deallocates a file buffer and closes its associated file.

Generally, use filebuf_open first to open a file buffer, and then filebuf_getc to access the information in the file. After you have finished using the file buffer, use filebuf_close to close it.

Syntax
void filebuf_close(filebuf *buf);

Returns
void

Parameters
filebuf *buf is the file buffer previously opened with filebuf_open.

Example
filebuf_close(buf);

See Also
filebuf_open, filebuf_open_nostat, filebuf_buf2sd, filebuf_getc
filebuf_getc

The `filebuf_getc` function retrieves a character from the current file position and returns it as an integer. It then increments the current file position.

Use `filebuf_getc` to sequentially read characters from a buffered file.

**Syntax**

```c
filebuf_getc(filebuf b);
```

**Returns**

An integer containing the character retrieved, or the constant `IO_EOF` or `IO_ERROR` upon an end of file or error.

**Parameters**

`filebuf b` is the name of the file buffer.

**See Also**

`filebuf_close`, `filebuf_buf2sd`, `filebuf_open`, `filter_create`

filebuf_open

The `filebuf_open` function opens a new file buffer for a previously opened file. It returns a new buffer structure. Buffered files provide more efficient file access by guaranteeing the use of buffered file I/O in environments where it is not supported by the operating system.

**Syntax**

```c
filebuf *filebuf_open(SYS_FILE fd, int sz);
```

**Returns**

A pointer to a new buffer structure to hold the data if successful, or NULL if no buffer could be opened.

**Parameters**

`SYS_FILE fd` is the platform-independent file descriptor of the file which has already been opened.

`int sz` is the size, in bytes, to be used for the buffer.
Example

```c
filebuf *buf = filebuf_open(fd, FILE_BUFFERSIZE);
if (!buf) {
    system_fclose(fd);
}
```

See Also

`filebuf_getc`, `filebuf_buf2sd`, `filebuf_close`, `filebuf_open_nostat`

---

**filebuf_open_nostat**

The `filebuf_open_nostat` function opens a new file buffer for a previously opened file. It returns a new buffer structure. Buffered files provide more efficient file access by guaranteeing the use of buffered file I/O in environments where it is not supported by the operating system.

This function is the same `filebuf_open`, but is more efficient, since it does not need to call the `request_stat_path` function. It requires that the stat information be passed in.

**Syntax**

```c
filebuf* filebuf_open_nostat(SYS_FILE fd, int sz,
                             struct stat *finfo);
```

**Returns**

A pointer to a new buffer structure to hold the data if successful, or NULL if no buffer could be opened.

**Parameters**

- `SYS_FILE fd` is the platform-independent file descriptor of the file that has already been opened.
- `int sz` is the size, in bytes, to be used for the buffer.
- `struct stat *finfo` is the file information of the file. Before calling the `filebuf_open_nostat` function, you must call the `request_stat_path` function to retrieve the file information.

**Example**

```c
filebuf *buf = filebuf_open_nostat(fd, FILE_BUFFERSIZE, &finfo);
if (!buf) {
    system_fclose(fd);
}
```
See Also
filebuf_close, filebuf_open, filebuf_getc, filebuf_buf2sd

**filter_create**

The `filter_create` function defines a new filter.

The name parameter specifies a unique name for the filter. If a filter with the specified name already exists, it will be replaced.

Names beginning with `magnus-` or `server-` are reserved by the server.

The order parameter indicates the position of the filter in the filter stack by specifying what class of functionality the filter implements.

The following table describes parameters allowed order constants and their associated meanings for the `filter_create` function. The left column lists the name of the constant, the middle column describes the functionality the filter implements, and the right column lists the position the filter occupies in the filter stack.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Functionality Filter Implements</th>
<th>Position in Filter Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILTER_CONTENT_TRANSLATION</td>
<td>Translates content from one form to another (for example, XSLT)</td>
<td>Top</td>
</tr>
<tr>
<td>FILTER_CONTENT_CODING</td>
<td>Encodes content (for example, HTTP gzip compression)</td>
<td>Middle</td>
</tr>
<tr>
<td>FILTER_TRANSFER_CODING</td>
<td>Encodes entity bodies for transmission (for example, HTTP chunking)</td>
<td>Bottom</td>
</tr>
</tbody>
</table>

The methods parameter specifies a pointer to a `FilterMethods` structure. Before calling `filter_create`, you must first initialize the `FilterMethods` structure using the `FILTER_METHODS_INITIALIZER` macro, and then assign function pointers to the individual `FilterMethods` members (for example, `insert`, `read`, `write`, and so on) that correspond to the filter methods the filter will support.

`filter_create` returns `const Filter *`, a pointer to an opaque representation of the filter. This value may be passed to `filter_insert` to insert the filter in a particular filter stack.
**Syntax**

```c
const Filter *filter_create(const char *name, int order, const FilterMethods *methods);
```

**Returns**
The `const Filter *` that identifies the filter or NULL if an error occurred.

**Parameters**
- `const char *name` is the name of the filter.
- `int order` is one of the order constants above.
- `const FilterMethods *methods` contains pointers to the filter methods the filter supports.

**Example**

```c
FilterMethods methods = FILTER_METHODS_INITIALIZER;
const Filter *filter;
/* This filter will only support the "read" filter method */
methods.read = my_input_filter_read;
/* Create the filter */
filter = filter_create("my-input-filter",
FILTER_CONTENT_TRANSITION,
&methods);
```

**filter_find**

The `filter_find` function finds the filter with the specified name.

**Syntax**

```c
const Filter *filter_find(const char *name);
```

**Returns**
The `const Filter *` that identifies the filter, or NULL if the specified filter does not exist.

**Parameters**
- `const char *name` is the name of the filter of interest.
filter_insert

The `filter_insert` function inserts a filter into a filter stack, creating a new filter layer and installing the filter at that layer. The filter layer's position in the stack is determined by the order value specified when `filter_create` was called, and any explicit ordering configured by `init-filter-order`. If a filter layer with the same order value already exists in the stack, the new layer is inserted above that layer.

Parameters may be passed to the filter using the `pb` and `data` parameters. The semantics of the data parameter are defined by individual filters. However, all filters must be able to handle a data parameter of NULL.

When possible, plugin developers should avoid calling `filter_insert` directly, and instead use the `insert-filter` SAF (applicable in Input-class directives).

**Syntax**

```c
int filter_insert(SYS_NETFD sd, pblock *pb, Session *sn, Request *rq, void *data, const Filter *filter);
```

**Returns**

Returns `REQ_PROCEED` if the specified filter was inserted successfully, or `REQ_NOACTION` if the specified filter was not inserted because it was not required. Any other return value indicates an error.

**Parameters**

- `SYS_NETFD sd` is NULL (reserved for future use).
- `pblock *pb` is a set of parameters to pass to the specified filter's init method.
- `Session *sn` is the Session.
- `Request *rq` is the Request.
- `void *data` is filter-defined private data.
- `const Filter *filter` is the filter to insert.

filter_layer

The `filter_layer` function returns the layer in a filter stack that corresponds to the specified filter.

**Syntax**

```c
FilterLayer *filter_layer(SYS_NETFD sd, const Filter *filter);
```
Returns
The topmost FilterLayer * associated with the specified filter, or NULL if the specified filter is not part of the specified filter stack.

Parameters
SYS_NETFD sd is the filter stack to inspect.
const Filter *filter is the filter of interest.

filter_name
The filter_name function returns the name of the specified filter. The caller should not free the returned string.

Syntax
const char *filter_name(const Filter *filter);

Returns
The name of the specified filter, or NULL if an error occurred.

Parameters
const Filter *filter is the filter of interest.

filter_remove
The filter_remove function removes the specified filter from the specified filter stack, destroying a filter layer. If the specified filter was inserted into the filter stack multiple times, only that filter’s topmost filter layer is destroyed.

When possible, plugin developers should avoid calling filter_remove directly, and instead use the remove-filter SAF (applicable in Input-, Output-, Service-, and Error-class directives).

Syntax
int filter_remove(SYS_NETFD sd, const Filter *filter);

Returns
Returns REQ_PROCEED if the specified filter was removed successfully or REQ_NOACTION if the specified filter was not part of the filter stack. Any other return value indicates an error.
Parameters
SYS_NETFD sd is the filter stack, sn->csd.

const Filter *filter is the filter to remove.

flush

The flush filter method is called when buffered data should be sent. Filters that buffer outgoing data should implement the flush filter method.

Upon receiving control, a flush implementation must write any buffered data to the filter layer immediately below it. Before returning success, a flush implementation must successfully call the net_flush function:

net_flush(layer->lower).

Syntax
int flush(FilterLayer *layer);

Returns
0 on success or -1 if an error occurred.

Parameters
FilterLayer *layer is the filter layer the filter is installed in.

Example
int myfilter_flush(FilterLayer *layer)
{
    MyFilterContext context = (MyFilterContext *)layer->context->data;
    if (context->buf.count) {
        int rv;
        rv = net_write(layer->lower, context->buf.data,
                        context->buf.count);
        if (rv != context->buf.count)
            return -1; /* failed to flush data */
        context->buf.count = 0;
    }
    return net_flush(layer->lower);
}

See Also
net_flush
FREE

The FREE macro is a platform-independent substitute for the C library routine free. It deallocates the space previously allocated by MALLOC, CALLOC, or STRDUP from the request’s memory pool.

Syntax
FREE(void *ptr);

Returns
void

Parameters
void *ptr is a (void *) pointer to a block of memory. If the pointer is not one created by MALLOC, CALLOC, or STRDUP, the behavior is undefined.

Example
char *name;
name = (char *) MALLOC(256);
...
FREE(name);

See Also
CALLOC, REALLOC, STRDUP, PERM_MALLOC, PERM_FREE, PERM_REALLOC, PERM_STRDUP

func_exec

The func_exec function executes the function named by the fn entry in a specified pblock. If the function name is not found, it logs the error and returns REQ_ABORTED.

You can use this function to execute a built-in Server Application Function (SAF) by identifying it in the pblock.

Syntax
int func_exec(pblock *pb, Session *sn, Request *rq);

Returns
The value returned by the executed function, or the constant REQ_ABORTED if no function was executed.
**Parameters**

- `pblock pb` is the pblock containing the function name (`fn`) and parameters.
- `Session *sn` is the Session.
- `Request *rq` is the Request.

The `Session` and `Request` parameters are the same as the ones passed into your SAF.

**See Also**

- `log_error`

---

**func_find**

The `func_find` function returns a pointer to the function specified by `name`. If the function does not exist, it returns NULL.

**Syntax**

```c
FuncPtr func_find(char *name);
```

**Returns**

A pointer to the chosen function, suitable for dereferencing, or NULL if the function could not be found.

**Parameters**

- `char *name` is the name of the function.

**Example**

```c
/* this block of code does the same thing as func_exec */
char *afunc = pblock_findval("afunction", pb);
FuncPtr afnptr = func_find(afunc);
if (afnptr)
    return (afnptr)(pb, sn, rq);
```

**See Also**

- `func_exec`

---

**func_insert**

The `func_insert` function dynamically inserts a named function into the server's table of functions. This function should only be called during the `Init` stage.
**Syntax**

```c
FuncStruct *func_insert(char *name, FuncPtr fn);
```

**Returns**

Returns the `FuncStruct` structure that identifies the newly inserted function. The caller should not modify the contents of the `FuncStruct` structure.

**Parameters**

- `char *name` is the name of the function.
- `FuncPtr fn` is the pointer to the function.

**Example**

```c
func_insert("my-service-saf", &my_service_saf);
```

**See Also**

`func_exec`, `func_find`

---

**insert**

The `insert` filter method is called when a filter is inserted into a filter stack by the `filter_insert` function or `insert-filter` SAF (applicable in Input-class directives).

**Syntax**

```c
int insert(FilterLayer *layer, pblock *pb);
```

**Returns**

Returns `REQ_PROCEED` if the filter should be inserted into the filter stack, `REQ_NOACTION` if the filter should not be inserted because it is not required, or `REQ_ABORTED` if the filter should not be inserted because of an error.

**Parameters**

- `FilterLayer *layer` is the filter layer at which the filter is being inserted.
- `pblock *pb` is the set of parameters passed to `filter_insert` or specified by the `fn="insert-filter"` directive.

**Example**

```c
FilterMethods myfilter_methods = FILTER_METHODS_INITIALIZER;
const Filter *myfilter;
int myfilter_insert(FilterLayer *layer, pblock *pb)
{
    if (pblock_findval("dont-insert-filter", pb))
```
return REQ_NOACTION;
return REQ_PROCEED;
}
...
myfilter_methods.insert = &myfilter_insert;
    myfilter = filter_create("myfilter", &myfilter_methods);
...

log_error

The log_error function creates an entry in an error log, recording the date, the severity, and a specified text.

**Syntax**

```c
int log_error(int degree, char *func, Session *sn, Request *rq, char *fmt, ...);
```

**Returns**

0 if the log entry was created, or -1 if the log entry was not created.

**Parameters**

- `int degree` specifies the severity of the error. It must be one of the following constants:
  - `LOG_WARN` -- warning
  - `LOG_MISCONFIG` -- a syntax error or permission violation
  - `LOG_SECURITY` -- an authentication failure or 403 error from a host
  - `LOG_FAILURE` -- an internal problem
  - `LOG_CATASTROPHE` -- a nonrecoverable server error
  - `LOG_INFORM` -- an informational message
- `char *func` is the name of the function where the error has occurred.
- `Session *sn` is the Session.
- `Request *rq` is the Request.
- `char *fmt` specifies the format for the printf function that delivers the message.
- The Session and Request parameters are the same as the ones passed into your SAF.
... represents a sequence of parameters for the printf function.

Example

```c
log_error(LOG_WARN, "send-file", sn, rq,
   "error opening buffer from %s (%s)"), path,
   system_errmsg(fd));
```

See Also

func_exec

M

MALLOC

The MALLOC macro is a platform-independent substitute for the C library routine malloc. It normally allocates from the request's memory pool. If pooled memory has been disabled in the configuration file (with the pool-init built-in SAF), PERM_MALLOC and MALLOC both obtain their memory from the system heap.

Syntax

```c
void *MALLOC(int size)
```

Returns

A void pointer to a block of memory.

Parameters

int size is the number of bytes to allocate.

Example

```c
/* Allocate 256 bytes for a name */
char *name;
name = (char *) MALLOC(256);
```

See Also

FREE, CALLOC, REALLOC, STRDUP, PERM_MALLOC, PERM_FREE, PERM_CALLOC, PERM_REALLOC, PERM_STRDUP
**net_flush**

The `net_flush` function flushes any buffered data. If you require that data be sent immediately, call `net_flush` after calling network output functions such as `net_write` or `net_sendfile`.

**Syntax**

```c
int net_flush(SYS_NETFD sd);
```

**Returns**

0 on success, or a negative value if an error occurred.

**Parameters**

`SYS_NETFD sd` is the socket to flush.

**Example**

```c
nen_write(sn->csd, "Please wait... ", 15);
net_flush(sn->csd);
/* Perform some time-intensive operation */
...
net_write(sn->csd, "Thank you.\n", 11);
```

**See Also**

`net_write`, `net_sendfile`

---

**net_ip2host**

The `net_ip2host` function transforms a textual IP address into a fully-qualified domain name and returns it.

---

**NOTE**

This function works only if the DNS directive is enabled in the `magnus.conf` file. For more information, see Chapter 2, “SAFs in the magnus.conf File.”

**Syntax**

```c
char *net_ip2host(char *ip, int verify);
```
**Returns**
A new string containing the fully-qualified domain name if the transformation was accomplished, or NULL if the transformation was not accomplished.

**Parameters**
char *ip is the IP address as a character string in dotted-decimal notation: nnn.nnn.nnn.nnn

int verify, if nonzero, specifies that the function should verify the fully-qualified domain name. Though this requires an extra query, you should use it when checking access control.

**net_read**
The net_read function reads bytes from a specified socket into a specified buffer. The function waits to receive data from the socket until either at least one byte is available in the socket or the specified time has elapsed.

**Syntax**
int net_read (SYS_NETFD sd, char *buf, int sz, int timeout);

**Returns**
The number of bytes read, which will not exceed the maximum size, sz. A negative value is returned if an error has occurred, in which case errno is set to the constant ETIMEDOUT if the operation did not complete before timeout seconds elapsed.

**Parameters**
SYS_NETFD sd is the platform-independent socket descriptor.

char *buf is the buffer to receive the bytes.

int sz is the maximum number of bytes to read.

int timeout is the number of seconds to allow for the read operation before returning. The purpose of timeout is not to return because not enough bytes were read in the given time, but to limit the amount of time devoted to waiting until some data arrives.

**See Also**
net_write
net_sendfile

The net_sendfile function sends the contents of a specified file to a specified a socket. Either the whole file or a fraction may be sent, and the contents of the file may optionally be preceded and/or followed by caller-specified data.

Parameters are passed to net_sendfile in the sendfiledata structure. Before invoking net_sendfile, the caller must initialize every sendfiledata structure member.

Syntax

int net_sendfile(SYS_NETFD sd, const sendfiledata *sfd);

Returns

A positive number indicates the number of bytes successfully written, including the headers, file contents, and trailers. A negative value indicates an error.

Parameters

SYS_NETFD sd is the socket to write to.

const sendfiledata *sfd identifies the data to send.

Example

The following Service SAF sends a file bracketed by the strings "begin" and "end."

```c
#include <string.h>
#include "nsapi.h"

NSAPI_PUBLIC int service_net_sendfile(pblock *pb, Session *sn, Request *rq)
{
    char *path;
    SYS_FILE fd;
    struct sendfiledata sfd;
    int rv;

    path = pblock_findval("path", rq->vars);
    fd = system_fopenRO(path);
    if (!fd) {
        log_error(LOG_MISCONFIG, "service-net-sendfile", sn, rq,
            "Error opening %s (%s)", path, system_errmsg());
        return REQ_ABORTED;
    }

    sfd.fd = fd; /* file to send */
    sfd.offset = 0; /* start sending from the
beginning */
  sfd.len = 0;         /* send the whole file */
  sfd.header = "begin";
  /* header data to send before the file */
  sfd.hlen = strlen(sfd.header);       /* length of header data */
  /* send the headers, file, and trailers to the client */
  rv = net_sendfile(sn->csd, &sfd);

  system_fclose(fd);
  if (rv < 0) {
    log_error(LOG_INFORM, "service-net-sendfile", sn, rq, "Error sending \%s (\%s)", path, system_errmsg());
    return REQ_ABORTED;
  }

  return REQ_PROCEED;
}

See Also
net_flush

net_write

The net_write function writes a specified number of bytes to a specified socket from a specified buffer.

Syntax
int net_write(SYS_NETFD sd, char *buf, int sz);

Returns
The number of bytes written, which may be less than the requested size if an error occurred.

Parameters
SYS_NETFD sd is the platform-independent socket descriptor.
char *buf is the buffer containing the bytes.
int  sz  is the number of bytes to write.

Example
if (net_write(sn->csd, FIRSTMSG, strlen(FIRSTMSG)) == IO_ERROR)
   return REQ_EXIT;

See Also
net_read

netbuf_buf2sd
The netbuf_buf2sd function sends a buffer to a socket. You can use this function
to send data from IPC pipes to the client.

Syntax
int netbuf_buf2sd(netbuf *buf, SYS_NETFD sd, int len);

Returns
The number of bytes transferred to the socket, if successful, or the constant
IO_ERROR if unsuccessful.

Parameters
netbuf *buf  is the buffer to send.
SYS_NETFD sd is the platform-independent identifier of the socket.
int  len  is the length of the buffer.

See Also
netbuf_close, netbuf_getc, netbuf_grab, netbuf_open

netbuf_close
The netbuf_close function deallocates a network buffer and closes its associated
files. Use this function when you need to deallocate the network buffer and close
the socket.
You should never close the netbuf parameter in a session structure.

Syntax
void netbuf_close(netbuf *buf);
netbuf_getc

The `netbuf_getc` function retrieves a character from the cursor position of the network buffer specified by `b`.

**Syntax**

```c
netbuf_getc(netbuf b);
```

**Returns**

The integer representing the character if one was retrieved, or the constant `IO_EOF` or `IO_ERROR` for end of file or error.

**Parameters**

`netbuf b` is the buffer from which to retrieve one character.

**See Also**

`netbuf_buf2sd`, `netbuf_getc`, `netbuf_grab`, `netbuf_open`

netbuf_grab

The `netbuf_grab` function reads `sz` number of bytes from the network buffer’s (buf) socket into the network buffer. If the buffer is not large enough it is resized. The data can be retrieved from `buf->inbuf` on success.

This function is used by the function `netbuf_buf2sd`.

**Syntax**

```c
int netbuf_grab(netbuf *buf, int sz);
```

**Returns**

The number of bytes actually read (between 1 and `sz`) if the operation was successful, or the constant `IO_EOF` or `IO_ERROR` for end of file or error.
Parameters

netbuf *buf is the buffer to read into.

int sz is the number of bytes to read.

See Also

netbuf_buf2sd, netbuf_close, netbuf_grab, netbuf_open

netbuf_open

The netbuf_open function opens a new network buffer and returns it. You can use netbuf_open to create a netbuf structure and start using buffered I/O on a socket.

Syntax

netbuf* netbuf_open(SYS_NETFD sd, int sz);

Returns

A pointer to a new netbuf structure (network buffer).

Parameters

SYS_NETFD sd is the platform-independent identifier of the socket.

int sz is the number of characters to allocate for the network buffer.

See Also

netbuf_buf2sd, netbuf_close, netbuf_getc, netbuf_grab

nsapi_module_init

Plugin developers may define an nsapi_module_init function, which is a module initialization entry point that enables a plugin to create filters when it is loaded. When an NSAPI module contains an nsapi_module_init function, the server will call that function immediately after loading the module. The nsapi_module_init presents the same interface as an Init SAF, and it must follow the same rules.

The nsapi_module_init function may be used to register SAFs with func_insert, create filters with filter_create, register virtual server initialization/destruction callbacks with vs_register_cb, and perform other initialization tasks.

Syntax

int nsapi_module_init(pblock *pb, Session *sn, Request *rq);
NSAPI Functions (in Alphabetical Order)

Returns
REQ_PROCEED on success, or REQ_ABORTED on error.

Parameters
pblock *pb is a set of parameters specified by the fn="load-modules" directive.
Session *sn (the Session) is NULL.
Request *rq (the Request) is NULL.

NSAPI_RUNTIME_VERSION

The NSAPI_RUNTIME_VERSION macro defines the NSAPI version available at runtime. This is the same as the highest NSAPI version supported by the server the plugin is running in. The NSAPI version is encoded as in USE_NSAPI_VERSION.

The value returned by the NSAPI_RUNTIME_VERSION macro is valid only in iPlanet™ Web Server 6.0, Netscape Enterprise Server 6.0, and Sun ONE Web Server 6.1. That is, the server must support NSAPI 3.1 for this macro to return a valid value. Additionally, to use NSAPI_RUNTIME_VERSION, you must compile against an nsapi.h header file that supports NSAPI 3.2 or higher.

Plugin developers should not attempt to set the value of the NSAPI_RUNTIME_VERSION macro directly. Instead, see the USE_NSAPI_VERSION macro.

Syntax
int NSAPI_RUNTIME_VERSION

Example
NSAPI_PUBLIC int log_nsapi_runtime_version(pbblock *pb, Session *sn, Request *rq) {
    log_error(LOG_INFO, "log-nsapi-runtime-version", sn, rq,
        "Server supports NSAPI version %d.%d\n",
        NSAPI_RUNTIME_VERSION / 100, NSAPI_RUNTIME_VERSION % 100);
    return REQ_PROCEED;
}

See Also
NSAPI_VERSION, USE_NSAPI_VERSION
**NSAPI_VERSION**

The `NSAPI_VERSION` macro defines the NSAPI version used at compile time. This value is determined by the value of the `USE_NSAPI_VERSION` macro, or, if the plugin developer did not define `USE_NSAPI_VERSION`, by the highest NSAPI version supported by the `nsapi.h` header the plugin was compiled against. The NSAPI version is encoded as in `USE_NSAPI_VERSION`.

Plugin developers should not attempt to set the value of the `NSAPI_VERSION` macro directly. Instead, see the `USE_NSAPI_VERSION` macro.

**Syntax**

```c
int NSAPI_VERSION
```

**Example**

```c
NSAPI_PUBLIC int log_nsapi_compile_time_version(pblock *pb, Session *sn, Request *rq) {
    log_error(LOG_INFORM, "log-nsapi-compile-time-version", sn, rq,
        "Plugin compiled against NSAPI version %d.%d\n",
        NSAPI_VERSION / 100,
        NSAPI_VERSION % 100);
    return REQ_PROCEED;
}
```

**See Also**

`NSAPI_RUNTIME_VERSION`, `USE_NSAPI_VERSION`

---

**param_create**

The `param_create` function creates a `pb_param` structure containing a specified name and value. The name and value are copied. Use this function to prepare a `pb_param` structure to be used in calls to `pblock` routines such as `pblock_pinsert`.

**Syntax**

```c
pb_param *param_create(char *name, char *value);
```
param_create

Returns
A pointer to a new pb_param structure.

Parameters
char *name is the string containing the name.
char *value is the string containing the value.

Example
pb_param *newpp = param_create("content-type","text/plain");
pblock_pinsert(newpp, rq->srvhdrs);

See Also
param_free, pblock_pinsert, pblock_remove

param_free

The param_free function frees the pb_param structure specified by pp and its associated structures. Use the param_free function to dispose a pb_param after removing it from a pblock with pblock_remove.

Syntax
int param_free(pb_param *pp);

Returns
1 if the parameter was freed or 0 if the parameter was NULL.

Parameters
pb_param *pp is the name-value pair stored in a pblock.

Example
if (param_free(pblock_remove("content-type", rq->srvhdrs)))
    return; /* we removed it */

See Also
param_create, pblock_pinsert, pblock_remove

pblock_copy

The pblock_copy function copies the entries of the source pblock and adds them into the destination pblock. Any previous entries in the destination pblock are left intact.
Syntax

void pblock_copy(pblock *src, pblock *dst);

Returns

void

Parameters

pblock *src is the source pblock.

pblock *dst is the destination pblock.

Names and values are newly allocated so that the original pblock may be freed, or the new pblock changed without affecting the original pblock.

See Also

pblock_create, pblock_dup, pblock_free, pblock_find, pblock_findval, pblock_remove, pblock_nvinsert

pblock_create

The pblock_create function creates a new pblock. The pblock maintains an internal hash table for fast name-value pair lookups.

Syntax

pblock *pblock_create(int n);

Returns

A pointer to a newly allocated pblock.

Parameters

int n is the size of the hash table (number of name-value pairs) for the pblock.

See Also

pblock_copy, pblock_dup, pblock_find, pblock_findval, pblock_free, pblock_nvinsert, pblock_remove

pblock_dup

The pblock_dup function duplicates a pblock. It is equivalent to a sequence of pblock_create and pblock_copy.
Syntax
pblock *pblock_dup(pblock *src);

Returns
A pointer to a newly allocated pblock.

Parameters
pblock *src is the source pblock.

See Also
pblock_create, pblock_find, pblock_findval, pblock_free,
pblock_nvinsert, pblock_remove

pblock_find

The pblock_find function finds a specified name-value pair entry in a pblock, and returns the pb_param structure. If you only want the value associated with the name, use the pblock_findval function.

This function is implemented as a macro.

Syntax
pb_param *pblock_find(char *name, pblock *pb);

Returns
A pointer to the pb_param structure if one was found, or NULL if name was not found.

Parameters
char *name is the name of a name-value pair.

pblock *pb is the pblock to be searched.

See Also
pblock_copy, pblock_dup, pblock_findval, pblock_free,
pblock_nvinsert, pblock_remove
pblock_findval

The `pblock_findval` function finds the value of a specified name in a pblock. If you just want the `pb_param` structure of the pblock, use the `pblock_find` function.

The pointer returned is a pointer into the pblock. Do not FREE it. If you want to modify it, do a `STRDUP` and modify the copy.

**Syntax**

```c
char *pblock_findval(char *name, pblock *pb);
```

**Returns**

A string containing the value associated with the name or NULL if no match was found.

**Parameters**

- `char *name` is the name of a name-value pair.
- `pblock *pb` is the pblock to be searched.

**Example**

see `pblock_nvinsert`.

**See Also**

`pblock_create`, `pblock_copy`, `pblock_find`, `pblock_free`, `pblock_nvinsert`, `pblock_remove`, `request_header`

pblock_free

The `pblock_free` function frees a specified pblock and any entries inside it. If you want to save a variable in the pblock, remove the variable using the function `pblock_remove` and save the resulting pointer.

**Syntax**

```c
void pblock_free(pblock *pb);
```

**Returns**

`void`

**Parameters**

- `pblock *pb` is the pblock to be freed.
See Also
pblock_copy, pblock_create, pblock_dup, pblock_find, pblock_findval, pblock_nvinsert, pblock_remove

pblock_nninsert

The pblock_nninsert function creates a new entry with a given name and a numeric value in the specified pblock. The numeric value is first converted into a string. The name and value parameters are copied.

Syntax
pb_param *pblock_nninsert(char *name, int value, pblock *pb);

Returns
A pointer to the new pb_param structure.

Parameters
char *name is the name of the new entry.
int value is the numeric value being inserted into the pblock. This parameter must be an integer. If the value you assign is not a number, then instead use the function pblock_nvinsert to create the parameter.
pblock *pb is the pblock into which the insertion occurs.

See Also
pblock_copy, pblock_create, pblock_find, pblock_free, pblock_nvinsert, pblock_remove, pblock_str2pblock

pblock_nvinsert

The pblock_nvinsert function creates a new entry with a given name and character value in the specified pblock. The name and value parameters are copied.

Syntax
pb_param *pblock_nvinsert(char *name, char *value, pblock *pb);

Returns
A pointer to the newly allocated pb_param structure.
Parameters
char *name is the name of the new entry.
char *value is the string value of the new entry.
pblock *pb is the pblock into which the insertion occurs.

Example
pblock_nvinsert("content-type", "text/html", rq->srvhdrs);

See Also
pblock_copy, pblock_create, pblock_find, pblock_free,
pblock_nvinsert, pblock_remove, pblock_str2pblock

pblock_pb2env
The pblock_pb2env function copies a specified pblock into a specified environment. The function creates one new environment entry for each name-value pair in the pblock. Use this function to send pblock entries to a program that you are going to execute.

Syntax
cchar **pblock_pb2env(pblock *pb, char **env);

Returns
A pointer to the environment.

Parameters
pblock *pb is the pblock to be copied.
char **env is the environment into which the pblock is to be copied.

See Also
pblock_copy, pblock_create, pblock_find, pblock_free,
pblock_nvinsert, pblock_remove, pblock_str2pblock

pblock_pblock2str
The pblock_pblock2str function copies all parameters of a specified pblock into a specified string. The function allocates additional nonheap space for the string if needed.

Use this function to stream the pblock for archival and other purposes.
NSAPI Functions (in Alphabetical Order)

Syntax
char *pblock_pblock2str(pblock *pb, char *str);

Returns
The new version of the str parameter. If str is NULL, this is a new string; otherwise, it is a reallocated string. In either case, it is allocated from the request’s memory pool.

Parameters
pblock *pb is the pblock to be copied.
char *str is the string into which the pblock is to be copied. It must have been allocated by MALLOC or REALLOC, not by PERM_MALLOC or PERM_REALLOC (which allocate from the system heap).

Each name-value pair in the string is separated from its neighbor pair by a space, and is in the format name="value."

See Also
pblock_copy, pblock_create, pblock_find, pblock_free, pblock_nvinsert, pblock_remove, pblock_str2pblock

pblock_pinsert
The function pblock_pinsert inserts a pb_param structure into a pblock.

Syntax
void pblock_pinsert(pb_param *pp, pblock *pb);

Returns
void

Parameters
pb_param *pp is the pb_param structure to insert.
pblock *pb is the pblock.

See Also
pblock_copy, pblock_create, pblock_find, pblock_free, pblock_nvinsert, pblock_remove, pblock_str2pblock
pblock_remove

The pblock_remove function removes a specified name-value entry from a specified pblock. If you use this function, you should eventually call param_free to deallocate the memory used by the pb_param structure.

Syntax

```c
pb_param *pblock_remove(char *name, pblock *pb);
```

Returns

A pointer to the named pb_param structure if it was found, or NULL if the named pb_param was not found.

Parameters

- `char *name` is the name of the pb_param to be removed.
- `pblock *pb` is the pblock from which the name-value entry is to be removed.

See Also

pblock_copy, pblock_create, pblock_find, pblock_free, pblock_nvinsert, param_create, param_free

pblock_str2pblock

The pblock_str2pblock function scans a string for parameter pairs, adds them to a pblock, and returns the number of parameters added.

Syntax

```c
int pblock_str2pblock(char *str, pblock *pb);
```

Returns

The number of parameter pairs added to the pblock, if any, or -1 if an error occurred.

Parameters

- `char *str` is the string to be scanned.

The name-value pairs in the string can have the format `name=value` or `name="value."`
All backslashes (\) must be followed by a literal character. If string values are found with no unescaped = signs (no name=), it assumes the names 1, 2, 3, and so on, depending on the string position. For example, if `pblock_str2pblock` finds "some strings together," the function treats the strings as if they appeared in name-value pairs as 1="some" 2="strings" 3="together."

`pblock *pb` is the `pblock` into which the name-value pairs are stored.

**See Also**
`pblock_copy`, `pblock_create`, `pblock_find`, `pblock_free`, `pblock_nvinsert`, `pblock_remove`, `pblock_pblock2str`

**PERM_CALLOC**

The `PERM_CALLOC` macro is a platform-independent substitute for the C library routine `calloc`. It allocates int size bytes of memory that persist after the request that is being processed has been completed. If pooled memory has been disabled in the configuration file (with the `pool-init` built-in SAF), `PERM_CALLOC` and `CALLOC` both obtain their memory from the system heap.

**Syntax**

```c
void *PERM_CALLOC(int size)
```

**Returns**

A void pointer to a block of memory.

**Parameters**

`int size` is the size in bytes of each element.

**Example**

```
char **name;
name = (char **) PERM_CALLOC(100);
```

**See Also**

`PERM_FREE`, `PERM_STRDUP`, `PERM_MALLOC`, `PERM_REALLOC`, `MALLOC`, `FREE`, `CALLOC`, `STRDUP`, `REALLOC`
PERM_FREE

The PERM_FREE macro is a platform-independent substitute for the C library routine free. It deallocates the persistent space previously allocated by PERM_MALLOC, PERM_CALLOC, or PERM_STRDUP. If pooled memory has been disabled in the configuration file (with the pool-init built-in SAF), PERM_FREE and FREE both deallocate memory in the system heap.

Syntax
PERM_FREE(void *ptr);

Returns
void

Parameters
void *ptr is a (void *) pointer to block of memory. If the pointer is not one created by PERM_MALLOC, PERM_CALLOC, or PERM_STRDUP, the behavior is undefined.

Example
char *name;
name = (char *) PERM_MALLOC(256);
...
PERM_FREE(name);

See Also
FREE, MALLOC, CALLOC, REALLOC, STRDUP, PERM_MALLOC, PERM_CALLOC,
PERM_REALLOC, PERM_STRDUP

PERM_MALLOC

The PERM_MALLOC macro is a platform-independent substitute for the C library routine malloc. It provides allocation of memory that persists after the request that is being processed has been completed. If pooled memory has been disabled in the configuration file (with the pool-init built-in SAF), PERM_MALLOC and MALLOC both obtain their memory from the system heap.

Syntax
void *PERM_MALLOC(int size)

Returns
A void pointer to a block of memory.
NSAPI Functions (in Alphabetical Order)

Parameters

int size is the number of bytes to allocate.

Example

/* Allocate 256 bytes for a name */
char *name;
name = (char *) PERM_MALLOC(256);

See Also

PERM_FREE, PERM_STRDUP, PERM_CALLOC, PERM_REALLOC, MALLOC, FREE,
CALLOC, STRDUP, REALLOC

PERM_REALLOC

The PERM_REALLOC macro is a platform-independent substitute for the C library routine realloc. It changes the size of a specified memory block that was originally created by MALLOC, CALLOC, or STRDUP. The contents of the object remains unchanged up to the lesser of the old and new sizes. If the new size is larger, the new space is uninitialized.

Warning

Calling PERM_REALLOC for a block that was allocated with MALLOC, CALLOC, or STRDUP will not work.

Syntax

void *PERM_REALLOC(void *ptr, int size)

Returns

A void pointer to a block of memory.

Parameters

void *ptr a void pointer to a block of memory created by PERM_MALLOC,
PERM_CALLOC, or PERM_STRDUP.

int size is the number of bytes to which the memory block should be resized.

Example

char *name;
name = (char *) PERM_MALLOC(256);
if (NotBigEnough())
   name = (char *) PERM_REALLOC(512);
PERM_STRDUP

The PERM_STRDUP macro is a platform-independent substitute for the C library routine strdup. It creates a new copy of a string in memory that persists after the request that is being processed has been completed. If pooled memory has been disabled in the configuration file (with the pool-init built-in SAF), PERM_STRDUP and strdup both obtain their memory from the system heap.

The PERM_STRDUP routine is functionally equivalent to:

```c
newstr = (char *) PERM_MALLOC(strlen(str) + 1);
strcpy(newstr, str);
```

A string created with PERM_STRDUP should be disposed with PERM_FREE.

**Syntax**

```c
char *PERM_STRDUP(char *ptr);
```

**Returns**

A pointer to the new string.

**Parameters**

char *ptr is a pointer to a string.

**See Also**

PERM_MALLOC, PERM_FREE, PERM_CALLOC, PERM_REALLOC, MALLOC, FREE, STRDUP, CALLOC, REALLOC

prepare_nsapi_thread

The prepare_nsapi_thread function allows threads that are not created by the server to act like server-created threads. This function must be called before any NSAPI functions are called from a thread that is not server-created.
prepare_nsapi_thread

Syntax
void prepare_nsapi_thread(Request *rq, Session *sn);

Returns
void

Parameters
Request *rq is the Request.
Session *sn is the Session.

The Request and Session parameters are the same as the ones passed into your SAF.

See Also
protocol_start_response

protocol_dump822

The protocol_dump822 function prints headers from a specified pblock into a specific buffer, with a specified size and position. Use this function to serialize the headers so that they can be sent, for example, in a mail message.

Syntax
char *protocol_dump822(pblock *pb, char *t, int *pos, int tsz);

Returns
A pointer to the buffer, which will be reallocated if necessary.
The function also modifies *pos to the end of the headers in the buffer.

Parameters
pblock *pb is the pblock structure.
char *t is the buffer, allocated with MALLOC, CALLOC, or STRDUP.
int *pos is the position within the buffer at which the headers are to be dumped.
int tsz is the size of the buffer.

See Also
protocol_start_response, protocol_status
protocol_set_finfo

The `protocol_set_finfo` function retrieves the content-length and last-modified date from a specified stat structure and adds them to the response headers (rq->srvhdrs). Call `protocol_set_finfo` before calling `protocol_start_response`.

**Syntax**

```c
int protocol_set_finfo(Session *sn, Request *rq, struct stat *finfo);
```

**Returns**

The constant `REQ_PROCEED` if the request can proceed normally, or the constant `REQ_ABORTED` if the function should treat the request normally but not send any output to the client.

**Parameters**

- `Session *sn` is the Session.
- `Request *rq` is the Request.
- `stat *finfo` is the stat structure for the file.

The `Session` and `Request` parameters are the same as the ones passed into your SAF.

**See Also**

`protocol_start_response`, `protocol_status`

protocol_start_response

The `protocol_start_response` function initiates the HTTP response for a specified session and request. If the protocol version is HTTP/0.9, the function does nothing, because that version has no concept of status. If the protocol version is HTTP/1.0, the function sends a status line followed by the response headers. Use this function to set up HTTP and prepare the client and server to receive the body (or data) of the response.

**Syntax**

```c
int protocol_start_response(Session *sn, Request *rq);
```
Returns
The constant \texttt{REQ\_PROCEED} if the operation succeeded, in which case you should send the data you were preparing to send.

The constant \texttt{REQ\_NOACTION} if the operation succeeded but the request method was \texttt{HEAD}, in which case no data should be sent to the client.

The constant \texttt{REQ\_ABORTED} if the operation did not succeed.

Parameters
\begin{itemize}
\item \texttt{Session *sn} is the Session.
\item \texttt{Request *rq} is the Request.
\end{itemize}

The \texttt{Session} and \texttt{Request} parameters are the same as the ones passed into your SAF.

Example
\begin{verbatim}
/* A noaction response from this function means the request was HEAD */
if (protocol_start_response(sn, rq) == REQ_NOACTION) {
    filebuf_close(groupbuf);  /* close our file*/
    return REQ_PROCEED;
}
\end{verbatim}

See Also
\texttt{protocol\_status}

\textbf{protocol\_status}

The \texttt{protocol\_status} function sets the session status to indicate whether an error condition occurred. If the reason string is NULL, the server attempts to find a reason string for the given status code. If it finds none, it returns “Unknown reason.” The reason string is sent to the client in the HTTP response line. Use this function to set the status of the response before calling the function \texttt{protocol\_start\_response}.

For the complete list of valid status code constants, please refer to the file “\texttt{nsapi.h}” in the server distribution.

Syntax
\begin{verbatim}
void protocol_status(Session *sn, Request *rq, int n, char *r);
\end{verbatim}
**NSAPI Functions (in Alphabetical Order)**

**Chapter 7 NSAPI Function Reference 201**

**Returns**

void, but it sets values in the Session/Request designated by *sn/*rq for the status code and the reason string.

**Parameters**

Session *sn is the Session.

Request *rq is the Request.

The Session and Request parameters are the same as the ones passed into your SAF.

int *n is one of the status code constants above.

char *r is the reason string.

**Example**

/* if we find extra path-info, the URL was bad so tell the */
/* browser it was not found */
if (t = pblock_findval("path-info", rq->vars)) {
    protocol_status(sn, rq, PROTOCOL_NOT_FOUND, NULL);
    log_error(LOG_WARN, "function-name", sn, rq, "%s not found",
              path);
    return REQ_ABORTED;
}

**See Also**

protocol_start_response

**protocol_uri2url**

The protocol_uri2url function takes strings containing the given URI prefix and URI suffix, and creates a newly allocated, fully qualified URL in the form http://(server):(port)(prefix)(suffix). See protocol_uri2url_dynamic.

If you want to omit either the URI prefix or suffix, use "" instead of NULL as the value for either parameter.

**Syntax**

char *protocol_uri2url(char *prefix, char *suffix);

**Returns**

A new string containing the URL.
NSAPI Functions (in Alphabetical Order)

Parameters

char *prefix is the prefix.
char *suffix is the suffix.

See Also

protocol_start_response, protocol_status, pblock_nvinsert, protocol_uri2url_dynamic

protocol_uri2url_dynamic

The protocol_uri2url function takes strings containing the given URI prefix and URI suffix, and creates a newly allocated, fully qualified URL in the form http://(server):(port)(prefix)(suffix).

If you want to omit either the URI prefix or suffix, use "" instead of NULL as the value for either parameter.

The protocol_uri2url_dynamic function is similar to the protocol_uri2url function, but should be used whenever the session and request structures are available. This ensures that the URL it constructs refers to the host that the client specified.

Syntax

char *protocol_uri2url(char *prefix, char *suffix, Session *sn, Request *rq);

Returns

A new string containing the URL.

Parameters

char *prefix is the prefix.
char *suffix is the suffix.
Session *sn is the Session.
Request *rq is the Request.

The Session and Request parameters are the same as the ones passed into your SAF.

See Also

protocol_start_response, protocol_status, protocol_uri2url_dynamic
read

The read filter method is called when input data is required. Filters that modify or consume incoming data should implement the read filter method.

Upon receiving control, a read implementation should fill buf with up to amount bytes of input data. This data may be obtained by calling the net_read function, as shown in the example below.

Syntax

```c
int read(FilterLayer *layer, void *buf, int amount, int timeout);
```

Returns

The number of bytes placed in buf on success, 0 if no data is available, or a negative value if an error occurred.

Parameters

FilterLayer *layer is the filter layer in which the filter is installed.
void *buf is the buffer in which data should be placed.
int amount is the maximum number of bytes that should be placed in the buffer.
int timeout is the number of seconds to allow for the read operation before returning. The purpose of timeout is not to return because not enough bytes were read in the given time, but to limit the amount of time devoted to waiting until some data arrives.

Example

```c
int myfilter_read(FilterLayer *layer, void *buf, int amount, int timeout)
{
    return net_read(layer->lower, buf, amount, timeout);
}
```

See Also

net_read
REALLOC

The REALLOC macro is a platform-independent substitute for the C library routine realloc. It changes the size of a specified memory block that was originally created by MALLOC, CALLOC, or STRDUP. The contents of the object remains unchanged up to the lesser of the old and new sizes. If the new size is larger, the new space is uninitialized.

Warning
Calling REALLOC for a block that was allocated with PERM_MALLOC, PERM_CALLOC, or PERM_STRDUP will not work.

Syntax
void *REALLOC(void *ptr, int size);

Returns
A pointer to the new space if the request could be satisfied.

Parameters
void *ptr is a (void *) pointer to a block of memory. If the pointer is not one created by MALLOC, CALLOC, or STRDUP, the behavior is undefined.

int size is the number of bytes to allocate.

Example
char *name;
nname = (char *) MALLOC(256);
if (!NotBigEnough())
    name = (char *) REALLOC(512);

See Also
MALLOC, FREE, STRDUP, CALLOC, PERM_MALLOC, PERM_FREE, PERM_REALLOC, PERM_CALLOC, PERM_STRDUP

remove

The remove filter method is called when the filter stack is destroyed, or when a filter is removed from a filter stack by the filter_remove function or remove-filter SAF (applicable in Input-, Output-, Service-, and Error-class directives).
Note that it may be too late to flush buffered data when the `remove` method is invoked. For this reason, filters that buffer outgoing data should implement the `flush` filter method.

**Syntax**

```c
void remove(FilterLayer *layer);
```

**Returns**

`void`

**Parameters**

`FilterLayer *layer` is the filter layer the filter is installed in.

**See Also**

`flush`

---

**request_get_vs**

The `request_get_vs` function finds the `VirtualServer*` to which a request is directed.

The returned `VirtualServer*` is valid only for the current request. To retrieve a virtual server ID that is valid across requests, use `vs_get_id`.

**Syntax**

```c
const VirtualServer* request_get_vs(Request* rq);
```

**Returns**

The `VirtualServer*` to which the request is directed.

**Parameters**

`Request *rq` is the request for which the `VirtualServer*` is returned.

**See Also**

`vs_get_id`

---

**request_header**

The `request_header` function finds an entry in the `pblock` containing the client’s HTTP request headers (`rq->headers`). You must use this function rather than `pblock_findval` when accessing the client headers, since the server may begin processing the request before the headers have been completely read.
Syntax
int request_header(char *name, char **value, Session *sn, Request *rq);

Returns
A result code, REQ_PROCEED if the header was found, REQ_ABORTED if the header was not found, REQ_EXIT if there was an error reading from the client.

Parameters
char *name is the name of the header.
char **value is the address where the function will place the value of the specified header. If none is found, the function stores a NULL.
Session *sn is the Session.
Request *rq is the Request.
The Session and Request parameters are the same as the ones passed into your SAF.

See Also
request_create, request_free

request_stat_path

The request_stat_path function returns the file information structure for a specified path or, if none is specified, the path entry in the vars pblock in the specified request structure. If the resulting file name points to a file that the server can read, request_stat_path returns a new file information structure. This structure contains information on the size of the file, its owner, when it was created, and when it was last modified.

You should use request_stat_path to retrieve information on the file you are currently accessing (instead of calling stat directly), because this function keeps track of previous calls for the same path and returns its cached information.

Syntax
struct stat *request_stat_path(char *path, Request *rq);
Returns
Returns a pointer to the file information structure for the file named by the path parameter. Do not free this structure. Returns NULL if the file is not valid or the server cannot read it. In this case, it also leaves an error message describing the problem in rq->staterr.

Parameters
char *path is the string containing the name of the path. If the value of path is NULL, the function uses the path entry in the vars pblock in the request structure denoted by rq.

Request *rq is the request identifier for a Server Application Function call.

Example
fi = request_stat_path(path, rq);

See Also
request_create, request_free, request_header

request_translate_uri

The request_translate_uri function performs virtual to physical mapping on a specified URI during a specified session. Use this function when you want to determine which file would be sent back if a given URI is accessed.

Syntax
char *request_translate_uri(char *uri, Session *sn);

Returns
A path string if it performed the mapping, or NULL if it could not perform the mapping.

Parameters
char *uri is the name of the URI.
Session *sn is the Session parameter that is passed into your SAF.

See Also
request_create, request_free, request_header
sendfile

The sendfile filter method is called when the contents of a file are to be sent. Filters that modify or consume outgoing data may choose to implement the sendfile filter method.

If a filter implements the write filter method but not the sendfile filter method, the server will automatically translate net_sendfile calls to net_write calls. As a result, filters interested in the outgoing data stream do not need to implement the sendfile filter method. However, for performance reasons, it is beneficial for filters that implement the write filter method to also implement the sendfile filter method.

Syntax

```
int sendfile(FilterLayer *layer, const sendfiledata *data);
```

Returns

The number of bytes consumed, which may be less than the requested amount if an error occurred.

Parameters

- `FilterLayer *layer` is the filter layer the filter is installed in.
- `const sendfiledata *sfd` identifies the data to send.

Example

```
int myfilter_sendfile(FilterLayer *layer, const sendfiledata *sfd)
{
    return net_sendfile(layer->lower, sfd);
}
```

See Also

- `net_sendfile`
The session_dns function resolves the IP address of the client associated with a specified session into its DNS name. It returns a newly allocated string. You can use session_dns to change the numeric IP address into something more readable.

The session_maxdns function verifies that the client is who it claims to be; the session_dns function does not perform this verification.

**NOTE**  This function works only if the DNS directive is enabled in the magnus.conf file. For more information, see Chapter 2, “SAFs in the magnus.conf File.”

### Syntax

```c
char *session_dns(Session *sn);
```

### Returns

A string containing the host name, or NULL if the DNS name cannot be found for the IP address.

### Parameters

- `Session *sn` is the Session.

The Session is the same as the one passed to your SAF.

### session_maxdns

The session_maxdns function resolves the IP address of the client associated with a specified session into its DNS5 name. It returns a newly allocated string. You can use session_maxdns to change the numeric IP address into something more readable.

**NOTE**  This function works only if the DNS directive is enabled in the magnus.conf file. For more information, see Chapter 2, “SAFs in the magnus.conf File.”

### Syntax

```c
char *session_maxdns(Session *sn);
```
Returns
A string containing the host name, or NULL if the DNS name cannot be found for the IP address.

Parameters
Session *sn is the Session.

The Session is the same as the one passed to your SAF.

shexp_casecmp

The shexp_casecmp function validates a specified shell expression and compares it with a specified string. It returns one of three possible values representing match, no match, and invalid comparison. The comparison (in contrast to that of the shexp_cmp function) is not case-sensitive.

Use this function if you have a shell expression like *\.netscape\..com and you want to make sure that a string matches it, such as foo\.netscape\.com.

Syntax
int shexp_casecmp(char *str, char *exp);

Returns
0 if a match was found.
1 if no match was found.
-1 if the comparison resulted in an invalid expression.

Parameters
char *str is the string to be compared.
char *exp is the shell expression (wildcard pattern) to compare against.

See Also
shexp_cmp, shexp_match, shexp_valid

shexp_cmp

The shexp_casecmp function validates a specified shell expression and compares it with a specified string. It returns one of three possible values representing match, no match, and invalid comparison. The comparison (in contrast to that of the shexp_casecmp function) is case-sensitive.
Use this function if you have a shell expression like *.netscape.com and you want to make sure that a string matches it, such as foo.netscape.com.

Syntax
int shexp_cmp(char *str, char *exp);

Returns
0 if a match was found.
1 if no match was found.
-1 if the comparison resulted in an invalid expression.

Parameters
char *str is the string to be compared.
char *exp is the shell expression (wildcard pattern) to compare against.

Example
/* Use wildcard match to see if this path is one we want */
char *path;
char *match = "/usr/netscape/*";
if (shexp_cmp(path, match) != 0)
    return REQ_NOACTION; /* no match */

See Also
shexp_casecmp, shexp_match, shexp_valid

shexp_match

The shexp_match function compares a specified prevalidated shell expression against a specified string. It returns one of three possible values representing match, no match, and invalid comparison. The comparison (in contrast to that of the shexp_casecmp function) is case-sensitive.

The shexp_match function doesn’t perform validation of the shell expression; instead the function assumes that you have already called shexp_valid.

Use this function if you have a shell expression such as *.netscape.com, and you want to make sure that a string matches it, such as foo.netscape.com.

Syntax
int shexp_match(char *str, char *exp);
shexp_valid

The `shexp_valid` function validates a specified shell expression named by `exp`. Use this function to validate a shell expression before using the function `shexp_match` to compare the expression with a string.

**Syntax**

```c
int shexp_valid(char *exp);
```

**Returns**

The constant `NON_SXP` if `exp` is a standard string.

The constant `INVALID_SXP` if `exp` is a shell expression, but invalid.

The constant `VALID_SXP` if `exp` is a valid shell expression.

**Parameters**

- `char *exp` is the shell expression (wildcard pattern) to be validated.

**See Also**

`shexp_casecmp`, `shexp_match`, `shexp_cmp`
STRDUP

The STRDUP macro is a platform-independent substitute for the C library routine strdup. It creates a new copy of a string in the request’s memory pool.

The STRDUP routine is functionally equivalent to:

```c
newstr = (char *) MALLOC(strlen(str) + 1);
strcpy(newstr, str);
```

A string created with STRDUP should be disposed with FREE.

Syntax
char *STRDUP(char *ptr);

Returns
A pointer to the new string.

Parameters
char *ptr is a pointer to a string.

Example
```c
char *name1 = "MyName";
char *name2 = STRDUP(name1);
```

See Also
MALLOC, FREE, CALLOC, REALLOC, PERM_MALLOC, PERM_FREE, PERM_CALLOC, PERM_REALLOC, PERM_STRDUP

system_errmsg

The system_errmsg function returns the last error that occurred from the most recent system call. This function is implemented as a macro that returns an entry from the global array sys_errlist. Use this macro to help with I/O error diagnostics.

Syntax
char *system_errmsg(int param1);
Returns
A string containing the text of the latest error message that resulted from a system call. Do not FREE this string.

Parameters
int param1 is reserved, and should always have the value 0.

See Also
system_fopenRO, system_fopenRW, system_fopenWA, system_lseek, system_fread, system_fwrite, system_fwrite_atomic, system_flock, system_ulock, system_fclose

system_fclose
The system_fclose function closes a specified file descriptor. The system_fclose function must be called for every file descriptor opened by any of the system_fopen functions.

Syntax
int system_fclose(SYS_FILE fd);

Returns
0 if the close succeeded, or the constant IO_ERROR if the close failed.

Parameters
SYS_FILE fd is the platform-independent file descriptor.

Example
SYS_FILE logfd;
system_fclose(logfd);

See Also
system_errmsg, system_fopenRO, system_fopenRW, system_fopenWA, system_lseek, system_fread, system_fwrite, system_fwrite_atomic, system_flock, system_ulock
system_flock

The system_flock function locks the specified file against interference from other processes. Use system_flock if you do not want other processes to use the file you currently have open. Overusing file locking can cause performance degradation and possibly lead to deadlocks.

Syntax
int system_flock(SYS_FILE fd);

Returns
The constant IO_OKAY if the lock succeeded, or the constant IO_ERROR if the lock failed.

Parameters
SYS_FILE fd is the platform-independent file descriptor.

See Also
system_errmsg, system_fopenRO, system_fopenRW, system_fopenWA, system_lseek, system_fread, system_fwrite, system_fwrite_atomic, system_ulock, system_fclose

system_fopenRO

The system_fopenRO function opens the file identified by path in read-only mode and returns a valid file descriptor. Use this function to open files that will not be modified by your program. In addition, you can use system_fopenRO to open a new file buffer structure using filebuf_open.

Syntax
SYS_FILE system_fopenRO(char *path);

Returns
The system-independent file descriptor (SYS_FILE) if the open succeeded, or 0 if the open failed.

Parameters
char *path is the file name.

See Also
system_errmsg, system_fopenRW, system_fopenWA, system_lseek, system_fread, system_fwrite, system_fwrite_atomic, system_flock, system_ulock, system_fclose
system_fopenRW

The system_fopenRW function opens the file identified by path in read-write mode and returns a valid file descriptor. If the file already exists, system_fopenRW does not truncate it. Use this function to open files that will be read from and written to by your program.

Syntax
SYS_FILE system_fopenRW(char *path);

Returns
The system-independent file descriptor (SYS_FILE) if the open succeeded, or 0 if the open failed.

Parameters
char *path is the file name.

Example
SYS_FILE fd;
fd = system_fopenRO(pathname);
if (fd == SYS_ERROR_FD)
    break;

See Also
system_errmsg, system_fopenRO, system_fopenWA, system_lseek,
system_fread, system_fwrite, system_fwrite_atomic, system_flock,
system_ulock, system_fclose

system_fopenWA

The system_fopenWA function opens the file identified by path in write-append mode and returns a valid file descriptor. Use this function to open those files to which your program will append data.

Syntax
SYS_FILE system_fopenWA(char *path);

Returns
The system-independent file descriptor (SYS_FILE) if the open succeeded, or 0 if the open failed.

Parameters
char *path is the file name.
system_fread

The `system_fread` function reads a specified number of bytes from a specified file into a specified buffer. It returns the number of bytes read. Before `system_fread` can be used, you must open the file using any of the `system_fopen` functions (except `system_fopenWA`).

Syntax

```c
int system_fread(SYS_FILE fd, char *buf, int sz);
```

Returns

The number of bytes read, which may be less than the requested size if an error occurred or the end of the file was reached before that number of characters were obtained.

Parameters

- **SYS_FILE fd** is the platform-independent file descriptor.
- **char *buf** is the buffer to receive the bytes.
- **int sz** is the number of bytes to read.

See Also

`system_errmsg`, `system_fopenRO`, `system_fopenRW`, `system_lseek`, `system_fread`, `system_fwrite`, `system_fwrite_atomic`, `system_flock`, `system_ulock`, `system_fclose`

system_fwrite

The `system_fwrite` function writes a specified number of bytes from a specified buffer into a specified file.

Before `system_fwrite` can be used, you must open the file using any of the `system_fopen` functions (except `system_fopenRO`).

Syntax

```c
int system_fwrite(SYS_FILE fd, char *buf, int sz);
```
Returns
The constant IO_OKAY if the write succeeded, or the constant IO_ERROR if the write failed.

Parameters
SYS_FILE fd is the platform-independent file descriptor.
char *buf is the buffer containing the bytes to be written.
int sz is the number of bytes to write to the file.

See Also
system_errmsg, system_fopenRO, system_fopenRW, system_fopenWA, system_lseek, system_fread, system_fwrite_atomic, system_flock, system_ulock, system_fclose

system_fwrite_atomic
The system_fwrite_atomic function writes a specified number of bytes from a specified buffer into a specified file. The function also locks the file prior to performing the write, and then unlocks it when done, thereby avoiding interference between simultaneous write actions. Before system_fwrite_atomic can be used, you must open the file using any of the system_fopen functions, except system_fopenRO.

Syntax
int system_fwrite_atomic(SYS_FILE fd, char *buf, int sz);

Returns
The constant IO_OKAY if the write/lock succeeded, or the constant IO_ERROR if the write/lock failed.

Parameters
SYS_FILE fd is the platform-independent file descriptor.
char *buf is the buffer containing the bytes to be written.
int sz is the number of bytes to write to the file.

Example
SYS_FILE logfd;
char *logmsg = "An error occurred."
system_fwrite_atomic(logfd, logmsg, strlen(logmsg));
system_gmtime

The system_gmtime function is a thread-safe version of the standard gmtime function. It returns the current time adjusted to Greenwich Mean Time.

Syntax

```
struct tm *system_gmtime(const time_t *tp, const struct tm *res);
```

Returns

A pointer to a calendar time (tm) structure containing the GMT time. Depending on your system, the pointer may point to the data item represented by the second parameter, or it may point to a statically-allocated item. For portability, do not assume either situation.

Parameters

- `time_t *tp` is an arithmetic time.
- `tm *res` is a pointer to a calendar time (tm) structure.

Example

```
time_t tp;
struct tm res, *resp;
tp = time(NULL);
resp = system_gmtime(&tp, &res);
```

See Also

system_localtime, util_strftime

system_localtime

The system_localtime function is a thread-safe version of the standard localtime function. It returns the current time in the local time zone.

Syntax

```
struct tm *system_localtime(const time_t *tp, const struct tm *res);
```
Returns
A pointer to a calendar time (tm) structure containing the local time. Depending on your system, the pointer may point to the data item represented by the second parameter, or it may point to a statically-allocated item. For portability, do not assume either situation.

Parameters
time_t *tp is an arithmetic time.
tm *res is a pointer to a calendar time (tm) structure.

See Also
system_gmtime, util_strftime

system_lseek
The system_lseek function sets the file position of a file. This affects where data from system_fread or system_fwrite is read or written.

Syntax
int system_lseek(SYS_FILE fd, int offset, int whence);

Returns
The offset, in bytes, of the new position from the beginning of the file if the operation succeeded, or -1 if the operation failed.

Parameters
SYS_FILE fd is the platform-independent file descriptor.
int offset is a number of bytes relative to whence. It may be negative.
int whence is one of the following constants:
SEEK_SET, from the beginning of the file.
SEEK_CUR, from the current file position.
SEEK_END, from the end of the file.

See Also
system_errmsg, system_fopenRO, system_fopenRW, system_fopenWA, system_fread, system_fwrite, system_fwrite_atomic, system_flock, system_ulock, system_fclose
system_rename

The `system_rename` function renames a file. It may not work on directories if the old and new directories are on different file systems.

Syntax

```c
int system_rename(char *old, char *new);
```

Returns

0 if the operation succeeded, or -1 if the operation failed.

Parameters

- `char *old` is the old name of the file.
- `char *new` is the new name for the file.

system_ulock

The `system_ulock` function unlocks the specified file that has been locked by the function `system_lock`. For more information about locking, see `system_flock`.

Syntax

```c
int system_ulock(SYS_FILE fd);
```

Returns

The constant `IO_OKAY` if the operation succeeded, or the constant `IO_ERROR` if the operation failed.

Parameters

- `SYS_FILE fd` is the platform-independent file descriptor.

See Also

`system_errmsg`, `system_fopenRO`, `system_fopenRW`, `system_fopenWA`, `system_fread`, `system_fwrite`, `system_fwrite_atomic`, `system_flock`, `system_fclose`
**system_unix2local**

The `system_unix2local` function converts a specified UNIX-style path name to a local file system path name. Use this function when you have a file name in the UNIX format (such as one containing forward slashes), and you need to access a file on another system such as Windows. You can use `system_unix2local` to convert the UNIX file name into the format that Windows accepts. In the UNIX environment this function does nothing, but may be called for portability.

**Syntax**

```
char *system_unix2local(char *path, char *lp);
```

**Returns**

A pointer to the local file system path string.

**Parameters**

- `char *path` is the UNIX-style path name to be converted.
- `char *lp` is the local path name.

You must allocate the parameter `lp`, and it must contain enough space to hold the local path name.

**See Also**

`system_fclose`, `system_flock`, `system_fopenRO`, `system_fopenRW`, `system_fopenWA`, `system_fwrite`

**systhread_attach**

The `systhread_attach` function makes an existing thread into a platform-independent thread.

**Syntax**

```
SYS_THREAD systhread_attach(void);
```

**Returns**

A `SYS_THREAD` pointer to the platform-independent thread.

**Parameters**

none
systhread_current

The `systhread_current` function returns a pointer to the current thread.

**Syntax**

```c
SYS_THREAD systhread_current(void);
```

**Returns**

A `SYS_THREAD` pointer to the current thread.

**Parameters**

none

**See Also**

`systhread_getdata`, `systhread_newkey`, `systhread_setdata`, `systhread_sleep`, `systhread_start`, `systhread_timerset`

systhread_getdata

The `systhread_getdata` function gets data that is associated with a specified key in the current thread.

**Syntax**

```c
void *systhread_getdata(int key);
```

**Returns**

A pointer to the data that was earlier used with the `systhread_setkey` function from the current thread, using the same value of `key` if the call succeeds. Returns NULL if the call did not succeed; for example, if the `systhread_setkey` function was never called with the specified key during this session.

**Parameters**

- `int key` is the value associated with the stored data by a `systhread_setdata` function. Keys are assigned by the `systhread_newkey` function.
systhread_newkey

The **systhread_newkey** function allocates a new integer key (identifier) for thread-private data. Use this key to identify a variable that you want to localize to the current thread, then use the **systhread_setdata** function to associate a value with the key.

**Syntax**

```c
int systhread_newkey(void);
```

**Returns**

An integer key.

**Parameters**

none

**See Also**

systhread_current, systhread_newkey, systhread_getdata, systhread_setdata, systhread_sleep, systhread_start, systhread_timerset

systhread_setdata

The **systhread_setdata** function associates data with a specified key number for the current thread. Keys are assigned by the **systhread_newkey** function.

**Syntax**

```c
void systhread_setdata(int key, void *data);
```

**Returns**

void

**Parameters**

- `int key` is the priority of the thread.
- `void *data` is the pointer to the string of data to be associated with the value of key.
systhread_sleep

The `systhread_sleep` function puts the calling thread to sleep for a given time.

Syntax
```c
void systhread_sleep(int milliseconds);
```

Returns
void

Parameters
- `int milliseconds` is the number of milliseconds the thread is to sleep.

See Also
- `systhread_current`, `systhread_getdata`, `systhread_newkey`, `systhread_start`, `systhread_timerset`

systhread_start

The `systhread_start` function creates a thread with the given priority, allocates a stack of a specified number of bytes, and calls a specified function with a specified argument.

Syntax
```c
SYS_THREAD systhread_start(int prio, int stksz,
       void (*fn)(void *), void *arg);
```

Returns
A new `SYS_THREAD` pointer if the call succeeded, or the constant `SYS_THREAD_ERROR` if the call did not succeed.

Parameters
- `int prio` is the priority of the thread. Priorities are system-dependent.
- `int stksz` is the stack size in bytes. If `stksz` is zero (0), the function allocates a default size.
- `void (*fn)(void *)` is the function to call.
void *arg is the argument for the fn function.

See Also
systhread_current, systhread_getdata, systhread_newkey,
systhread_setdata, systhread_sleep, systhread_timerset

systhread_timerset

The systhread_timerset function starts or resets the interrupt timer interval for
a thread system.

Because most systems don’t allow the timer interval to be changed, this should be
considered a suggestion, rather than a command.

Syntax
void systhread_timerset(int usec);

Returns
void

Parameters
int usec is the time, in microseconds

See Also
systhread_current, systhread_getdata, systhread_newkey,
systhread_setdata, systhread_sleep, systhread_start

USE_NSAPI_VERSION

Plugin developers can define the USE_NSAPI_VERSION macro before including the
nsapi.h header file to request a particular version of NSAPI. The requested
NSAPI version is encoded by multiplying the major version number by 100 and
then adding this to the minor version number. For example, the following code
requests NSAPI 3.2 features:

#define USE_NSAPI_VERSION 302 /* We want NSAPI 3.2 (Web Server 6.1) */
To develop a plugin that is compatible across multiple server versions, define `USE_NSAPI_VERSION` to the highest NSAPI version supported by all of the target server versions.

The following table lists server versions and the highest NSAPI version supported by each:

<table>
<thead>
<tr>
<th>Server Version</th>
<th>NSAPI Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPlanet Web Server 4.1</td>
<td>3.0</td>
</tr>
<tr>
<td>iPlanet Web Server 6.0</td>
<td>3.1</td>
</tr>
<tr>
<td>Netscape Enterprise Server 6.0</td>
<td>3.1</td>
</tr>
<tr>
<td>Netscape Enterprise Server 6.1</td>
<td>3.1</td>
</tr>
<tr>
<td>Sun ONE Application Server 7.0</td>
<td>3.1</td>
</tr>
<tr>
<td>Sun ONE Web Server 6.1</td>
<td>3.2</td>
</tr>
</tbody>
</table>

It is an error to request a version of NSAPI higher than the highest version supported by the `nsapi.h` header that the plugin is being compiled against. Additionally, to use `USE_NSAPI_VERSION`, you must compile against an `nsapi.h` header file that supports NSAPI 3.2 or higher.

**Syntax**

```c
int USE_NSAPI_VERSION
```

**Example**

The following code can be used when building a plugin designed to work with iPlanet Web Server 4.1 and Sun ONE Web Server 6.1:

```c
#define USE_NSAPI_VERSION 300 /* We want NSAPI 3.0 (Web Server 4.1) */
#include "nsapi.h"
```

**See Also**

`NSAPI_RUNTIME_VERSION`, `NSAPI_VERSION`
util_can_exec

UNIX Only
The `util_can_exec` function checks that a specified file can be executed, returning either a 1 (executable) or a 0. The function checks if the file can be executed by the user with the given user and group ID.

Use this function before executing a program using the `exec` system call.

Syntax
```c
int util_can_exec(struct stat *finfo, uid_t uid, gid_t gid);
```

Returns
1 if the file is executable, or 0 if the file is not executable.

Parameters
- `stat *finfo` is the `stat` structure associated with a file.
- `uid_t uid` is the UNIX user id.
- `gid_t gid` is the UNIX group id. Together with `uid`, this determines the permissions of the UNIX user.

See Also
util_env_create, util_getline, util_hostname

util_chdir2path

The `util_chdir2path` function changes the current directory to a specified directory, where you will access a file.

When running under Windows, use a critical section to ensure that more than one thread does not call this function at the same time.

Use `util_chdir2path` when you want to make file access a little quicker, because you do not need to use a full path.

Syntax
```c
int util_chdir2path(char *path);
```

Returns
0 if the directory was changed, or -1 if the directory could not be changed.
Parameters
char *path is the name of a directory.

The parameter must be a writable string because it isn’t permanently modified.

util_chdir2path

The `util_chdir2path` function changes the current directory to a specified directory, where you will access a file.

When running under Windows, use a critical section to ensure that more than one thread does not call this function at the same time.

Use `util_chdir2path` when you want to make file access a little quicker, because you do not need to use a full path.

Syntax
```c
int util_chdir2path(char *path);
```

Returns
0 if the directory was changed, or -1 if the directory could not be changed.

Parameters
char *path is the name of a directory.

The parameter must be a writable string because it isn’t permanently modified.

util_cookie_find

The `util_cookie_find` function finds a specific cookie in a cookie string and returns its value.

Syntax
```c
char *util_cookie_find(char *cookie, char *name);
```

Returns
If successful, returns a pointer to the NULL-terminated value of the cookie. Otherwise, returns NULL. This function modifies the cookie string parameter by null-terminating the name and value.

Parameters
char *cookie is the value of the `Cookie: request header`. 
char *name is the name of the cookie whose value is to be retrieved.

util_env_find

The util_env_find function locates the string denoted by a name in a specified environment and returns the associated value. Use this function to find an entry in an environment.

Syntax
char *util_env_find(char **env, char *name);

Returns
The value of the environment variable if it is found, or NULL if the string was not found.

Parameters
char **env is the environment.
char *name is the name of an environment variable in env.

See Also
util_env_replace, util_env_str, util_env_free, util_env_create

util_env_free

The util_env_free function frees a specified environment. Use this function to deallocate an environment you created using the function util_env_create.

Syntax
void util_env_free(char **env);

Returns
void

Parameters
char **env is the environment to be freed.

See Also
util_env_replace, util_env_str, util_env_find, util_env_create
util_env_replace

The util_env_replace function replaces the occurrence of the variable denoted by a name in a specified environment with a specified value. Use this function to change the value of a setting in an environment.

Syntax
void util_env_replace(char **env, char *name, char *value);

Returns
void

Parameters
char **env is the environment.
char *name is the name of a name-value pair.
char *value is the new value to be stored.

See Also
util_env_str, util_env_free, util_env_find, util_env_create

util_env_str

The util_env_str function creates an environment entry and returns it. This function does not check for nonalphanumeric symbols in the name (such as the equal sign “=”). You can use this function to create a new environment entry.

Syntax
char *util_env_str(char *name, char *value);

Returns
A newly allocated string containing the name-value pair.

Parameters
char *name is the name of a name-value pair.
char *value is the new value to be stored.

See Also
util_env_replace, util_env_free, util_env_find, util_env_create
util_getline

The `util_getline` function scans the specified file buffer to find a line feed or carriage return/line feed terminated string. The string is copied into the specified buffer, and NULL-terminates it. The function returns a value that indicates whether the operation stored a string in the buffer, encountered an error, or reached the end of the file.

Use this function to scan lines out of a text file, such as a configuration file.

**Syntax**

```c
int util_getline(filebuf *buf, int lineno, int maxlen, char *line);
```

**Returns**

0 if successful; `line` contains the string.

1 if the end of file was reached; `line` contains the string.

-1 if an error occurred; `line` contains a description of the error.

**Parameters**

- `filebuf *buf` is the file buffer to be scanned.
- `int lineno` is used to include the line number in the error message when an error occurs. The caller is responsible for making sure the line number is accurate.
- `int maxlen` is the maximum number of characters that can be written into `line`.
- `char *line` is the buffer in which to store the string. The user is responsible for allocating and deallocating `line`.

**See Also**

`util_can_exec`, `util_env_create`, `util_hostname`

util_hostname

The `util_hostname` function retrieves the local host name and returns it as a string. If the function cannot find a fully-qualified domain name, it returns NULL. You may reallocate or free this string. Use this function to determine the name of the system you are on.

**Syntax**

```c
char *util_hostname(void);
```
Returns
If a fully-qualified domain name was found, returns a string containing that name; otherwise, returns NULL if the fully-qualified domain name was not found.

Parameters
none

util_is_mozilla

The `util_is_mozilla` function checks whether a specified user-agent header string is a Netscape browser of at least a specified revision level, returning a 1 if it is, and 0 otherwise. It uses strings to specify the revision level to avoid ambiguities such as 1.56 > 1.5.

Syntax
```
int util_is_mozilla(char *ua, char *major, char *minor);
```

Returns
1 if the user-agent is a Netscape browser, or 0 if the user-agent is not a Netscape browser.

Parameters
cchar *ua is the user-agent string from the request headers.
cchar *major is the major release number (to the left of the decimal point).
cchar *minor is the minor release number (to the right of the decimal point).

See Also
util_is_url, util_later_than

util_is_url

The `util_is_url` function checks whether a string is a URL, returning 1 if it is and 0 otherwise. The string is a URL if it begins with alphabetic characters followed by a colon (:).

Syntax
```
int util_is_url(char *url);
```
NSAPI Functions (in Alphabetical Order)

Returns
1 if the string specified by url is a URL, or 0 if the string specified by url is not a URL.

Parameters
char *url is the string to be examined.

See Also
util_is_mozilla, util_later_than

util_itoa
The util_itoa function converts a specified integer to a string, and returns the length of the string. Use this function to create a textual representation of a number.

Syntax
int util_itoa(int i, char *a);

Returns
The length of the string created.

Parameters
int i is the integer to be converted.
char *a is the ASCII string that represents the value. The user is responsible for the allocation and deallocation of a, and it should be at least 32 bytes long.

util_later_than
The util_later_than function compares the date specified in a time structure against a date specified in a string. If the date in the string is later than or equal to the one in the time structure, the function returns 1. Use this function to handle RFC 822, RFC 850, and ctime formats.

Syntax
int util_later_than(struct tm *lms, char *ims);

Returns
1 if the date represented by ims is the same as or later than that represented by the lms, or 0 if the date represented by ims is earlier than that represented by the lms.
Parameters

`tm *lms` is the time structure containing a date.

`char *ims` is the string containing a date.

See Also

`util_strftime`

util_sh_escape

The `util_sh_escape` function parses a specified string and places a backslash (`\`) in front of any shell-special characters, returning the resultant string. Use this function to ensure that strings from clients won’t cause a shell to do anything unexpected.

The shell-special characters are the space plus the following characters:

`&;=~|^*?-<>^()[]{}$\#!

Syntax

```c
char *util_sh_escape(char *s);
```

Returns

A newly allocated string.

Parameters

`char *s` is the string to be parsed.

See Also

`util_uri_escape`

util_snprintf

The `util_snprintf` function formats a specified string, using a specified format, into a specified buffer using the `printf`-style syntax and performs bounds checking. It returns the number of characters in the formatted buffer.

For more information, see the documentation on the `printf` function for the runtime library of your compiler.

Syntax

```c
int util_snprintf(char *s, int n, char *fmt, ...);
```
Returns
The number of characters formatted into the buffer.

Parameters
char *s is the buffer to receive the formatted string.
int n is the maximum number of bytes allowed to be copied.
char *fmt is the format string. The function handles only %d and %s strings; it
does not handle any width or precision strings.
... represents a sequence of parameters for the printf function.

See Also
util_sprintf, util_vsnprintf, util_vsprintf

util_sprintf

The util_sprintf function formats a specified string, using a specified format,
into a specified buffer, using the printf-style syntax without bounds checking. It
returns the number of characters in the formatted buffer.

Because util_sprintf doesn’t perform bounds checking, use this function only if
you are certain that the string fits the buffer. Otherwise, use the function
util_snprintf. For more information, see the documentation on the printf
function for the runtime library of your compiler.

Syntax
int util_sprintf(char *s, char *fmt, ...);

Returns
The number of characters formatted into the buffer.

Parameters
char *s is the buffer to receive the formatted string.
char *fmt is the format string. The function handles only %d and %s strings; it
does not handle any width or precision strings.
... represents a sequence of parameters for the printf function.
Example
char *logmsg;
int len;
logmsg = (char *) MALLOC(256);
len = util_sprintf(logmsg, "%s %s %s\n", ip, method, uri);

See Also
util_snprintf, util_vsnprintf, util_vsprintf

util_strcasecmp

The `util_strcasecmp` function performs a comparison of two alphanumeric strings and returns a -1, 0, or 1 to signal which is larger or that they are identical. The comparison is not case-sensitive.

Syntax
```
int util_strcasecmp(const char *s1, const char *s2);
```

Returns
1 if `s1` is greater than `s2`.
0 if `s1` is equal to `s2`.
-1 if `s1` is less than `s2`.

Parameters
- `char *s1` is the first string.
- `char *s2` is the second string.

See Also
util_strncasecmp

util_strftime

The `util_strftime` function translates a `tm` structure, which is a structure describing a system time, into a textual representation. It is a thread-safe version of the standard `strftime` function.

Syntax
```
int util_strftime(char *s, const char *format, const struct tm *t);
```
**Returns**
The number of characters placed into \( s \), not counting the terminating NULL character.

**Parameters**
- `char *s` is the string buffer to put the text into. There is no bounds checking, so you must make sure that your buffer is large enough for the text of the date.
- `const char *format` is a format string, a bit like a `printf` string in that it consists of text with certain \%x substrings. You may use the constant `HTTP_DATE_FMT` to create date strings in the standard Internet format. For more information, see the documentation on the `printf` function for the runtime library of your compiler. Refer to Chapter 10, “Time Formats” for details on time formats.
- `const struct tm *t` is a pointer to a calendar time (tm) structure, usually created by the function `system_localtime` or `system_gmtime`.

**See Also**
- `system_localtime`, `system_gmtime`

#### `util_strncasecmp`

The `util_strncasecmp` function performs a comparison of the first \( n \) characters in the alphanumeric strings and returns a \(-1, 0, \) or \( 1 \) to signal which is larger or that they are identical.

The function’s comparison is not case-sensitive.

**Syntax**

```c
int util_strncasecmp(const char *s1, const char *s2, int n);
```

**Returns**
- \( 1 \) if \( s1 \) is greater than \( s2 \).
- \( 0 \) if \( s1 \) is equal to \( s2 \).
- \(-1 \) if \( s1 \) is less than \( s2 \).

**Parameters**
- `char *s1` is the first string.
- `char *s2` is the second string.
- `int n` is the number of initial characters to compare.
util_uri_escape

The `util_uri_escape` function converts any special characters in the URI into the URI format (`%XX`, where `XX` is the hexadecimal equivalent of the ASCII character), and returns the escaped string. The special characters are `?#:+*<>`, space, carriage return, and line feed.

Use `util_uri_escape` before sending a URI back to the client.

**Syntax**

```c
char *util_uri_escape(char *d, char *s);
```

**Returns**
The string (possibly newly allocated) with escaped characters replaced.

**Parameters**

- `char *d` is a string. If `d` is not NULL, the function copies the formatted string into `d` and returns it. If `d` is NULL, the function allocates a properly sized string and copies the formatted special characters into the new string, then returns it.

The `util_uri_escape` function does not check bounds for the parameter `d`. Therefore, if `d` is not NULL, it should be at least three times as large as the string `s`.

- `char *s` is the string containing the original unescaped URI.

**See Also**

`util_uri_is_evil`, `util_uri_parse`, `util_uri_unescape`

util_uri_is_evil

The `util_uri_is_evil` function checks a specified URI for insecure path characters. Insecure path characters include `//`, `/./`, `../` and `/../` (also for Windows `./`) at the end of the URI. Use this function to see if a URI requested by the client is insecure.

**Syntax**

```c
int util_uri_is_evil(char *t);
```
Returns
1 if the URI is insecure, or 0 if the URI is OK.

Parameters
char *t is the URI to be checked.

See Also
util_uri_escape, util_uri_parse

util_uri_parse
The util_uri_parse function converts //, /., and */../ into / in the specified URI (where * is any character other than /). You can use this function to convert a URI’s bad sequences into valid ones. First use the function util_uri_is_evil to determine whether the function has a bad sequence.

Syntax
void util_uri_parse(char *uri);

Returns
void

Parameters
char *uri is the URI to be converted.

See Also
util_uri_is_evil, util_uri_unescape

util_uri_unescape
The util_uri_unescape function converts the encoded characters of a URI into their ASCII equivalents. Encoded characters appear as %XX, where XX is a hexadecimal equivalent of the character.

NOTE
You cannot use an embedded null in a string, because NSAPI functions assume that a null is the end of the string. Therefore, passing unicode-encoded content through an NSAPI plugin doesn’t work.
**util_uri_unescape**

Syntax

```c
void util_uri_unescape(char *uri);
```

**Returns**

void

**Parameters**

char *uri is the URI to be converted.

**See Also**

util_uri_escape, util_uri_is_evil, util_uri_parse

**util_vsnprintf**

The `util_vsnprintf` function formats a specified string, using a specified format, into a specified buffer using the `vprintf`-style syntax and performs bounds checking. It returns the number of characters in the formatted buffer.

For more information, see the documentation on the `printf` function for the runtime library of your compiler.

**Syntax**

```c
int util_vsnprintf(char *s, int n, register char *fmt, va_list args);
```

**Returns**

The number of characters formatted into the buffer.

**Parameters**

char *s is the buffer to receive the formatted string.

int n is the maximum number of bytes allowed to be copied.

register char *fmt is the format string. The function handles only `%d` and `%s` strings; it does not handle any width or precision strings.

va_list args is an STD argument variable obtained from a previous call to `va_start`.

**See Also**

util_sprintf, util_vsprintf
util_vsprintf

The `util_vsprintf` function formats a specified string, using a specified format, into a specified buffer using the `vprintf`-style syntax without bounds checking. It returns the number of characters in the formatted buffer.

For more information, see the documentation on the `printf` function for the runtime library of your compiler.

Syntax

```c
int util_vsprintf(char *s, register char *fmt, va_list args);
```

Returns

The number of characters formatted into the buffer.

Parameters

- `char *s` is the buffer to receive the formatted string.
- `register char *fmt` is the format string. The function handles only `%d` and `%s` strings; it does not handle any width or precision strings.
- `va_list args` is an STD argument variable obtained from a previous call to `va_start`.

See Also

`util_snprintf`, `util_vsnprintf`

V

vs_alloc_slot

The `vs_alloc_slot` function allocates a new slot for storing a pointer to data specific to a certain `VirtualServer*`. The returned slot number may be used in subsequent `vs_set_data` and `vs_get_data` calls. The returned slot number is valid for any `VirtualServer*`.

The value of the pointer (which may be returned by a call to `vs_set_data`) defaults to NULL for every `VirtualServer*`.

Syntax

```c
int vs_alloc_slot(void);
```
vs_get_data

The vs_get_data function finds the value of a pointer to data for a given VirtualServer* and slot. The slot must be a slot number returned from vs_alloc_slot or vs_set_data.

Syntax

void* vs_get_data(const VirtualServer* vs, int slot);

Returns

The value of the pointer previously stored via vs_set_data, or NULL on failure.

Parameters

const VirtualServer* vs represents the virtual server to query the pointer for.

int slot is the slot number to retrieve the pointer from.

See Also

vs_set_data, vs_alloc_slot

vs_get_default_httpd_object

The vs_get_default_httpd_object function obtains a pointer to the default (or root) httpd_object from the virtual server's httpd_objset (in the configuration defined by the obj.conf file of the virtual server class). The default object is typically named default. Plugins may only modify the httpd_object at VSInitFunc time (see vs_register_cb for an explanation of VSInitFunc time).

Do not FREE the returned object.

Syntax

httpd_object* vs_get_default_httpd_object(VirtualServer* vs);

Returns

A pointer the default httpd_object, or NULL on failure. Do not FREE this object.
Parameters

VirtualServer* vs represents the virtual server for which to find the default object.

See Also

vs_get_httpd_objset, vs_register_cb

**vs_get_doc_root**

The `vs_get_doc_root` function finds the document root for a virtual server. The returned string is the full operating system path to the document root.

The caller should FREE the returned string when done with it.

Syntax

```
char* vs_get_doc_root(const VirtualServer* vs);
```

Returns

A pointer to a string representing the full operating system path to the document root. It is the caller’s responsibility to FREE this string.

Parameters

const VirtualServer* vs represents the virtual server for which to find the document root.

**vs_get_httpd_objset**

The `vs_get_httpd_objset` function obtains a pointer to the `httpd_objset` (the configuration defined by the `obj.conf` file of the virtual server class) for a given virtual server. Plugins may only modify the `httpd_objset` at `VSInitFunc` time (see `vs_register_cb` for an explanation of `VSInitFunc` time).

Do not FREE the returned objset.

Syntax

```
httpd_objset* vs_get_httpd_objset(VirtualServer* vs);
```

Returns

A pointer to the `httpd_objset`, or NULL on failure. Do not FREE this objset.

Parameters

VirtualServer* vs represents the virtual server for which to find the objset.
See Also
vs_get_default_httpd_object, vs_register_cb

vs_get_id

The vs_get_id function finds the ID of a VirtualServer*.

The ID of a virtual server is a unique null-terminated string that remains constant across configurations. Note that while IDs remain constant across configurations, the value of VirtualServer* pointers do not.

Do not FREE the virtual server ID string. If called during request processing, the string will remain valid for the duration of the current request. If called during VSInitFunc processing, the string will remain valid until after the corresponding VSDestroyFunc function has returned (see vs_register_cb).

To retrieve a VirtualServer* that is valid only for the current request, use request_get_vs.

Syntax
const char* vs_get_id(const VirtualServer* vs);

Returns
A pointer to a string representing the virtual server ID. Do not FREE this string.

Parameters
const VirtualServer* vs represents the virtual server of interest.

See Also
vs_register_cb, request_get_vs

vs_get_mime_type

The vs_get_mime_type function determines the MIME type that would be returned in the Content-Type: header for the given URI.

The caller should FREE the returned string when done with it.

Syntax
char* vs_get_mime_type(const VirtualServer* vs, const char* uri);
NSAPI Functions (in Alphabetical Order)

Returns
A pointer to a string representing the MIME type. It is the caller's responsibility to FREE this string.

Parameters
const VirtualServer* vs represents the virtual server of interest.
const char* uri is the URI whose MIME type is of interest.

vs_lookup_config_var
The vs_lookup_config_var function finds the value of a configuration variable for a given virtual server.
Do not FREE the returned string.

Syntax
const char* vs_lookup_config_var(const VirtualServer* vs, const char* name);

Returns
A pointer to a string representing the value of variable name on success, or NULL if variable name was not found. Do not FREE this string.

Parameters
const VirtualServer* vs represents the virtual server of interest.
const char* name is the name of the configuration variable.

vs_register_cb
The vs_register_cb function allows a plugin to register functions that will receive notifications of virtual server initialization and destruction events. The vs_register_cb function would typically be called from an Init SAF in magnus.conf.

When a new configuration is loaded, all registered VSInitFunc (virtual server initialization) callbacks are called for each of the virtual servers before any requests are served from the new configuration. VSInitFunc callbacks are called in the same order they were registered; that is, the first callback registered is the first called.
When the last request has been served from an old configuration, all registered VSDestroyFunc (virtual server destruction) callbacks are called for each of the virtual servers before any virtual servers are destroyed. VSDestroyFunc callbacks are called in reverse order; that is, the first callback registered is the last called.

Either initfn or destroyfn may be NULL if the caller is not interested in callbacks for initialization or destruction, respectively.

**Syntax**

```c
int vs_register_cb(VSInitFunc* initfn, VSDestroyFunc* destroyfn);
```

**Returns**

The constant `REQ_PROCEED` if the operation succeeded.

The constant `REQ_ABORTED` if the operation failed.

**Parameters**

VSInitFunc* initfn is a pointer to the function to call at virtual server initialization time, or NULL if the caller is not interested in virtual server initialization events.

VSDestroyFunc* destroyfn is a pointer to the function to call at virtual server destruction time, or NULL if the caller is not interested in virtual server destruction events.

---

### vs_set_data

The vs_set_data function sets the value of a pointer to data for a given virtual server and slot. The *slot must be -1 or a slot number returned from vs_alloc_slot. If *slot is -1, vs_set_data calls vs_alloc_slot implicitly and returns the new slot number in *slot.

Note that the stored pointer is maintained on a per-VirtualServer* basis, not a per-ID basis. Distinct VirtualServer*s from different configurations may exist simultaneously with the same virtual server IDs. However, since these are distinct VirtualServer*s, they each have their own VirtualServer* specific data. As a result, vs_set_data should generally not be called outside of VSInitFunc processing (see vs_register_cb for an explanation of VSInitFunc processing).

**Syntax**

```c
void* vs_set_data(const VirtualServer* vs, int* slot, void* data);
```

**Returns**

Data on success, or NULL on failure.
**Parameters**

- `const VirtualServer* vs` represents the virtual server to set the pointer for.
- `int* slot` is the slot number to store the pointer at.
- `void* data` is the pointer to store.

**See Also**

- `vs_get_data`, `vs_alloc_slot`, `vs_register_cb`

### vs_translate_uri

The `vs_translate_uri` function translates a URI as though it were part of a request for a specific virtual server. The returned string is the full operating system path.

The caller should FREE the returned string when done with it.

**Syntax**

```c
char* vs_translate_uri(const VirtualServer* vs, const char* uri);
```

**Returns**

A pointer to a string representing the full operating system path for the given URI. It is the caller’s responsibility to FREE this string.

**Parameters**

- `const VirtualServer* vs` represents the virtual server for which to translate the URI.
- `const char* uri` is the URI to translate to an operating system path.

### write

The `write` filter method is called when output data is to be sent. Filters that modify or consume outgoing data should implement the `write` filter method.

Upon receiving control, a write implementation should first process the data as necessary, and then pass it on to the next filter layer; for example, by calling `net_write(layer->lower, ..., )`. If the filter buffers outgoing data, it should implement the `flush` filter method.

**Syntax**

```c
int write(FilterLayer *layer, const void *buf, int amount);
```
writev

The `writev` filter method is called when multiple buffers of output data are to be sent. Filters that modify or consume outgoing data may choose to implement the `writev` filter method.

If a filter implements the `write` filter method but not the `writev` filter method, the server automatically translates `net_writev` calls to `net_write` calls. As a result, filters interested in the outgoing data stream do not need to implement the `writev` filter method. However, for performance reasons, it is beneficial for filters that implement the `write` filter method to also implement the `writev` filter method.

Syntax

```c
int writev(FilterLayer *layer, const struct iovec *iov, int iov_size);
```

Returns

The number of bytes consumed, which may be less than the requested amount if an error occurred.

Parameters

`FilterLayer *layer` is the filter layer the filter is installed in.
const struct iovec *iov is an array of iovec structures, each of which contains outgoing data.

int iov_size is the number of iovec structures in the iov array.

**Example**

```c
int myfilter_writev(FilterLayer *layer, const struct iovec *iov, int iov_size)
{
    return net_writev(layer->lower, iov, iov_size);
}
```

**See Also**

flush, net_write, write
NSAPI uses many data structures that are defined in the nsapi.h header file, which is in the directory server-root/plugins/include.

The NSAPI functions described in Chapter 7, “NSAPI Function Reference” provide access to most of the data structures and data fields. Before directly accessing a data structure in nsapi.h, check to see if an accessor function exists for it.

For information about the privatization of some data structures in Sun ONE Web Server 4.x, see “Privatization of Some Data Structures” on page 252.

The rest of this chapter describes public data structures in nsapi.h. Note that data structures in nsapi.h that are not described in this chapter are considered private and may change incompatibly in future releases.

This chapter has the following sections:

- Session
- pblock
- pb_entry
- pb_param
- Session->client
- Request
- stat
- shmem_s
- cinfo
- sendfiledata
- Filter
Privatization of Some Data Structures

- FilterContext
- FilterLayer
- FilterMethods

Privatization of Some Data Structures

In Sun ONE Web Server 4.x, some data structures were moved from nsapi.h to nsapi_pvt.h. The data structures in nsapi_pvt.h are now considered to be private data structures, and you should not write code that accesses them directly. Instead, use accessor functions. We expect that very few people have written plugins that access these data structures directly, so this change should have very little impact on customer-defined plugins. Look in nsapi_pvt.h to see which data structures have been removed from the public domain, and to see the accessor functions you can use to access them from now on.

Plugins written for Enterprise Server 3.x that access contents of data structures defined in nsapi_pvt.h will not be source compatible with Sun ONE Web Server 4.x and 6.x, that is, it will be necessary to include "nsapi_pvt.h" to build such plugins from source. There is also a small chance that these programs will not be binary compatible with Sun ONE Web Server 4.x and 6.x, because some of the data structures in nsapi_pvt.h have changed size. In particular, the directive structure is larger, which means that a plugin that indexes through the directives in a dtable will not work without being rebuilt (with nsapi_pvt.h included).

We hope that the majority of plugins do not reference the internals of data structures in nsapi_pvt.h, and therefore that most existing NSAPI plugins will be both binary and source compatible with Sun ONE Web Server 6.1.
Session

A session is the time between the opening and closing of the connection between the client and the server. The session data structure holds variables that apply session wide, regardless of the requests being sent, as shown here:

```c
typedef struct {
   /* Information about the remote client */
   pblock *client;

   /* The socket descriptor to the remote client */
   SYS_NETFD csd;

   /* The input buffer for that socket descriptor */
   netbuf *inbuf;

   /* Raw socket information about the remote */
   /* client (for internal use) */
   struct in_addr iaddr;
} Session;
```

pbblock

The parameter block is the hash table that holds pb_entry structures. Its contents are transparent to most code. This data structure is frequently used in NSAPI; it provides the basic mechanism for packaging up parameters and values. There are many functions for creating and managing parameter blocks, and for extracting, adding, and deleting entries. See the functions whose names start with pblock in Chapter 7, “NSAPI Function Reference.” You should not need to write code that accesses pblock data fields directly.

```c
typedef struct {
   int hsize;
   struct pb_entry **ht;
} pbblock;
```
### pb_entry

The `pb_entry` is a single element in the parameter block.

```c
struct pb_entry {
    pb_param *param;
    struct pb_entry *next;
};
```

### pb_param

The `pb_param` represents a name-value pair, as stored in a `pb_entry`.

```c
typedef struct {
    char *name,*value;
} pb_param;
```

### Session->client

The `Session->client` parameter block structure contains two entries:

- The `ip` entry is the IP address of the client machine.
- The `dns` entry is the DNS name of the remote machine. This member must be accessed through the `session_dns` function call:

```c
/*
 * session_dns returns the DNS host name of the client for this
 * session and inserts it into the client pbblock. Returns NULL if
 * unavailable.
 */
char *session_dns(Session *sn);
```
Under HTTP protocol, there is only one request per session. The request structure contains the variables that apply to the request in that session (for example, the variables include the client’s HTTP headers).

```
typedef struct {
    /* Server working variables */
    pblock *vars;

    /* The method, URI, and protocol revision of this request */
    block *reqpb;

    /* Protocol specific headers */
    int loadhdrs;
    pbloc* headers;

    /* Server’s response headers */
    int senthdrs;
    pbloc* srvhdrs;

    /* The object set constructed to fulfill this request */
    httpd_objset *os;
} Request;
```
When a program calls the `stat()` function for a given file, the system returns a structure that provides information about the file. The specific details of the structure should be obtained from your platform’s implementation, but the basic outline of the structure is as follows:

```c
struct stat {
    dev_t st_dev; /* device of inode */
    inot_t st_ino; /* inode number */
    short st_mode; /* mode bits */
    short st_nlink; /* number of links to file */
    short st_uid; /* owner’s user id */
    short st_gid; /* owner’s group id */
    dev_t st_rdev; /* for special files */
    off_t st_size; /* file size in characters */
    time_t st_atime; /* time last accessed */
    time_t st_mtime; /* time last modified */
    time_t st_ctime; /* time inode last changed*/
}
```

The elements that are most significant for server plugin API activities are `st_size`, `st_atime`, `st_mtime`, and `st_ctime`.

```c
typedef struct {
    void *data; /* the data */
    HANDLE fdmap;
    int size; /* the maximum length of the data */
    char *name; /* internal use: filename to unlink if exposed */
    SYS_FILE fd; /* internal use: file descriptor for region */
} shmem_s;
```
The `cinfo` data structure records the content information for a file.

```
typedef struct {
    char *type; /* Identifies what kind of data is in the file*/
    char *encoding; /* encoding identifies any compression or other */
    /* content-independent transformation that's been */
    /* applied to the file, such as uuencode*/
    char *language; /* Identifies the language a text document is in. */
} cinfo;
```

The `sendfiledata` data structure is used to pass parameters to the `net_sendfile` function. It is also passed to the `sendfile` method in an installed filter in response to a `net_sendfile` call.

```
typedef struct {
    SYS_FILE fd; /* file to send */
    size_t offset; /* offset in file to start sending from */
    size_t len; /* number of bytes to send from file */
    const void *header; /* data to send before file */
    int hlen; /* number of bytes to send before file */
    const void *trailer; /* data to send after file */
    int tlen; /* number of bytes to send after file */
} sendfiledata;
```
Filter

The Filter data structure is an opaque representation of a filter. A Filter structure is created by calling filter_create.

```c
typedef struct Filter Filter;
```

FilterContext

The FilterContext data structure stores context associated with a particular filter layer. Filter layers are created by calling filter_insert.

Filter developers may use the data member to store filter-specific context information.

```c
typedef struct {
    pool_handle_t *pool; /* pool context was allocated from */
    Session *sn;         /* session being processed */
    Request *rq;         /* request being processed */
    void *data;          /* filter-defined private data */
} FilterContext;
```

FilterLayer

The FilterLayer data structure represents one layer in a filter stack. The FilterLayer structure identifies the filter installed at that layer and provides pointers to layer-specific context and a filter stack that represents the layer immediately below it in the filter stack.

```c
typedef struct {
    Filter *filter; /* the filter at this layer in the filter stack */
    FilterContext *context; /* context for the filter */
    SYS_NETFD lower; /* access to the next filter layer in the stack */
} FilterLayer;
```
FilterMethods

The FilterMethods data structure is passed to filter_create to define the filter methods a filter supports. Each new FilterMethods instance must be initialized with the FILTER_METHODS_INITIALIZER macro. For each filter method a filter supports, the corresponding FilterMethods member should point to a function that implements that filter method.

typedef struct {
    size_t size;
    FilterInsertFunc *insert;
    FilterRemoveFunc *remove;
    FilterFlushFunc *flush;
    FilterReadFunc *read;
    FilterWriteFunc *write;
    FilterWritevFunc *writev;
    FilterSendfileFunc *sendfile;
} FilterMethods;
Chapter 9

Using Wildcard Patterns

This chapter describes the format of wildcard patterns used by the Sun ONE Web Server. These wildcards are used in:

- Directives in the configuration file `obj.conf` (see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference for detailed information about `obj.conf`).
- Various built-in SAFs (see the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference for more information about these predefined SAFs).
- Some NSAPI functions (see Chapter 2, “SAFs in the magnus.conf File”).

Wildcard patterns use special characters. If you want to use one of these characters without the special meaning, precede it with a backslash (\) character.

This chapter has the following sections:

- Wildcard Patterns
- Wildcard Examples

### Wildcard Patterns

The following table describes wildcard patterns, listing the pattern and its use.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Match zero or more characters.</td>
</tr>
<tr>
<td>?</td>
<td>Match exactly one occurrence of any character.</td>
</tr>
</tbody>
</table>
The following table provides wildcard examples, listing the pattern and the result.

### Table 9-2  Wildcard Examples

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>* .netscape.com</td>
<td>Matches any string ending with the characters .netscape.com.</td>
</tr>
<tr>
<td>(quark</td>
<td>energy).netscape.com</td>
</tr>
<tr>
<td>198 .93 .9 [23] .???</td>
<td>Matches a numeric string starting with either 198 .93 .92 or 198 .93 .93 and ending with any 3 characters.</td>
</tr>
<tr>
<td>*. *</td>
<td>Matches any string with a period in it.</td>
</tr>
<tr>
<td>* - netscape - *</td>
<td>Matches any string except those starting with netscape -.</td>
</tr>
<tr>
<td>* . netscape . com - quark . netscap e . com</td>
<td>Matches any host from domain . netscape . com except for a single host quark . netscape . com.</td>
</tr>
</tbody>
</table>
## Wildcard Examples

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>.com-</em>.netscape.com</td>
<td>Matches any host from domain .com except for hosts from subdomain netscape.com.</td>
</tr>
<tr>
<td>type=-magnus-internal/*</td>
<td>Matches any type that does not start with magnus-internal/.</td>
</tr>
<tr>
<td></td>
<td>This wildcard pattern is used in the file obj.conf in the catch-all Service directive.</td>
</tr>
</tbody>
</table>
This chapter describes the format strings used for dates and times. These formats are used by the NSAPI function `util_strftime`, by some built-in SAFs such as `append-trailer`, and by server-parsed HTML (parse-html). The formats are similar to those used by the `strftime` C library routine, but not identical.

The following table describes the formats, listing the symbols and their meanings.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Abbreviated weekday name (3 chars)</td>
</tr>
<tr>
<td>%d</td>
<td>Day of month as decimal number (01-31)</td>
</tr>
<tr>
<td>%S</td>
<td>Second as decimal number (00-59)</td>
</tr>
<tr>
<td>%M</td>
<td>Minute as decimal number (00-59)</td>
</tr>
<tr>
<td>%H</td>
<td>Hour in 24-hour format (00-23)</td>
</tr>
<tr>
<td>%Y</td>
<td>Year with century, as decimal number, up to 2099</td>
</tr>
<tr>
<td>%b</td>
<td>Abbreviated month name (3 chars)</td>
</tr>
<tr>
<td>%h</td>
<td>Abbreviated month name (3 chars)</td>
</tr>
<tr>
<td>%T</td>
<td>Time &quot;HH:MM:SS&quot;</td>
</tr>
<tr>
<td>%X</td>
<td>Time &quot;HH:MM:SS&quot;</td>
</tr>
<tr>
<td>%A</td>
<td>Full weekday name</td>
</tr>
<tr>
<td>%B</td>
<td>Full month name</td>
</tr>
<tr>
<td>%C</td>
<td>&quot;%a %b %e %H:%M:%S %Y&quot;</td>
</tr>
<tr>
<td>%c</td>
<td>Date &amp; time &quot;%m/%d/%y %H:%M:%S&quot;</td>
</tr>
<tr>
<td>%D</td>
<td>Date &quot;%m/%d/%y&quot;</td>
</tr>
</tbody>
</table>
### Table 10-1  Time Formats

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%e</td>
<td>Day of month as decimal number (1-31) without leading zeros</td>
</tr>
<tr>
<td>%I</td>
<td>Hour in 12-hour format (01-12)</td>
</tr>
<tr>
<td>%j</td>
<td>Day of year as decimal number (001-366)</td>
</tr>
<tr>
<td>%k</td>
<td>Hour in 24-hour format (0-23) without leading zeros</td>
</tr>
<tr>
<td>%l</td>
<td>Hour in 12-hour format (1-12) without leading zeros</td>
</tr>
<tr>
<td>%m</td>
<td>Month as decimal number (01-12)</td>
</tr>
<tr>
<td>%n</td>
<td>line feed</td>
</tr>
<tr>
<td>%p</td>
<td>A.M./P.M. indicator for 12-hour clock</td>
</tr>
<tr>
<td>%R</td>
<td>Time &quot;%H:%M&quot;</td>
</tr>
<tr>
<td>%r</td>
<td>Time &quot;%I:%M:%S %p&quot;</td>
</tr>
<tr>
<td>%t</td>
<td>tab</td>
</tr>
<tr>
<td>%U</td>
<td>Week of year as decimal number, with Sunday as first day of week (00-51)</td>
</tr>
<tr>
<td>%W</td>
<td>Week of year as decimal number, with Monday as first day of week (00-51)</td>
</tr>
<tr>
<td>%x</td>
<td>Date &quot;%m/%d/%y&quot;</td>
</tr>
<tr>
<td>%y</td>
<td>Year without century, as decimal number (00-99)</td>
</tr>
<tr>
<td>%%</td>
<td>Percent sign</td>
</tr>
</tbody>
</table>
The functions described in this chapter allow you to write a results caching plugin for Sun ONE Web Server. A results caching plugin, which is a Service SAF, caches data, a page, or part of a page in the web server address space, which the web server can refresh periodically on demand. An Init SAF initializes the callback function that performs the refresh.

A results caching plugin can generate a page for a request in three parts:

- A header, such as a page banner, which changes for every request
- A body, which changes less frequently
- A footer, which also changes for every request

Without this feature, a plugin would have to generate the whole page for every request (unless an IFRAME is used, where the header or footer is sent in the first response along with an IFRAME pointing to the body; in this case the browser must send another request for the IFRAME).

If the body of a page has not changed, the plugin needs to generate only the header and footer and to call the dr_net_write function (instead of net_write) with the following arguments:

- header
- footer
- handle to cache
- key to identify the cached object

The web server constructs the whole page by fetching the body from the cache. If the cache has expired, it calls the refresh function and sends the refreshed page back to the client.
An Init SAF that is visible to the plugin creates the handle to the cache. The Init SAF must pass the following parameters to the `dr_cache_init` function:

- `RefreshFunctionPointer`
- `FreeFunctionPointer`
- `KeyComparatorFunctionPtr`
- `RefreshInterval`

The `RefreshInterval` value must be a `PrIntervalTime` type. For more information, see the NSPR reference at:


As an alternative, if the body is a file that is present in a directory within the web server system machine, the plugin can generate the header and footer and call the `fc_net_write` function along with the file name.

This chapter lists the most important functions a results caching plugin can use. For more information, see the following file:

`server_root/plugins/include/drnsapi.h`

This chapter has the following sections:

- `dr_cache_destroy`
- `dr_cache_init`
- `dr_cache_refresh`
- `dr_net_write`
- `fc_net_write`

**dr_cache_destroy**

The `dr_cache_destroy` function destroys and frees resources associated with a previously created and used cache handle. This handle can no longer be used in subsequent calls to any of the above functions unless another `dr_cache_init` is performed.

**Syntax**

```c
void dr_cache_destroy(DrHdl *hdl);
```
Parameters
DrHdl *hdl is a pointer to a previously initialized handle to a cache (see dr_cache_init).

Returns
void

Example
dr_cache_destroy(&myHdl);

dr_cache_init

The dr_cache_init function creates a persistent handle to the cache, or NULL on failure. It is called by an Init SAF.

Syntax
PRInt32 dr_cache_init(DrHdl *hdl, RefreshFunc_t ref, FreeFunc_t fre, CompareFunc_t cmp, PRUint32 maxEntries, PRIntervalTime maxAge);

Returns
1 if successful.
0 if an error occurs.

Parameters
The following table describes parameters for the dr_cache_init function.

Table 11-1   dr_cache_init parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrHdl hdl</td>
<td>Pointer to an unallocated handle.</td>
</tr>
<tr>
<td>RefreshFunc_t ref</td>
<td>Pointer to a cache refresh function. This can be NULL; see the DR_CHECK flag and DR_EXPIR return value for dr_net_write.</td>
</tr>
<tr>
<td>FreeFunc_t fre</td>
<td>Pointer to a function that frees an entry.</td>
</tr>
<tr>
<td>CompareFunc_t cmp</td>
<td>Pointer to a key comparator function.</td>
</tr>
<tr>
<td>PRUint32 maxEntries</td>
<td>Maximum number of entries possible in the cache for a given hdl.</td>
</tr>
<tr>
<td>PRIntervalTime maxAge</td>
<td>The maximum amount of time that an entry is valid. If 0, the cache never expires.</td>
</tr>
</tbody>
</table>
dr_cache_refresh

Example

if (!dr_cache_init(&hdl, (RefreshFunc_t)FnRefresh,
   (FreeFunc_t)FnFree, (CompareFunc_t)FnCompare, 150000,
   PR_SecondsToInterval(7200)))
{
    ereport(LOG_FAILURE, "dr_cache_init() failed");
    return(REQ_ABORTED);
}

dr_cache_refresh

The dr_cache_refresh function provides a way of refreshing a cache entry when
the plugin requires it. This can be achieved by passing NULL for the ref parameter
in dr_cache_init and by passing DR_CHECK in a dr_net_write call. If DR_CHECK is
passed to dr_net_write and it returns with DR_EXPIR, the plugin should generate
new content in the entry and call dr_cache_refresh with that entry before calling
dr_net_write again to send the response.

The plugin may simply decide to replace the cached entry even if it has not expired
(based on some other business logic). The dr_cache_refresh function is useful in
this case. This way the plugin does the cache refresh management actively by itself.

Syntax

PRInt32 dr_cache_refresh(DrHdl hdl, const char *key, PRUint32 klen,
   PRIntervalTime timeout, Entry *entry, Request *rq, Session *sn);

Returns

1 if successful.

0 if an error occurs.

Parameters

The following table describes parameters for the dr_cache_refresh function.

Table 11-2 dr_cache_refresh parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrHdl</td>
<td>Persistent handle created by the dr_cache_init function.</td>
</tr>
<tr>
<td>const char *key</td>
<td>Key to cache, search, or refresh.</td>
</tr>
<tr>
<td>PRUint32</td>
<td>Length of the key in bytes.</td>
</tr>
</tbody>
</table>
Example

```c
Entry entry;
char *key = "MOVIES"
GenNewMovieList(&entry.data, &entry.dataLen); // Implemented by
// plugin developer
if(!dr_cache_refresh(hdl, key, strlen(key), 0, &entry, rq, sn))
{
    ereport(LOG_FAILURE, "dr_cache_refresh() failed");
    return REQ_ABORTED;
}
```

---

### dr_cache_refresh parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIntervalTime</td>
<td>Expiration time of this entry; if a value of 0 is passed, the maxAge value passed to dr_cache_init is used.</td>
</tr>
<tr>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>Entry *entry</td>
<td>The not NULL entry to be cached.</td>
</tr>
<tr>
<td>Request *rq</td>
<td>Pointer to the request.</td>
</tr>
<tr>
<td>Session *sn</td>
<td>Pointer to the session.</td>
</tr>
</tbody>
</table>

---

**dr_net_write**

The `dr_net_write` function sends a response back to the requestor after constructing the full page with `hdr`, the content of the cached entry as the body (located using the `key`), and `ftr`. The `hdr`, `ftr`, or `hdl` can be NULL, but not all of them can be NULL. If `hdl` is NULL, no cache lookup is done; the caller must pass `DR_NONE` as the flag.

By default, this function refreshes the cache entry if it has expired by making a call to the `ref` function passed to `dr_cache_init`. If no cache entry is found with the specified `key`, this function adds a new cache entry by calling the `ref` function before sending out the response. However, if the `DR_CHECK` flag is passed in the `flags` parameter and if either the cache entry has expired or the cache entry corresponding to the `key` does not exist, `dr_net_write` does not send any data out. Instead it returns with `DR_EXPIR`.

If `ref` (passed to `dr_cache_init`) is NULL, the `DR_CHECK` flag is not passed in the `flags` parameter, and the cache entry corresponding to the `key` has expired or does not exist, then `dr_net_write` fails with `DR_ERROR`. However, `dr_net_write` refreshes the cache if `ref` is not NULL and `DR_CHECK` is not passed.

---

Table 11-2  dr_cache_refresh parameters
If ref (passed to dr_cache_init) is NULL and the DR_CHECK flag is not passed but DR_IGNORE is passed and the entry is present in the cache, dr_net_write sends out the response even if the entry has expired. However, if the entry is not found, dr_net_write returns DR_ERROR.

If ref (passed to dr_cache_init) is not NULL and the DR_CHECK flag is not passed but DR_IGNORE is passed and the entry is present in the cache, dr_net_write sends out the response even if the entry has expired. However, if the entry is not found, dr_net_write calls the ref function and stores the new entry returned from ref before sending out the response.

**Syntax**

```c
PRInt32 dr_net_write(DrHdl hdl, const char *key, PRUint32 klen,
const char *hdr, const char *ftr, PRUint32 hlen, PRUint32 flen,
PRIntervalTime timeout, PRUint32 flags, Request *rq, Session *sn);
```

**Returns**

- IO_OKAY if successful.
- IO_ERROR if an error occurs.
- DR_ERROR if an error in cache handling occurs.
- DR_EXPIR if the cache has expired.

**Parameters**

The following table describes parameters for the `dr_net_write` function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrHdl hdl</td>
<td>Persistent handle created by the <code>dr_cache_init</code> function.</td>
</tr>
<tr>
<td>const char *key</td>
<td>Key to cache, search, or refresh.</td>
</tr>
<tr>
<td>PRUint32 klen</td>
<td>Length of the key in bytes.</td>
</tr>
<tr>
<td>const char *hdr</td>
<td>Any header data (which can be NULL).</td>
</tr>
<tr>
<td>const char *ftr</td>
<td>Any footer data (which can be NULL).</td>
</tr>
<tr>
<td>PRUint32 hlen</td>
<td>Length of the header data in bytes (which can be 0).</td>
</tr>
<tr>
<td>PRUint32 flen</td>
<td>Length of the footer data in bytes (which can be 0).</td>
</tr>
<tr>
<td>PRIntervalTime</td>
<td>Timeout before this function aborts.</td>
</tr>
</tbody>
</table>
Table 11-3  dr_net_write parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRUint32 flags</td>
<td>ORed directives for this function (see the Flags table, below).</td>
</tr>
<tr>
<td>Request *rq</td>
<td>Pointer to the request.</td>
</tr>
<tr>
<td>Session *sn</td>
<td>Pointer to the session.</td>
</tr>
</tbody>
</table>

Flags

The following table describes flags for dr_net_write.

Table 11-4  Flags for dr_net_write

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DR_NONE</td>
<td>Specifies that no cache is used, so the function works as net_write does; DrHdl can be NULL.</td>
</tr>
<tr>
<td>DR_FORCE</td>
<td>Forces the cache to refresh, even if it has not expired.</td>
</tr>
<tr>
<td>DR_CHECK</td>
<td>Returns DR_EXPIR if the cache has expired; if the calling function has not provided a refresh function and this flag is not used, DR_ERROR is returned.</td>
</tr>
<tr>
<td>DR_IGNORE</td>
<td>Ignores cache expiration and sends out the cache entry even if it has expired.</td>
</tr>
<tr>
<td>DR_CNTLEN</td>
<td>Supplies the Content-Length header and does a PROTOCOL_START_RESPONSE.</td>
</tr>
<tr>
<td>DR_PROTO</td>
<td>Does a PROTOCOL_START_RESPONSE.</td>
</tr>
</tbody>
</table>

Example

```c
if(dr_net_write(Dr, szFileName, iLenK, NULL, NULL, 0, 0, 0, DR_CNTLEN | DR_PROTO, rq, sn) == IO_ERROR)
{
    return(REQ_EXIT);
}
```
fc_net_write

The fc_net_write function is used to send a header and/or footer and a file that exists somewhere in the system. The fileName should be the full path name of a file.

Syntax

PRInt32 fc_net_write(const char *fileName, const char *hdr, const char *ftr, PRUint32 hlen, PRUint32 flen, PRUint32 flags, PRIntervalTime timeout, Session *sn, Request *rq);

Returns

IO_OKAY if successful.

IO_ERROR if an error occurs.

FC_ERROR if an error in file handling occurs.

Parameters

The following table describes parameters for the fc_net_write function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>const char *fileName</td>
<td>File to be inserted.</td>
</tr>
<tr>
<td>const char *hdr</td>
<td>Any header data (which can be NULL).</td>
</tr>
<tr>
<td>const char *ftr</td>
<td>Any footer data (which can be NULL).</td>
</tr>
<tr>
<td>PRUint32 hlen</td>
<td>Length of the header data in bytes (which can be 0).</td>
</tr>
<tr>
<td>PRUint32 flen</td>
<td>Length of the footer data in bytes (which can be 0).</td>
</tr>
<tr>
<td>PRUint32 flags</td>
<td>ORed directives for this function (see the Flags table, below).</td>
</tr>
<tr>
<td>PRIntervalTime timeout</td>
<td>Timeout before this function aborts.</td>
</tr>
<tr>
<td>Request *rq</td>
<td>Pointer to the request.</td>
</tr>
<tr>
<td>Session *sn</td>
<td>Pointer to the session.</td>
</tr>
</tbody>
</table>
**Flags**

The following table describes flags for fc_net_write.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC_CNTLEN</td>
<td>supplies the Content-Length header and does a PROTOCOL_START_RESPONSE.</td>
</tr>
<tr>
<td>FC_PROTO</td>
<td>does a PROTOCOL_START_RESPONSE.</td>
</tr>
</tbody>
</table>

**Example**

```c
const char *fileName = "/docs/myads/file1.ad";
char *hdr = GenHdr(); // Implemented by plugin
char *ftr = GenFtr(); // Implemented by plugin

if(fc_net_write(fileName, hdr, ftr, strlen(hdr), strlen(ftr),
                FC_CNTLEN, PR_INTERVAL_NO_TIMEOUT, sn, rq) != IO_OKEY)
{
    ereport(LOG_FAILURE, "fc_net_write() failed");
    return REQ_ABORTED;
}
```
fc_net_write
Chapter 12

Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is a protocol (a set of rules that describes how information is exchanged) that allows a client (such as a web browser) and a web server to communicate with each other.

HTTP is based on a request-response model. The browser opens a connection to the server and sends a request to the server. The server processes the request and generates a response, which it sends to the browser. The server then closes the connection.

This chapter provides a short introduction to a few HTTP basics. For more information on HTTP, see the IETF home page at:

http://www.ietf.org/home.html

This chapter has the following sections:

- Compliance
- Requests
- Responses
- Buffered Streams

Compliance

Sun ONE Web Server 6.1 supports HTTP/1.1. Previous versions of the server supported HTTP/1.0. The server is conditionally compliant with the HTTP/1.1 proposed standard, as approved by the Internet Engineering Steering Group (IESG), and the Internet Engineering Task Force (IETF) HTTP working group.
Requests

A request from a browser to a server includes the following information:

- Request Method, URI, and Protocol Version
- Request Headers
- Request Data

Request Method, URI, and Protocol Version

A browser can request information using a number of methods. The commonly used methods include the following:

- **GET** -- Requests the specified resource (such as a document or image)
- **HEAD** -- Requests only the header information for the document
- **POST** -- Requests that the server accept some data from the browser, such as form input for a CGI program
- **PUT** -- Replaces the contents of a server’s document with data from the browser

Request Headers

The browser can send headers to the server. Most are optional.

The following table lists some of the commonly used request headers.

<table>
<thead>
<tr>
<th>Request Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept</td>
<td>File types the browser can accept.</td>
</tr>
<tr>
<td>Authorization</td>
<td>Used if the browser wants to authenticate itself with a server; information such as the user name and password are included.</td>
</tr>
</tbody>
</table>
Request Data

If the browser has made a POST or PUT request, it sends data after the blank line following the request headers. If the browser sends a GET or HEAD request, there is no data to send.

Responses

The server’s response includes the following:

- HTTP Protocol Version, Status Code, and Reason Phrase
- Response Headers
- Response Data

HTTP Protocol Version, Status Code, and Reason Phrase

The server sends back a status code, which is a three-digit numeric code. The five categories of status codes are:

- 100-199 a provisional response.
- 200-299 a successful transaction.
- 300-399 the requested resource should be retrieved from a different location.
- 400-499 an error was caused by the browser.
- 500-599 a serious error occurred in the server.
The following table lists some common status codes.

**Table 12-2 Common HTTP Status Codes**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK; request has succeeded for the method used (GET, POST, HEAD).</td>
</tr>
<tr>
<td>201</td>
<td>The request has resulted in the creation of a new resource reference by the returned URI.</td>
</tr>
<tr>
<td>206</td>
<td>The server has sent a response to byte range requests.</td>
</tr>
<tr>
<td>302</td>
<td>Found. Redirection to a new URL. The original URL has moved. This is not an error; most browsers will get the new page.</td>
</tr>
<tr>
<td>304</td>
<td>Use a local copy. If a browser already has a page in its cache, and the page is requested again, some browsers (such as Netscape Navigator) relay to the web server the “last-modified” timestamp on the browser’s cached copy. If the copy on the server is not newer than the browser’s copy, the server returns a 304 code instead of returning the page, reducing unnecessary network traffic. This is not an error.</td>
</tr>
<tr>
<td>400</td>
<td>Sent if the request is not a valid HTTP/1.0 or HTTP/1.1 request. For example HTTP/1.1 requires a host to be specified either in the Host header or as part of the URI on the request line.</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized. The user requested a document but didn’t provide a valid user name or password.</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden. Access to this URL is forbidden.</td>
</tr>
<tr>
<td>404</td>
<td>Not found. The document requested isn’t on the server. This code can also be sent if the server has been told to protect the document by telling unauthorized people that it doesn’t exist.</td>
</tr>
<tr>
<td>408</td>
<td>If the client starts a request but does not complete it within the keep-alive timeout configured in the server, then this response will be sent and the connection closed. The request can be repeated with another open connection.</td>
</tr>
<tr>
<td>411</td>
<td>The client submitted a POST request with chunked encoding, which is of variable length. However, the resource or application on the server requires a fixed length - a Content-Length header to be present. This code tells the client to resubmit its request with content-length.</td>
</tr>
<tr>
<td>413</td>
<td>Some applications (e.g., certain NSAPI plugins) cannot handle very large amounts of data, so they will return this code.</td>
</tr>
<tr>
<td>414</td>
<td>The URI is longer than the maximum the web server is willing to serve.</td>
</tr>
<tr>
<td>416</td>
<td>Data was requested outside the range of a file.</td>
</tr>
</tbody>
</table>
Response Headers

The response headers contain information about the server and the response data. The following table lists some common response headers.

<table>
<thead>
<tr>
<th>Response Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>Name and version of the web server.</td>
</tr>
<tr>
<td>Date</td>
<td>Current date (in Greenwich Mean Time).</td>
</tr>
<tr>
<td>Last-Modified</td>
<td>Date when the document was last modified.</td>
</tr>
<tr>
<td>Expires</td>
<td>Date when the document expires.</td>
</tr>
<tr>
<td>Content-Length</td>
<td>Length of the data that follows (in bytes).</td>
</tr>
<tr>
<td>Content-Type</td>
<td>MIME type of the following data.</td>
</tr>
<tr>
<td>WWW-Authenticate</td>
<td>Used during authentication and includes information that tells the browser software what is necessary for authentication (such as user name and password).</td>
</tr>
</tbody>
</table>

Response Data

The server sends a blank line after the last header. It then sends the response data such as an image or an HTML page.
Buffered Streams

Buffered streams improve the efficiency of network I/O (for example, the exchange of HTTP requests and responses), especially for dynamic content generation. Buffered streams are implemented as transparent NSPR I/O layers, which means even existing NSAPI modules can use them without any change.

The buffered streams layer adds the following features to the Sun ONE Web Server:

- **Enhanced keep-alive support:** When the response is smaller than the buffer size, the buffering layer generates the `Content-Length` header so that the client can detect the end of the response and reuse the connection for subsequent requests.

- **Response length determination:** If the buffering layer cannot determine the length of the response, it uses HTTP/1.1 chunked encoding instead of the `Content-Length` header to convey the delineation information. If the client only understands HTTP/1.0, the server must close the connection to indicate the end of the response.

- **Deferred header writing:** Response headers are written out as late as possible to give the servlets a chance to generate their own headers (for example, the session management header `set-cookie`).

- **Ability to understand request entity bodies with chunked encoding:** Though popular clients do not use chunked encoding for sending `POST` request data, this feature is mandatory for HTTP/1.1 compliance.

The improved connection handling and response length header generation provided by buffered streams also addresses the HTTP/1.1 protocol compliance issues, where absence of the response length headers is regarded as a category 1 failure. In previous Enterprise Server versions, it was the responsibility of the dynamic content generation programs to send the length headers. If a CGI script did not generate the `Content-Length` header, the server had to close the connection to indicate the end of the response, breaking the keep-alive mechanism. However, it is often very inconvenient to keep track of response length in CGI scripts or servlets, and as an application platform provider, the web server is expected to handle such low-level protocol issues.

Output buffering has been built in to the functions that transmit data, such as `net_write` (see Chapter 7, “NSAPI Function Reference”). You can specify the following Service SAF parameters that affect stream buffering, which are described in detail in the chapter “Syntax and Use of magnus.conf” in the Sun ONE Web Server 6.1 Administrator’s Configuration File Reference.
• UseOutputStreamSize
• ChunkedRequestBufferSize
• ChunkedRequestTimeout

The UseOutputStreamSize, ChunkedRequestBufferSize, and ChunkedRequestTimeout parameters also have equivalent magnus.conf directives; see “Chunked Encoding” in the chapter “Syntax and Use of magnus.conf” in the Sun ONE Web Server 6.1 Administrator's Configuration File Reference. The obj.conf parameters override the magnus.conf directives.

NOTE
The UseOutputStreamSize parameter can be set to zero (0) in the obj.conf file to disable output stream buffering. For the magnus.conf file, setting UseOutputStreamSize to zero has no effect.

To override the default behavior when invoking an SAF that uses one of the functions net_read or netbuf_grab, you can specify the value of the parameter in obj.conf, for example:

Service fn="my-service-saf" type=perf UseOutputStreamSize=8192
Buffered Streams
Appendix A

Alphabetical List of NSAPI Functions and Macros

This appendix provides an alphabetical list for the easy lookup of NSAPI functions and macros.

C

CALLOC
cinfo_find
condvar_init
condvar_notify
condvar_terminate
condvar_wait
crit_enter
crit_exit
crit_init
crit_terminate

D
daemon_atrestart
fc_close
fc_open
filebuf_buf2sd
filebuf_close
filebuf_getc
filebuf_open
filebuf_open_nostat
filter_find
filter_insert
filter_layer
filter_name
filter_remove
filter_create
flush
FREE
func_exec
func_find
func_insert
insert
log_error
MALLOC

net_flush
net_ip2host
net_read
net_sendfile
net_write
netbuf_buf2sd
netbuf_close
netbuf_getc
netbuf_grab
netbuf_open
nsapi_module_init
NSAPI_RUNTIME_VERSION
NSAPI_VERSION

param_create
param_free
pblock_copy
pblock_create
pblock_dup
pblock_find
pblock_findval
pblock_free
pblock_nminsert
pblock_nvinser
pblock_pb2env
pblock_pb2str
pblock_pinsert
pblock_remove
pblock_str2pblock
PERM_CALLOC
PERM_FREE
PERM_MALLOC
PERM_REALLOC
PERM_STRDUP
prepare_nsapi_thread
protocol_dump
protocol_set_finfo
protocol_start_response
protocol_status
protocol_uri2url
protocol_uri2url_dynamic

read
REALLOC
remove
request_get_vs
request_header
request_stat_path
request_translate_uri

sendfile
session_dns
session_maxdns
shexp_casecmp
shexp_cmp
shexp_match
shexp_valid
STRDUP
system_errmsg
system_fclose
system_flock
system_fopenRO
system_fopenRW
system_fopenWA
system_fread
system_fwrite
system_fwrite_atomic
system_gmtime
system_localtime
system_lseek
system_rename
system_unlock
system_unix2local
systhread_attach
systhread_current
systhread_getdata
systhread_newkey
systhread_setdata
systhread_sleep
systhread_start
systhread_timerset

USE_NSAPI_VERSION
util_can_exec
util_chdir2path
util_chdir2path
util_cookie_find
util_env_find
util_env_free
util_env_replace
util_env_str
util_getline
util_hostname
util_is_mozilla
util_is_url
util_itoa
util_later_than
util_sh_escape
util_snprintf
util_sprintf
util_strcasecmp
util_strftime
util_strncasecmp
util_uri_escape
util_uri_is_evil
util_uri_parse
util_uri_unescape
util_vsnprintf
util_vsprintf

vs_alloc_slot
vs_get_data
vs_get_default_httpd_object
vs_get_doc_root
vs_get_httpd_objset
vs_get_id
vs_get_mime_type
vs_lookup_config_var
vs_register_cb
vs_set_data
vs_translate_uri

write
writev
Index

A

about this guide 13
contents 16
other resources 14
AddLog 25
example of custom SAF 138
flow of control 46
requirements for SAFs 96, 100
summary 30
Administration interface
more information about 15
alphabetical reference
NSAPI functions 155, 285
API functions
CALLOC 156
cinfo_find 156
condvar_init 157
condvar_notify 158
condvar_terminate 158
condvar_wait 159
crit_enter 159
crit_exit 160
crit_init 160
crit_terminate 161
daemon_atrestart 161
dr_cache_init 269
dr_cache_refresh 270
dr_net_write 271
fc_close 163
fc_net_write 274
filebuf_buf2sd 162, 163
filebuf_close 164
filebuf_getc 165
filebuf_open 165
filebuf_open_nostat 166
filter_create 167
filter_find 168
filter_insert 169
filter_layer 169
filter_name 170
filter_remove 170
flush 171
FREE 172
func_exec 172
func_find 173
func_insert 173
insert 174
log_error 175
malloc 176
net_ip2host 177
net_read 178
net_write 180
netbuf_buf2sd 181
netbuf_close 181
netbuf_getc 182
netbuf_grab 182
netbuf_open 183
param_create 185
param_free 186
pblock_copy 186
pblock_create 187
pblock_dup 187
pblock_find 188
pblock_findval 189
pblock_free 189
pblock_rninsert 190
pblock_nvinsert 190
pblock_pb2env 191
pblock_pb2str 191
pblock_pinser 192
pblock_remove 193
pblock_str2pblock 193
PERM_FREE 195
PERM_MALLOC 194, 195, 196
PERM_SCRDUP 197
prepare_nsapi_thread 197
protocol_dump822 198
protocol_set_finfo 199
protocol_start_response 199
protocol_status 200
protocol_uri2url 201, 202
read 203
REALLOC 204
remove 204
request_get_vs 205
request_header 205
request_stat_path 206
request_translate_uri 207
sendfile 208
session_dns 209
session_maxdns 209
shexp_casemc 210
shexpcmp 210
shexp_match 211
shexp_valid 212
STRDUP 213
system_errmsg 213
system_fclose 214
system_flock 215
system_fopenRO 215
system_fopenRW 216
system_fopenWA 216
system_fread 217
system_fwrite 217
system_fwriteAtomic 218
system_gmtime 219
system_localtime 219
system_sseek 220
system_rename 221
system_unlock 220, 221
system_unix2local 222
systhread_attach 222
systhread_current 223

tutil_can_exec 228
tutil_chdir2path 228, 229
tutil_cookie_find 229
tutil_env_find 230
tutil_env_free 230
util_env_replace 231
util_env_str 231
util_getline 232
util_hostname 232
util_is_mozilla 233
util_is_url 233
util_itoa 234
util_later_than 234
util_sh_escape 235
util_snprintf 235
util_strcscmp 237
util_strerror 237
util_strmcscmp 238
util_uri_escape 239
util_uri_is_evil 239
util_uri_parse 240
util_uri_unescape 240
util_vsnprintf 241
util_vsnprintf 242
utilcookie_find 229
util-sprintf 236
vs_alloc_slot 242
vs_get_data 243
vs_get_default_httpd_object 243
vs_get_doc_root 244
vs_get_httpd_objset 244
vs_get_id 245
vs_get_mime_type 245
vs_lookup_config_var 246
vs_register_cb 246
vs_set_data 247
vs_translate_uri 248
write 248
writev 249

AUTH_TYPE environment variable 101
AUTH_USER environment variable 101
AuthTrans 24
  example of custom SAF 117
  flow of control 38
  requirements for SAFs 96, 98
  summary 27

B
  browsers 22
  buffered streams 282
  buffer-size parameter 60
  buffers-per-file parameter 60

C
  cache
    enabling memory allocation pool 74
  cache-size parameter 58
  caching
    results caching plugin 267
  CALLOC API function 156
  case sensitivity in obj.conf 49
  CGI
    environment variables in NSAPI 101
    execution 65
    to NSAPI conversion 101
cgистub-path parameter 66
  chunked encoding 282, 283
  cindex-init function 55
  cinfo NSAPI data structure 257
  cinfo_find API function 156
  client
    field in session parameter 81
    getting DNS name for 254
    getting IP address for 254
    requests 22
    sessions and 253
  Client tag 32, 34
  CLIENT_CERT environment variable 102
  comments in obj.conf 50
  Common Log subsystem
    initializing 67
  compatibility issues 80, 252
  compiling custom SAFs 85
  compression, HTTP 30
  condvar_init API function 157
  condvar_notify API function 158
  condvar_terminate API function 158
  condvar_wait API function 159
  configuration
    dynamic 26
  CONTENT_LENGTH environment variable 101
  CONTENT_TYPE environment variable 101
  context->data 105
  context->rq 105
  context->sn 105
  creating
    custom filters 103, 111
    custom SAFs 79
    custom server-parsed HTML tags 149
crit_enter API function 159
crit_exit API function 160
crit_init API function 160
crit_terminate API function 161
csd field in session parameter 81
custom
  filters 103, 115
  SAFs 79, 115
  server-parsed HTML tags 149

D
  daemon_atrestart API function 161
  data structures 251
    cinfo 257
  compatibility issues 252
  Filter 258
  FilterContext 258
  FilterLayer 258
  FilterMethods 259
  nsapi.h header file 251
  nsapi_pvt.h 252
pb_entry 254
pb_param 254
pblock 253
privatization of 252
removed from nsapi.h 252
request 255
sendfiledata 257
session 253
Session->client 254
shmem_s 256
stat 256
day of month 265
define-perf-bucket function 57
defining
custom SAFs 79
server-side tags 149
description parameter 57
directives
for handling requests 25
order of 48
summary for obj.conf 27
syntax in obj.conf 26
disable parameter 74, 75
DNS names
getting clients 254
dns_cache_init function 58
documentation
Sun ONE Web Server 13
dr_cache_init API function 269
dr_cache_refresh API function 270
dr_net_write API function 271
dynamic link library
loading 72
dynamic reconfiguration 26
dynamic results caching 267

E
environment variables
and init-cgi function 65
CGI to NSAPI conversion 101
env-variables parameter 66
Error directive 25
flow of control 47
requirements for SAFs 96, 100
summary 30
errors
finding most recent system error 213
examples
location in the build 116
of custom filters 115
of custom SAFs (plugins) 115
of custom SAFs in the build 116
quality of service 141
wildcard patterns 262
expire parameter 58

F
fancy indexing 55
fc_close API function 163
fc_net_write API function 274
file descriptor
closing 214
locking 215
opening read-only 215
opening read-write 216
opening write-append 216
reading into a buffer 217
unlocking 220, 221
writing from a buffer 217
writing without interruption 218
file I/O routines 94
file name extensions
object type 41
filebuf_buf2sd API function 162, 163
filebuf_close API function 164
filebuf_getc API function 165
filebuf_open API function 165
filebuf_open_nostat API function 166
filter methods 104
C prototypes for 104
FilterLayer data structure 105
flush 106
insert 105
remove 106
Section I

HTTPS_KEYSIZE environment variable 102
HTTPS_SECRETKEYSIZE environment variable 102

I
icon-uri parameter 56
IETF home page 277
ignore parameter 56
include directory
  for SAFs 85
indexing
  fancy 55
Init SAFs in magnus.conf 53
  requirements for SAFs 96, 97
init-cgi function 65
init-clf function 67
init-dav function 68
initializing
  for CGI 65
  global settings 53
  plugins 89
  SAFs 89
  the WebDAV subsystem 68
init-uhome function 71
Input 25
  flow of control 43
  requirements for SAFs 96, 99
  summary 28
input filters 110
insert API function 105, 174
IP address
  getting client’s 254
  custom SAFs 89
  plugins 89
  SAFs 89
load-modules function 72
  example 89
localtime
  getting thread-safe value 219
log entries, chronological order 60
log file format 60
log_error API function 175
logFileName parameter 60, 68
logging
  cookies 61
  flexible 59
  rotating logs 64

M
magnus.conf
  about 53
  SAFs in 53
Makefile file 88
MALLOC API function 176
matching
  special characters 261
maxthreads parameter 78
memory allocation
  pool-init function 74
memory management routines 93
methods parameter 76
minthreads parameter 78
month name 265

L
LateInit parameter 53
layer parameter 105
line continuation 50
linking SAFs 85
loading

N
name attribute
  in obj.conf objects 32
  in objects 33
name parameter 57, 77
NameTrans 24
example of custom SAF 119
flow of control 38
requirements for SAFs 96, 98
summary 27
native thread pools
defining in obj.conf 77
NativeThread parameter 72, 77
net_ip2host API function 177
net_read API function 178
net_write API function 180
netbuf_buf2sd API function 181
netbuf_close API function 181
netbuf_getc API function 182
netbuf_grab API function 182
netbuf_open API function 183
network I/O routines 94
nondefault objects
processing 39
NSAPI
alphabetical function reference 155, 285
CGI environment variables 101
data structure reference 251
filter interface 104
function overview 92
NSAPI filters
interface 104
methods 104
nsapi.h 251
nsapi_pvt.h 252
nt-console-init function 73

O

obj.conf
adding directives for new SAFs 89
case sensitivity 49
Client tag 34
comments 50
directive syntax 26
directives 26
directives summary 27
flow of control 37
Object tag 32

order of directives 48
parameters for directives 49
predefined SAFs 21
processing other objects 39
server instructions 26
standard directives 21
syntax rules 48
use 21
Object tag 32
name attribute 32
ppath attribute 32
object type
forcing 41
setting by file extension 41
objects
processing nondefault objects 39
ObjectType 25
example of custom SAF 126
flow of control 40
requirements for SAFs 96, 99
summary 28
opts parameter 55
order
of directives in obj.conf 48
of filters in filter stack 108
Output 25
example of custom SAF 129
flow of control 43
requirements for SAFs 96, 99
summary 29
output filters 110

P

param_create API function 185
param_free API function 186
parameter block
manipulation routines 92
SAF parameter 81
parameters
for obj.conf directives 49
for SAFs 80
path name
REMOTE_USER environment variable 101
remove API function 106, 204
replace.c 129
REQ_ABORTED response code 83
REQ_EXIT response code 83
REQ_NOACTION response code 83
REQ_PROCEED response code 83
reqpb
  field in request parameter 82
request
  NSAPI data structure 255
  SAF parameter 82
request headers 278
request_get_vs API function 205
request_header API function 205
REQUEST_METHOD environment variable 101
request_stat_path API function 206
request_translate_uri API function 207
request-handling process 96
  flow of control 37
  steps 24
request-response model 277
requests
  directives for handling 25
  how server handles 22
  HTTP 278
  methods 22
  steps in handling 24
requirements for SAFs 96
  AddLog 100
  AuthTrans 98
  Error directive 100
  Init 97
  Input 99
  NameTrans 98
  ObjectType 99
  Output 99
  PathCheck 98
  Service 100
response headers 281
responses
  HTTP 279
result codes 83
results caching plugin 267
  important functions used by 268
  rotate-access parameter 65
  rotate-callback parameter 65
  rotate-error parameter 65
  rotate-interval parameter 65
  rotate-start parameter 65
  rotating logs 64
rq SAF parameter 82
rq->headers 82
rq->reqpb 82
rq->srvhdrs 82
rq->vars 82
rules
  for editing obj.conf 48

S
SAFs
  compiling and linking 85
  creating 79
  examples of custom SAFs 115
  in magnus.conf 53
  include directory 85
  interface 80
  loading and initializing 89
  parameters 80
  predefined 21
  result codes 83
  return values 83
  signature 80
  testing 91
SCRIPT_NAME environment variable 101
search patterns 261
sendfile API function 107, 208
sendfiledata NSAPI data structure 257
separators 49
server
  flow of control 37
  initialization directives in magnus.conf 53
  instructions for using plugins 89
  instructions in obj.conf 26
  processing nondefault objects 39
request handling 22
server.xml
  variables defined in 37
SERVER_NAME environment variable 101
SERVER_PORT environment variable 101
SERVER_PROTOCOL environment variable 101
SERVER_SOFTWARE environment variable 102
SERVER_URL environment variable 102
server-parsed HTML tags
  creating custom 149
  more information 149
Service 25
  default directive 46
  directives for new SAFs (plugins) 91
  example of custom SAF 135
  examples 44
  flow of control 44
  requirements for SAFs 96, 100
  summary 29
session
  defined 253
  NSAPI data structure 253
  resolving the IP address of 209
session SAF parameter 81
Session->client NSAPI data structure 254
session_dns API function 209
session_maxdns API function 209
shared library
  loading 72
shell expression
  comparing (case-blind) to a string 210
  comparing (case-sensitive) to a string 210, 211
  validating 212
shexp_casecmp API function 210
shexp_cmp API function 210
shexp_match API function 211
shexp_valid API function 212
shlib parameter 72, 89
shmem_s NSAPI data structure 256
ShtmlTagInstanceLoad function 151
ShtmlTagInstanceUnload function 151
ShtmlTagPageLoadFunc function 151
ShtmlTagPageUnloadFn 151
sn SAF parameter 81
sn->client 81
sn->csd 81
socket
  closing 181
  reading from 178
  sending a buffer to 181
  sending file buffer to 163
  writing to 180
spaces 49
special characters 261
sprintf, see util_sprintf 236
srvhdrs
  field in request parameter 82
stackSize parameter 78
stat NSAPI data structure 256
stats-init function 76
status codes 279
stderr parameter 73
stdout parameter 73
STRDUP API function 213
streams
  buffered 282
string
  creating a copy of 213
Sun ONE Web Server documentation 14
support 19
syntax
  directives in obj.conf 26
  for editing obj.conf 48
system_errno API function 213
system_fclose API function 214
system_flock API function 215
system_fopenRO API function 215
system_fopenRW API function 216
system_fopenWA API function 216
system_fread API function 217
system_fwrite API function 217
system_fwrite_atomic API function 218
system_gmtime API function 219
system_localtime API function 219
system_lseek API function 220
system_rename API function 221
system_rename API function 220, 221
system_unix2local API function 222
systhread_attach API function 222
systhread_current API function 223
systhread_getdata API function 223
systhread_newkey API function 224
systhread_setdata API function 224
systhread_sleep API function 225
systhread_start API function 225
systhread_timerset API function 226
util_chdir2path API function 228, 229
util_cookie_find API function 229
util_env_find API function 230
util_env_free API function 230
util_env_replace API function 231
util_env_str API function 231
util_getline API function 232
util_hostname API function 232
util_is_mozilla API function 233
util_is_url API function 233
util_itoa API function 234
util_later_than API function 234
util_sh_escape API function 235
util_snprintf API function 235
util_sprintf API function 236
util_strcasecmp API function 237
util_strftime API function 237, 265
util_strncasecmp API function 238
util_uri_escape API function 239
util_uri_is_evil API function 239
util_uri_parse API function 240
util_uri_unescape API function 240
util_vsnprintf API function 241
util_vsprintf API function 242
utility routines 95

T

tags
Client 34
creating custom 149
Object 32
testing custom SAFs 91
thread
allocating a key for 224
creating 225
getting a pointer to 223
getting data belonging to 223
putting to sleep 225
setting data belonging to 224
setting interrupt timer 226
thread pools
defining in obj.conf 77
thread routines 94
thread-pool-init function 77
time formats 265
timeout parameter 66
timezones parameter 56

U

unicode 95, 240
update-interval parameter 76
URL
translated to file path 27
util_can_exec API function 228

V

vars
field in request parameter 82
virtual server routines 96
virtual-servers parameter 76
vs_alloc_slot API function 242
vs_get_data API function 243
vs_get_default_httpd_object API function 243
vs_get_doc_root API function 244
vs_get_httpd_objset API function 244
vs_get_id API function 245
vs_get_mime_type API function 245
vs_lookup_config_var API function 246
Section W

vs_register_cb API function 246
vs_set_data API function 247
vs_translate_uri API function 248
vsnprintf, see util_vsnprintf 241
vsprintf, see util_vsprintf 242

W

weekday 265
widths parameter 56
wildcard patterns 261
write API function 107, 248
writev API function 107, 249