NSAPI Programmer's Guide

iPlanet Web Server, Enterprise Edition

Version 6.0

8xx-xxxx-xx
May 2001

Contents

About This Book

This book was last updated 5/15/01.

This book discusses how to use Netscape Server Application Programmer's Interface (NSAPI) to build plugins that define Server Application Functions (SAFs) to extend and modify iPlanet™ Web Server, Edition version 6.0. The book also discusses the purpose and use of the configuration files `obj.conf`, `magnus.conf`, server.xml, and `mime.types`, and provides comprehensive lists of the directives and functions that can be used in these configuration files. It also provides a reference of the NSAPI functions you can use to define new plugins.

This book has the following chapters and appendices:

- Chapter 1, "Basics of Server Operation"

  This chapter discusses how the iPlanet Web Server uses configuration files to perform initialization tasks and to process client requests.

- Chapter 2, "Syntax and Use of obj.conf"

  This chapter goes into detail on the configuration file `obj.conf`. The chapter discusses the syntax and use of directives in this file, which instruct the server how to process requests.

- Chapter 3, "Predefined SAFs and the Request Handling Process"

  This chapter discusses each of the stages in the request handling process, and provides an API reference of the Server Application Functions (SAFs) that can be invoked at each stage.

- Chapter 4, "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 5, "NSAPI Function Reference"

  This chapter presents a reference of the functions in the Netscape Server Application Programming Interface (API). You use NSAPI functions to define SAFs.

- Chapter 6, "Examples of Custom SAFs"

  This chapter discusses examples of custom SAFs to use at each stage in the request handling process.

- Chapter 7, "Syntax and Use of magnus.conf"

  This appendix discusses the variables you can set in the configuration file `magnus.conf` to configure the iPlanet Web Server during initialization.

- Chapter 8, "Virtual Server Configuration Files"

  This appendix discusses the variables you can set in the configuration file `server.xml` to configure virtual servers in iPlanet Web Server.

- Appendix A, "Data Structure Reference"

  This appendix discusses some of the commonly used NSAPI data structures.

- Appendix B, "MIME Types"

  This appendix discusses the MIME types file, which maps file extensions to file types.

- Appendix C, "Wildcard Patterns"

  This appendix lists the wildcard patterns you can use when specifying values in `obj.conf`, various predefined SAFs, and in some NSAPI functions.

- Appendix D, "Time Formats"

  This appendix lists time formats.

- Appendix E, "HyperText Transfer Protocol"

  This appendix gives an overview of HTTP.

- Appendix F, "Dynamic Results Caching Functions"

  This appendix explains how to create a results caching plugin.

- Appendix G, "Alphabetical List of NSAPI Functions and Macros"
  Appendix H, "Alphabetical List of Directives in magnus.conf"
  Appendix I, "Alphabetical List of Pre-defined SAFs"

  These appendices provide alphabetical lists for easy lookup of NSAPI functions, predefined SAFs, and variables in `magnus.conf`.

NOTE Throughout this manual, all Unix-specific descriptions apply to the Linux operating system as well, except where Linux is specifically mentioned.

Chapter 1

Basics of Server Operation

The configuration and behavior of iPlanet Web Server is determined by a set of configuration files. You can change the settings in these configuration files either by using the Server Manager interface or by manually editing the files.

The configuration file that contains instructions for how the server processes requests from clients is called `obj.conf`. You can modify and extend the request handling process by adding or changing the instructions in `obj.conf`. You can use the Netscape Server Application Programming Interface (API) to create new Server Application Functions (SAFs) to use in instructions in `obj.conf`.

This chapter discusses the configuration files used by the iPlanet Web Server. Then the chapter looks in more detail at the server's process for handling requests. The chapter closes by introducing the use of Netscape Server Application Programming Interface (NSAPI) to define new functions to modify the request-handling process.

This chapter has the following sections:

- Configuration Files
- Dynamic Reconfiguration
- How the Server Handles Requests from Clients
- Writing New Server Application Functions

Configuration Files

The configuration and operation of the iPlanet Web Server is controlled by configuration files. The configuration files reside in the directory *server-root*/*server-id*/`config/`. This directory contains various configuration files for controlling different components. The exact number and names of configuration files depends on which components have been enabled or loaded into the server.

However, this directory always contains four configuration files that are essential for the server to operate. These files are:

- `magnus.conf` -- contains global server initialization information.
- `server.xml` -- contains initialization information for virtual servers and listen sockets.
- `obj.conf` -- contains instructions for handling requests from clients.
- `mime.types` -- contains information for determining the content type of requested resources.

magnus.conf

This file sets values of variables that configure the server during initialization. The server looks at this file and executes the settings on startup. The server does not look at this file again until it is restarted.

See Chapter 7, "Syntax and Use of magnus.conf" for a list of all the variables and `Init` directives that can be set in `magnus.conf`.

server.xml

This file configures the addresses and ports that the server listens on and assigns virtual server classes and virtual servers to these listen sockets. A master file, `server.dtd`, defines its format and content.

For more information about how the server uses `server.dtd` and `server.xml`, see Chapter 8, "Virtual Server Configuration Files."

NOTE	Virtual servers are not the same thing as server instances. Each server instance is a completely separate server that contains one or more virtual servers.

obj.conf

This file contains instructions for the server about how to process requests from clients (such as browsers). The server looks at the configuration defined by this file every time it processes a request from a client.

There is one `obj.conf` file for each virtual server class, or grouping of virtual servers. 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.

All the `obj.conf` files are located in the *server_root*`/`*server_id*`/config` directory. They are typically named *vsclass*`.obj.conf`, where *vsclass* is the virtual server class name.

The `obj.conf` file is essential to the operation of the iPlanet Web Server. When you make changes to the server through the Server Manager interface, the system automatically updates `obj.conf`.

The file `obj.conf` contains a series of instructions (directives) that tell the iPlanet Web Server what to do at each stage in the request-response process. Each directive invokes a Server Application Function (SAF). These functions are written using the Netscape Server Application Programming Interface (NSAPI). The iPlanet Web Server comes with a set of pre-defined SAFs, but you can also write your own using NSAPI to create new instructions that modify the way the server handles requests.

For more information about how the server uses `obj.conf`, see Chapter 2, "Syntax and Use of obj.conf."

mime.types

This file maps file extensions to MIME types to enable the server to determine the content type of a requested resource. For example, requests for resources with `.html` extensions indicate that the client is requesting an HTML file, while requests for resources with `.gif` extensions indicate that the client is requesting an image file in GIF format.

For more information about how the server uses `mime.types`, see Appendix B, "MIME Types."

Dynamic Reconfiguration

You do not have to restart the server for changes to `obj.conf`, `mime.types`, `server.xml`, and virtual-server-specific ACL files to take effect. 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 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.

How the Server Handles Requests from Clients

iPlanet Web Server is a web server that accepts and responds to HyperText Transfer Protocol (HTTP) requests. Browsers like Netscape Communicator communicate using several protocols including HTTP, FTP, and gopher. The iPlanet Web Server handles HTTP specifically.

For more information about the HTTP protocol refer to Appendix E, "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 which 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: Netscape-Enterprise/6.0
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."

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 "Virtual Server Selection for Request Processing," on page 299.

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. **Service** (generate the response)

   generate and return the response to the client.

6. **AddLog** (adding log entries)

   add entries to log file(s).

7. **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 iPlanet 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`, `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:/Netscape/Server4/docs`. (The `document-root` function translates the `http://`*server_name/* part of the URL to the document root, which in this example is `D:/Netscape/Server4/docs`.)

```
NameTrans fn="document-root" root="D:/Netscape/Server4/docs"
```

The functions invoked by the directives in `obj.conf` are known as Server Application Functions (SAFs).

Writing New Server Application Functions

The iPlanet Web Server comes with a variety of pre-defined SAFs that you can use to create more directives in `obj.conf`. You can also write your own SAF using the functions provided by the NSAPI. After you write the SAF, you would add a directive to `obj.conf` so that your new function gets invoked by the server at the appropriate time.

Each SAF has its own arguments, which are passed to it by the directive in `obj.conf`. Every SAF is also passed additional arguments that contain information about the request (such as what resource was requested and what kind of client requested it) and any other server variables created or modified by SAFs called by previously invoked directives. Each SAF may examine, modify, or create server variables.

Each SAF returns a result code which tells the server whether it succeeded, did nothing, or failed.

For more information about `obj.conf`, see Chapter 2, “Syntax and Use of obj.conf.”

For more information on the pre-defined SAFs, see Chapter 3, “Predefined SAFs and the Request Handling Process.”

For more information on writing your own SAFs, see Chapter 4, “Creating Custom SAFs.”

Chapter 2

Syntax and Use of obj.conf

The `obj.conf` configuration file contains directives that instruct the iPlanet Web Server how to handle requests from clients. 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.

The sections in this chapter are:

- Server Instructions in obj.conf
- The Object Tag
- Variables Defined in server.xml
- Flow of Control in obj.conf
- Syntax Rules for Editing obj.conf
- About obj.conf Directive Examples

Server Instructions in obj.conf

The `obj.conf` file contains directives that instruct the server how to handle requests received from clients such as browser. 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:/Netscape/Server4/docs"
```

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

The value of the `fn` argument is the name of the Server Application Function (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.

iPlanet 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`, as discussed in Chapter 3, "Predefined SAFs and the Request Handling Process." You can also define new SAFs, as discussed in Chapter 4, "Creating Custom SAFs."

The `magnus.conf` file contains `Init` directive SAFs that initialize the server. For more information, see Chapter 7, "Syntax and Use of magnus.conf."

Summary of the Directives

Here 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 34 explains exactly how the server decides which directive or directives to execute in at each stage.

- `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:/Netscape/Server4/docs"
```

This example calls the `document-root` function with a `root` argument of `D:/Netscape/Server4/docs`. The function `document-root` function translates the `http://`*server_name*`/` part of the requested to URL to the document root, which in this case is `D:/Netscape/Server4/docs`. Thus a request for `http://`*server-name*`/doc1.html` is translated to `D:/Netscape/Server4/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.

- `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:/netscape/server4/errors/unauthorized.html"
  ```

  In this example, the server sends the file in `D:/netscape/server4/errors/unauthorized.html` whenever a client requests a resource that it is not authorized to access.

The Object Tag

Directives in the `obj.conf` file are grouped into objects that begin with an `<Object>` tag and end with a `</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/netscape/server4/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 pathname 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 a 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 the section "Flow of Control in obj.conf," on page 34.

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/`.

```
<Object name="default">
NameTrans fn="pfx2dir" from="/cgi"
dir="D:/netscape/server4/docs/mycgi" name="cgi"
...
</Object>
```

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

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

Object 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 pathname. If this physical pathname 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:/Netscape/Server4/docs/` (which is the document root directory).

```
<Object name="default">
NameTrans fn="document-root" root="D:/Netscape/Server4/docs"
...
</Object>
```

The URL `http://`*server_name*`/internalplan1.html` would be translated to `D:/Netscape/Server4/docs/internalplan1.html`. However, suppose that `obj.conf` contains the following additional object:

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

In this case, the partial path `*internal*` matches the path `D:/Netscape/Server4/docs/internalplan1.html`. So now the server starts processing the directives in this object before processing the remaining directives in the default object.

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`:

```
<!DOCTYPE SERVER SYSTEM "server.dtd" [
<!ATTLIST VARS
   docroot CDATA #IMPLIED
>
]>
...
      <VS id="a.com" connections="maingroup" urlhosts="a.com"
            mime="mime1" aclids="std">
         <VARS docroot="/u/server6/a/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 iPlanet Web Server configuration files.

Any variable referenced in an `obj.conf` file must be defined in the `server.xml` file at the SERVER, VSCLASS, or VS level. Defining variables with default values at the SERVER or VSCLASS level and overriding them in the VS is recommended.

For more information, see Chapter 8, “Virtual Server Configuration Files.”

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 “Virtual Server Selection for Request Processing,” on page 299.

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`.

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 pathname 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 functions 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:/Netscape/Server4/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 pathname `D:/netscape/server4/docs/mycgi/`:

```
NameTrans fn="pfx2dir" from="/cgi"
dir="D:/netscape/server4/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 pathname would be `D:/netscape/server4/docs/cgi/` (not `mycgi`). This illustrates why the directive that invokes `document-root` must be the last one in the `NameTrans` section.

How the Server Knows to Process 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/`.

```
<Object name="default">
...
NameTrans fn="pfx2dir" from="/cgi"
dir="D:/netscape/server4/docs/mycgi" name="cgi"
...
</Object>
```

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

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

When a `NameTrans` directive has been successfully executed, there will be a physical pathname associated with the requested resource. If the resultant pathname 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:

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

Now suppose the successful `NameTrans` directive translates the requested URL to the pathname `D:/Netscape/Server4/docs/internalplan1.html`. In this case, the partial path `*internal*` matches the path `D:/Netscape/Server4/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 pathname 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 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 pathname 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 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 that have 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 Appendix B, “MIME Types.”

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`.

```
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:/netscape/server4/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`.

```
NameTrans fn="pfx2dir" from="/cgi"
dir="D:/netscape/server4/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.

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 round 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:/netscape/server4/docs/jos.html`:

  ```
  NameTrans fn="document-root" root="D:/Netscape/Server4/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 Appendix C, "Wildcard Patterns.") This directive sends the requested file, `jos.html`, to the client.

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

For an example that involves using another object, consider what happens when the server receives a request for `http://`*server_name*`/servlet/doCalculation.class`. This example assumes that servlets have been activated and the directory `D://netscape/server4/docs/servlet/` has been registered as a servlet directory (that is, the server treats all files in that directory as servlets).

- The following `NameTrans` directive translates the requested URL to `D:netscape/Server4/docs/servlet/doCalculation.class`. This directive also assigns the name `ServletByExt` to the request.

```
NameTrans fn="pfx2dir" from="/servlet"
dir="D:/Netscape/Server4/docs/servlet" name="ServletByExt"
```

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

```
<Object name="ServletByExt">
ObjectType fn="force-type" type="magnus-internal/servlet"
Service type="magnus-internal/servlet" fn="NSServletService"
</Object>
```

- The `ServletByExt` object has no `PathCheck` directives, so the server processes the `PathCheck` directives in the default object. Let's assume that all `PathCheck` directives succeed.
- Next, the server processes the `ObjectType` directives, starting with the one in the `ServletByExt` object. This directive sets the `type` attribute to `magnus-internal/servlet`.

  ```
  ObjectType fn="force-type" type="magnus-internal/servlet"
  ```

  The server continues processing all the `ObjectType` directives in the default object, but since the `type` attribute is already set its value cannot be changed.
- When processing `Service` directives, the server starts by considering the `Service` directive in the `ServletByExt` object which is:

  ```
  Service type="magnus-internal/servlet" fn="NSServletService"
  ```

- The `type` argument of this directive matches the `type` value that was set by the `ObjectType` directive. So the server goes ahead and executes this `Service` directive which calls the `NSServletService` function. This function invokes the requested file as a servlet and sends the output from the servlet as the response to the client. (If the requested resource is not a servlet, an error occurs.)

  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 thing (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 Appendix C, “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 7, “Syntax and Use of magnus.conf.”

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, then the server immediately stops executing all other directives and immediately starts executing the `Error` directives.

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 incorrectly.

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, in the `ObjectType` section the order 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 round), 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.

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

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 back-slashes (\) in path names under Windows NT. Back-slash 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
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:/netscape/server4/docs/mycgi" name="cgi"
```

Chapter 3

Predefined SAFs and the Request Handling Process

This chapter describes the standard directives and pre-defined Server Application Functions (SAFs) that are used in the `obj.conf` file to give instructions to the server. For a discussion of the use and syntax of `obj.conf`, see the previous chapter, Chapter 2, “Syntax and Use of obj.conf.”

For a list of `Init` (initialization) SAFs, see Chapter 7, “Syntax and Use of magnus.conf.”

This chapter includes functions that are part of the core functionality of iPlanet Web Server. It does not include functions that are available only if additional components, such as servlets and server-parsed HTML, are enabled.

This chapter contains a section for each directive which lists all the pre-defined Server Application Functions that can be used with that directive.

The directives are:

- `AuthTrans Stage`
- `NameTrans Stage`
- `PathCheck Stage`
- `ObjectType Stage`
- `Service Stage`
- `AddLog Stage`
- `Error Stage`

For an alphabetical list of pre-defined SAFs, see Appendix H, “Alphabetical List of Directives in magnus.conf.”

Table 3-1 lists the SAFs that can be used with each directive.

Table 3-1 Available Server Application Functions (SAFs) Per Directive

AuthTrans Stage	`basic-auth`
	`basic-ncsa`
	`get-sslid`
	`qos-handler`

NameTrans Stage	`assign-name`
	`document-root`
	`home-page`
	`pfx2dir`
	`redirect`
	`strip-params`
	`unix-home`

PathCheck Stage	`check-acl`
	`deny-existence`
	`find-index`
	`find-links`
	`find-pathinfo`
	`get-client-cert`
	`load-config`
	`nt-uri-clean`
	`ntcgicheck`
	`require-auth`
	`set-virtual-index`
	`ssl-check`
	`ssl-logout`
	`unix-uri-clean`

ObjectType Stage	`force-type`
	`set-default-type`
	`shtml-hacktype`
	`type-by-exp`
	`type-by-extension`

Table 3-1 Available Server Application Functions (SAFs) Per Directive

Service Stage	add-footer add-header append-trailer imagemap index-common index-simple key-toosmall list-dir make-dir query-handler remove-dir remove-file rename-file send-cgi send-file send-range send-shellcgi send-wincgi service-dump shtml_send stats-xml upload-file
AddLog Stage	common-log flex-log record-useragent
Error Stage	send-error qos-error

The bucket Parameter

The following performance buckets are predefined in iPlanet Web Server:

- The `default-bucket` records statistics for the functions not associated with any user-defined or built-in bucket.
- The `all-requests` bucket records `.perf` statistics for all NSAPI SAFs, including those in the `default-bucket`.

You can define additional performance buckets in the `magnus.conf` file (see the `perf-init` and `define-perf-bucket` functions).

You can measure the performance of any SAF in `obj.conf` by adding a `bucket=`*bucket-name* parameter to the function, for example `bucket=cache-bucket`.

To list the performance statistics, use the `service-dump` Service function.

As an alternative, you can use the `stats-xml` Service function to generate performance statistics; use of buckets is optional.

For more information about performance buckets, see the *Performance Tuning, Sizing, and Scaling Guide for iPlanet Web Server*.

AuthTrans Stage

`AuthTrans` stands for Authorization Translation. `AuthTrans` directives give the server instructions for checking authorization before allowing a client to access resources. `AuthTrans` directives work in conjunction with `PathCheck` directives. Generally, an `AuthTrans` function checks if the username 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.

The server handles the authorization of client users in two steps.

- AuthTrans Directive - validates authorization information sent by the client in the Authorization header.
- PathCheck Stage - checks that the authorized user is allowed access to the requested resource.

The authorization process is split into two steps so that multiple authorization schemes can be easily incorporated, as well as providing the flexibility to have resources that record authorization information but do not require it.

`AuthTrans` functions get the username and password from the headers associated with the request. When a client initially makes a request, the username and password are unknown so the `AuthTrans` functions and `PathCheck` functions work together to reject the request, since they can't validate the username and password. When the client receives the rejection, its usual response is to pop up a dialog box asking for the username and password to enter the appropriate realm, and then the client submits the request again, this time including the username and password in the headers.

If there is more than one `AuthTrans` directive in `obj.conf`, each function is executed in order until one succeeds in authorizing the user.

The following AuthTrans-class functions are described in detail in this section:

- basic-auth calls a custom function to verify user name and password. Optionally determines the user's group.
- basic-ncsa verifies user name and password against an NCSA-style or system DBM database. Optionally determines the user's group.
- get-sslid retrieves a string that is unique to the current SSL session and stores it as the ssl-id variable in the Session->client parameter block.
- qos-handler handles the current quality of service statistics.

basic-auth

Applicable in AuthTrans-class directives.

The basic-auth function calls a custom function to verify authorization information sent by the client. The Authorization header is sent as part of the basic server authorization scheme.

This function is usually used in conjunction with the PathCheck-class function require-auth.

Parameters

auth-type	specifies the type of authorization to be used. This should always be basic.
userdb	(optional) specifies the full path and file name of the user database to be used for user verification. This parameter will be passed to the user function.
userfn	is the name of the user custom function to verify authorization. This function must have been previously loaded with load-modules. It has the same interface as all the SAFs, but it is called with the user name (user), password (pw), user database (userdb), and group database (groupdb) if supplied, in the pb parameter. The user function should check the name and password using the database and return REQ_NOACTION if they are not valid. It should return REQ_PROCEED if the name and password are valid. The basic-auth function will then add auth-type, auth-user (user), auth-db (userdb), and auth-password (pw, Windows NT only) to the rq->vars pblock.
groupdb	(optional) specifies the full path and file name of the user database. This parameter will be passed to the group function.

`groupfn`	(optional) is the name of the group custom function that must have been previously loaded with `load-modules`. It has the same interface as all the SAFs, but it is called with the user name (user), password (pw), user database (`userdb`), and group database (`groupdb`) in the pb parameter. It also has access to the `auth-type`, `auth-user` (`user`), `auth-db` (`userdb`), and `auth-password` (`pw`, Windows NT only) parameters in the `rq->vars pblock.` The group function should determine the user's group using the group database, add it to `rq->vars` as `auth-group`, and return `REQ_PROCEED` if found. It should return `REQ_NOACTION` if the user's group is not found.
`bucket`	optional, common to all `obj.conf` functions

Examples

in `magnus.conf`:

```
Init fn=load-modules shlib=/path/to/mycustomauth.so
funcs=hardcoded_auth
```

in `obj.conf`:

```
AuthTrans fn=basic-auth auth-type=basic userfn=hardcoded_auth

PathCheck fn=require-auth auth-type=basic realm="Marketing Plans"
```

See Also

`require-auth`

basic-ncsa

Applicable in `AuthTrans`-class directives.

The `basic-ncsa` function verifies authorization information sent by the client against a database. The Authorization header is sent as part of the basic server authorization scheme.

This function is usually used in conjunction with the PathCheck-class function `require-auth`.

Parameters

`auth-type`	specifies the type of authorization to be used. This should always be `basic`.
`dbm`	(optional) specifies the full path and base file name of the user database in the server's native format. The native format is a system DBM file, which is a hashed file format allowing instantaneous access to billions of users. If you use this parameter, don't use the `userfile` parameter as well.
`userfile`	(optional) specifies the full path name of the user database in the NCSA-style HTTPD user file format. This format consists of lines using the format *name:password*, where *password* is encrypted. If you use this parameter, don't use `dbm`.
`grpfile`	(optional) specifies the NCSA-style HTTPD group file to be used. Each line of a group file consists of *group*: *user1 user2 ... userN* where each user is separated by spaces.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
AuthTrans fn=basic-ncsa auth-type=basic
dbm=/netscape/server4/userdb/rs

PathCheck fn=require-auth auth-type=basic realm="Marketing Plans"
AuthTrans fn=basic-ncsa auth-type=basic
userfile=/netscape/server4/.htpasswd
grpfile=/netscape/server4/.grpfile

PathCheck fn=require-auth auth-type=basic realm="Marketing Plans"
```

See Also

`require-auth`

get-sslid

Applicable in `AuthTrans`-class directives.

NOTE This function is provided for backward compatibility only. The functionality of `get-sslid` has been incorporated into the standard processing of an SSL connection.

The `get-sslid` function retrieves a string that is unique to the current SSL session, and stores it as the `ssl-id` variable in the `Session->client` parameter block.

If the variable `ssl-id` is present when a CGI is invoked, it is passed to the CGI as the `HTTPS_SESSIONID` environment variable.

The `get-sslid` function has no parameters and always returns `REQ_NOACTION`. It has no effect if SSL is not enabled.

Parameters

`bucket`	optional, common to all `obj.conf` functions

qos-handler

Applicable in `AuthTrans`-class directives.

The `qos-handler` function examines the current quality of service statistics for the virtual server, virtual server class, and global server, logs the statistics, and enforces the QOS parameters by returning an error. This must be the first `AuthTrans` function configured in the `default` object in order to work properly.

The code for this SAF is one of the examples in Chapter 6, "Examples of Custom SAFs."

For more information, see the *Performance Tuning, Sizing, and Scaling Guide for iPlanet Web Server.*

Parameters

`bucket`	optional, common to all `obj.conf` functions

Example

```
AuthTrans fn=qos-handler
```

See Also

qos-error

NameTrans Stage

NameTrans stands for Name Translation. NameTrans directives translate virtual URLs to physical directories on your server. For example, the URL

http://www.test.com/some/file.html

could be translated to the full file-system path

/usr/netscape/server4/docs/some/file.html

NameTrans directives should appear in the default object. If there is more than one NameTrans directive in an object, the server executes each one in order until one succeeds.

The following NameTrans-class functions are described in detail in this section:

- assign-name tells the server to process directives in a named object.
- document-root translates a URL into a file system path by replacing the http://*server-name/* part of the requested resource with the document root directory.
- home-page translates a request for the server's root home page (/) to a specific file.
- pfx2dir translates any URL beginning with a given prefix to a file system directory and optionally enables directives in an additional named object.
- redirect redirects the client to a different URL.
- strip-params removes embedded semicolon-delimited parameters from the path.
- unix-home translates a URL to a specified directory within a user's home directory.

assign-name

Applicable in NameTrans-class directives.

The assign-name function specifies the name of an object in obj.conf that matches the current request. The server then processes the directives in the named object in preference to the ones in the default object.

For example, consider the following directive in the default object:

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

Let's suppose the server receives a request for http://*server-name*/personnel. After processing this NameTrans directive, the server looks for an object named personnel in obj.conf, and continues by processing the directives in the personnel object.

The assign-name function always returns REQ_NOACTION.

Parameters

from	is a wildcard pattern that specifies the path to be affected.
name	specifies an additional named object in obj.conf whose directives will be applied to this request.
find-pathinfo-forward	(optional) makes the server look for the PATHINFO forward in the path right after the ntrans-base instead of backward from the end of path as the server function assign-name does by default. The value you assign to this parameter is ignored. If you do not wish to use this parameter, leave it out. The find-pathinfo-forward parameter is ignored if the ntrans-base parameter is not set in rq->vars. By default, ntrans-base is set. This feature can improve performance for certain URLs by reducing the number of stats performed.

`nostat`	(optional) prevents the server from performing a stat on a specified URL whenever possible. The effect of `nostat="`*virtual-path*`"` in the `NameTrans` function `assign-name` is that the server assumes that a stat on the specified *virtual-path* will fail. Therefore, use `nostat` only when the path of the *virtual-path* does not exist on the system, for example, for NSAPI plugin URLs, to improve performance by avoiding unnecessary stats on those URLs. When the default `PathCheck` server functions are used, the server does not stat for the paths `/`*ntrans-base/virtual-path* and `/`*ntrans-base/virtual-path*`/*` if *ntrans-base* is set (the default condition); it does not stat for the URLs `/`*virtual-path* and `/`*virtual-path*`/*` if *ntrans-base* is not set.
`bucket`	optional, common to all `obj.conf` functions

Example

```
# This NameTrans directive is in the default object.
NameTrans fn=assign-name name=personnel from=/a/b/c/pers
...
<Object name=personnel>
...additional directives..
</Object>
NameTrans fn="assign-name" from="/perf" find-pathinfo-forward=""
name="perf"
NameTrans fn="assign-name" from="/nsfc"  nostat="/nsfc"
name="nsfc"
```

document-root

Applicable in `NameTrans`-class directives.

The `document-root` function specifies the root document directory for the server. If the physical path has not been set by a previous `NameTrans` function, the `http://`*server-name*`/` part of the path is replace by the physical pathname for the document root.

When the server receives a request for `http://`*server-name*`/somepath/somefile`, the `document-root` function replaces `http://`*server-name*`/` with the value of its `root` parameter. For example, if the document root directory is `/usr/netscape/server4/docs`, then when the server receives a request for `http://`*server-name*`/a/b/file.html`, the `document-root` function translates the pathname for the requested resource to `/usr/netscape/server4/docs/a/b/file.html`.

This function always returns `REQ_PROCEED`. `NameTrans` directives listed after this will never be called, so be sure that the directive that invokes `document-root` is the last `NameTrans` directive.

There can be only one root document directory. To specify additional document directories, use the `pfx2dir` function to set up additional path name translations.

Parameters

`root`	is the file system path to the server's root document directory.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
NameTrans fn=document-root root=/usr/netscape/server4/docs
NameTrans fn=document-root root=$docroot
```

See also

`pfx2dir`

home-page

Applicable in `NameTrans`-class directives.

The `home-page` function specifies the home page for your server. Whenever a client requests the server's home page (`/`), they'll get the document specified.

Parameters

`path`	is the path and name of the home page file. If `path` starts with a slash (/), it is assumed to be a full path to a file. This function sets the server's `path` variable and returns `REQ_PROCEED`. If `path` is a relative path, it is appended to the URI and the function returns `REQ_NOACTION` continuing on to the other NameTrans directives.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
NameTrans fn="home-page" path="homepage.html"
NameTrans fn="home-page" path="/httpd/docs/home.html"
```

pfx2dir

Applicable in `NameTrans`-class directives.

The `pfx2dir` function replaces a directory prefix in the requested URL with a real directory name. It also optionally allows you to specify the name of an object that matches the current request. (See the discussion of `assign-name` for details of using named objects.)

Parameters

`from`	is the URI prefix to convert. It should not have a trailing slash (/).
`dir`	is the local file system directory path that the prefix is converted to. It should not have a trailing slash (/).
`name`	(optional) specifies an additional named object in `obj.conf` whose directives will be applied to this request.

`find-pathinfo-forward`	(optional) makes the server look for the PATHINFO forward in the path right after the ntrans-base instead of backward from the end of path as the server function `find-pathinfo` does by default. The value you assign to this parameter is ignored. If you do not wish to use this parameter, leave it out. The `find-pathinfo-forward` parameter is ignored if the `ntrans-base` parameter is not set in `rq->vars` when the server function `find-pathinfo` is called. By default, `ntrans-base` is set. This feature can improve performance for certain URLs by reducing the number of stats performed in the server function `find-pathinfo`. On NT, this feature can also be used to prevent the PATHINFO from the server URL normalization process (changing '\' to '/') when the `PathCheck` server function `find-pathinfo` is used. Some double-byte characters have hex values that may be parsed as URL separator characters such as \ or ~. Using the `find-pathinfo-forward` parameter can sometimes prevent incorrect parsing of URLs containing double-byte characters.
`bucket`	optional, common to all `obj.conf` functions

Examples

In the first example, the URL `http://`*server-name*`/cgi-bin/`*resource* (such as `http://x.y.z/cgi-bin/test.cgi`) is translated to the physical pathname `/httpd/cgi-local/`*resource*, (such as `/httpd/cgi-local/test.cgi`) and the server also starts processing the directives in the object named `cgi`.

```
NameTrans fn=pfx2dir from=/cgi-bin dir=/httpd/cgi-local name=cgi
```

In the second example, the URL `http://`*server-name*`/icons/`*resource* (such as `http://x.y.z/icons/happy/smiley.gif`) is translated to the physical pathname `/users/nikki/images/`*resource*, (such as `/users/nikki/images/smiley.gif`)

```
NameTrans fn=pfx2dir from=/icons/happy dir=/users/nikki/images
```

The third example shows the use of the `find-pathinfo-forward` parameter. The URL `http://`*server-name*`/cgi-bin/`*resource* is translated to the physical pathname `/export/home/cgi-bin/`*resource*.

```
NameTrans fn="pfx2dir" find-pathinfo-forward="" from="/cgi-bin"
dir="/export/home/cgi-bin" name="cgi"
```

redirect

Applicable in `NameTrans`-class directives.

The `redirect` function lets you change URLs and send the updated URL to the client. When a client accesses your server with an old path, the server treats the request as a request for the new URL.

Parameters

`from`	specifies the prefix of the requested URI to match.
`url`	(maybe optional) specifies a complete URL to return to the client. If you use this parameter, don't use `url-prefix` (and vice-versa).
`url-prefix`	(maybe optional) is the new URL prefix to return to the client. The `from` prefix is simply replaced by this URL prefix. If you use this parameter, don't use `url` (and vice-versa).
`escape`	(optional) is a flag which tells the server to `util_uri_escape` the URL before sending it. It should be `yes` or `no`. The default is `yes`.
`bucket`	optional, common to all `obj.conf` functions

Examples

In the first example, any request for `http://`*server-name*`/`*whatever* is translated to a request for `http://tmpserver/`*whatever*.

```
NameTrans fn=redirect from=/ url-prefix=http://tmpserver
```

In the second example, any request for `http://`*server-name*`/toopopular/`*whatever* is translated to a request for `http://bigger/better/stronger/morepopular/`*whatever*.

```
NameTrans fn=redirect from=/toopopular
url=http://bigger/better/stronger/morepopular
```

strip-params

Applicable in `NameTrans`-class directives.

The `strip-params` function removes embedded semicolon-delimited parameters from the path. For example, a URI of `/dir1;param1/dir2` would become a path of `/dir1/dir2`. When used, the `strip-params` function should be the first `NameTrans` directive listed.

Parameters

`bucket`	optional, common to all `obj.conf` functions

Example

`NameTrans fn=strip-params`

unix-home

Applicable in `NameTrans`-class directives.

Unix Only. The `unix-home` function translates user names (typically of the form ~username) into the user's home directory on the server's Unix machine. You specify a URL prefix that signals user directories. Any request that begins with the prefix is translated to the user's home directory.

You specify the list of users with either the `/etc/passwd` file or a file with a similar structure. Each line in the file should have this structure (elements in the `passwd` file that are not needed are indicated with *):

```
username:*:*:groupid:*:homedir:*
```

If you want the server to scan the password file only once at startup, use the Init-class function `init-uhome` in `magnus.conf`.

Parameters

`from`	is the URL prefix to translate, usually "`/~`".
`subdir`	is the subdirectory within the user's home directory that contains their web documents.
`pwfile`	(optional) is the full path and file name of the password file if it is different from `/etc/passwd`.
`name`	(optional) specifies an additional named object whose directives will be applied to this request.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
NameTrans fn=unix-home from=/~ subdir=public_html
NameTrans fn=unix-home from /~ pwfile=/mydir/passwd
subdir=public_html
```

See Also

`init-uhome, find-links`

PathCheck Stage

`PathCheck` directives check the local file system path that is returned after the `NameTrans` step. The path is checked for things such as CGI path information and for dangerous elements such as `/./`and `/../` and `//`, and then any access restriction is applied.

If there is more than one `PathCheck` directive, each of the functions are executed in order.

The following PathCheck-class functions are described in detail in this section:

- `check-acl` checks an access control list for authorization.
- `deny-existence` indicates that a resource was not found.
- `find-index` locates a default file when a directory is requested.
- `find-links` denies access to directories with certain file system links
- `find-pathinfo` locates extra path info beyond the file name for the PATH_INFO CGI environment variable.
- `get-client-cert` gets the authenticated client certificate from the SSL3 session.
- `load-config` finds and loads extra configuration information from a file in the requested path
- `nt-uri-clean` denies access to requests with unsafe path names by indicating not found.
- `ntcgicheck` looks for a CGI file with a specified extension.
- `require-auth` denies access to unauthorized users or groups.
- `set-virtual-index` specifies a virtual index for a directory.
- `ssl-check` checks the secret keysize.
- `ssl-logout` invalidates the current SSL session in the server's SSL session cache.
- `unix-uri-clean` denies access to requests with unsafe path names by indicating not found.

check-acl

Applicable in `PathCheck`-class directives.

The `check-acl` function specifies an Access Control List (ACL) to use to check whether the client is allowed to access the requested resource. An access control list contains information about who is or is not allowed to access a resource, and under what conditions access is allowed.

Regardless of the order of `PathCheck` directives in the object, `check-acl` functions are executed first. They cause user authentication to be performed, if required by the specified ACL, and will also update the access control state.

Parameters

`acl`	is the name of an Access Control List.
`path`	(optional) is a wildcard pattern that specifies the path for which to apply the ACL.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
PathCheck fn=check-acl acl="*HRonly*"
```

deny-existence

Applicable in `PathCheck`-class directives.

The `deny-existence` function sends a “not found” message when a client tries to access a specified path. The server sends “not found” instead of “forbidden,” so the user cannot tell whether the path exists or not.

Parameters

`path`	(optional) is a wildcard pattern of the file-system path to hide. If the path does not match, the function does nothing and returns `REQ_NOACTION`. If the path is not provided, it is assumed to match.
`bong-file`	(optional) specifies a file to send rather than responding with the “not found” message. It is a full file-system path.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
PathCheck fn=deny-existence
path=/usr/netscape/server4/docs/private
PathCheck fn=deny-existence bong-file=/svr/msg/go-away.html
```

find-index

Applicable in `PathCheck`-class directives.

The `find-index` function investigates whether the requested path is a directory. If it is, the function searches for an index file in the directory, and then changes the path to point to the index file. If no index file is found, the server generates a directory listing.

Note that if the file `obj.conf` has a `NameTrans` directive that calls `home-page`, and the requested directory is the root directory, then the home page rather than the index page, is returned to the client.

The `find-index` function does nothing if there is a query string, if the HTTP method is not GET, or if the path is that of a valid file.

Parameters

`index-names`	is a comma-separated list of index file names to look for. Use spaces only if they are part of a file name. Do not include spaces before or after the commas. This list is case-sensitive if the file system is case-sensitive.
`bucket`	optional, common to all `obj.conf` functions

Examples

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

find-links

Applicable in `PathCheck`-class directives.

Unix Only. The `find-links` function searches the current path for symbolic or hard links to other directories or file systems. If any are found, an error is returned. This function is normally used for directories that are not trusted (such as user home directories). It prevents someone from pointing to information that should not be made public.

Parameters

`disable`	is a character string of links to disable: • `h` is hard links • `s` is soft links • `o` allows symbolic links from user home directories only if the user owns the target of the link.
`dir`	is the directory to begin checking. If you specify an absolute path, any request to that path and its subdirectories is checked for symbolic links. If you specify a partial path, any request containing that partial path is checked for symbolic links. For example, if you use `/user/` and a request comes in for `some/user/directory`, then that directory is checked for symbolic links.
`bucket`	optional, common to all `obj.conf` functions
`checkFileExistence`	check linked file for existence and abort request with `403 (forbidden)` if this check fails.

Examples

```
PathCheck fn=find-links disable=sh dir=/foreign-dir
PathCheck fn=find-links disable=so dir=public_html
```

See Also

`init-uhome`, `unix-home`

find-pathinfo

Applicable in PathCheck-class directives.

The find-pathinfo function finds any extra path information after the file name in the URL and stores it for use in the CGI environment variable PATH_INFO.

Parameters

bucket	optional, common to all obj.conf functions

Examples

```
PathCheck fn=find-pathinfo
PathCheck fn=find-pathinfo find-pathinfo-forward=""
```

get-client-cert

Applicable in PathCheck-class directives.

The get-client-cert function gets the authenticated client certificate from the SSL3 session. It can apply to all HTTP methods, or only to those that match a specified pattern. It only works when SSL is enabled on the server.

If the certificate is present or obtained from the SSL3 session, the function returns REQ_NOACTION, allowing the request to proceed, otherwise it returns REQ_ABORTED and sets the protocol status to 403 FORBIDDEN, causing the request to fail and the client to be given the FORBIDDEN status.

Parameters

`dorequest` controls whether to actually try to get the certificate, or just test for its presence. If `dorequest` is absent the default value is `0`.

- 1 tells the function to redo the SSL3 handshake to get a client certificate, if the server does not already have the client certificate. This typically causes the client to present a dialog box to the user to select a client certificate. The server may already have the client certificate if it was requested on the initial handshake, or if a cached SSL session has been resumed.
- 0 tells the function not to redo the SSL3 handshake if the server does not already have the client certificate.

If a certificate is obtained from the client and verified successfully by the server, the ASCII base64 encoding of the DER-encoded X.509 certificate is placed in the parameter `auth-cert` in the `Request->vars` pblock, and the function returns `REQ_PROCEED`, allowing the request to proceed.

`require` controls whether failure to get a client certificate will abort the HTTP request. If `require` is absent the default value is 1.

- 1 tells the function to abort the HTTP request if the client certificate is not present after `dorequest` is handled. In this case, the HTTP status is set to `PROTOCOL_FORBIDDEN`, and the function returns `REQ_ABORTED`.
- 0 tells the function to return `REQ_NOACTION` if the client certificate is not present after dorequest is handled.

`method` (optional) specifies a wildcard pattern for the HTTP methods for which the function will be applied. If `method` is absent, the function is applied to all requests.

`bucket` optional, common to all `obj.conf` functions

Examples

```
# Get the client certificate from the session.
# If a certificate is not already associated with the
# session, request one.
# The request fails if the client does not present a
# valid certificate.

PathCheck fn="get-client-cert" dorequest="1"
```

load-config

Applicable in PathCheck-class directives.

The load-config function searches for configuration files in document directories and adds the file's contents to the server's existing configuration. These configuration files (also known as dynamic configuration files) specify additional access control information for the requested resource. Depending on the rules in the dynamic configuration files, the server might or might not allow the client to access the requested resource.

Each directive that invokes load-config is associated with a base directory, which is either stated explicitly through the basedir parameter or derived from the root directory for the requested resource. The base directory determines two things:

- the top-most directory for which requests will invoke this call to the load-config function.

  For example, if the base directory is D:/Netscape/Server4/docs/nikki/, then only requests for resources in this directory or its subdirectories (and their subdirectories and so on) trigger the search for dynamic configuration files. A request for the resource D:/Netscape/Server4/docs/somefile.html does not trigger the search in this case, since the requested resource is in a parent directory of the base directory.

- the top-most directory in which the server looks for dynamic configuration files to apply to the requested resource.

  If the base directory is D:/Netscape/Server4/docs/nikki/, the server starts its search for dynamic configuration files in this directory. It may or may not also search subdirectories (but never parent directories) depending on other factors.

When you enable dynamic configuration files through the Server Manager interface, the system writes additional objects with ppath parameters into the obj.conf file. If you manually add directives that invoke load-config to the default object (rather than putting them in separate objects), the Server Manager interface might not reflect your changes.

If you manually add PathCheck directives that invoke load-config to the file obj.conf, put them in additional objects (created with the <OBJECT> tag) rather than putting them in the default object. Use the ppath attribute of the OBJECT tag to specify the partial pathname for the resources to be affected by the access rules in the dynamic configuration file. The partial pathname can be any pathname that matches a pattern, which can include wildcard characters.

For example, the following `<OBJECT>` tag specifies that requests for resources in the directory `D:/Netscape/Server4/docs` are subject to the access rules in the file `my.nsconfig`.

```
<Object ppath="D:/Netscape/Server4/docs/*">
PathCheck fn="load-config" file="my.nsconfig" descend=1
basedir="D:/Netscape/Server4/docs"
</Object>
```

NOTE If the `ppath` resolves to a resource or directory that is higher in the directory tree (or is in a different branch of the tree) than the base directory, the `load-config` function is not invoked. This is because the base directory specifies the highest-level directory for which requests will invoke the `load-config` function.

The `load-config` function returns `REQ_PROCEED` if configuration files were loaded, `REQ_ABORTED` on error, or `REQ_NOACTION` when no files are loaded.

Parameters

`file`	(optional) is the name of the dynamic configuration file containing the access rules to be applied to the requested resource. If not provided, the file name is assumed to be `.nsconfig`.
`disable-types`	(optional) specifies a wildcard pattern of types to disable for the base directory, such as `magnus-internal/cgi`. Requests for resources matching these types are aborted.
`descend`	(optional) if present, specifies that the server should search in subdirectories of this directory for dynamic configuration files. For example, `descend=1` specifies that the server should search subdirectories. No `descend` parameter specifies that the function should search only the base directory.

`basedir`	(optional) specifies base directory. This is the highest-level directory for which requests will invoke the `load-config` function and is also the directory where the server starts searching for configuration files. If `basedir` is not specified, the base directory is assumed to be the root directory that results from translating the requested resource's URL to a physical pathname. For example, if the request was for `http://`*server-name*`/a/b/file.html`, the physical file name would be `/`*document-root*`/a/b/file.html`.
`bucket`	optional, common to all `obj.conf` functions

Examples

In this example, whenever the server receives a request for any resource containing the substring `secret` that resides in `D:/Netscape/Server4/docs/nikki/` or a subdirectory thereof, it searches for a configuration file called `checkaccess.nsconfig`.

The server starts the search in the directory `D:/Netscape/Server4/docs/nikki`, and searches subdirectories too. It loads each instance of `checkaccess.nsconfig` that it finds, applying the access control rules contained therein to determine whether the client is allowed to access the requested resource or not.

```
<Object ppath="*secret*">
PathCheck fn="load-config" file="checkaccess.nsconfig"
basedir="D:/Netscape/Server4/docs/nikki" descend="1"
</Object>
```

nt-uri-clean

Applicable in `PathCheck`-class directives.

Windows NT Only. The `nt-uri-clean` function denies access to any resource whose physical path contains `\.\`, `\..\` or `\\` (these are potential security problems).

Parameters

`bucket`	optional, common to all `obj.conf` functions
`tildeok`	if present, allows tilde"~" characters in URIs. This is a potential security risk on the NT platform, where `longfi~1.htm` might reference `longfilename.htm` but does not go through the proper ACL checking. If present, "`//`" sequences are allowed.
`dotdirok`	If present, "`//`" sequences are allowed.

Examples

```
PathCheck fn=nt-uri-clean
```

See Also

`unix-uri-clean`

ntcgicheck

Applicable in `PathCheck`-class directives.

Windows NT Only. The `ntcgicheck` function specifies the file name extension to be added to any file name that does not have an extension, or to be substituted for any file name that has the extension `.cgi`.

Parameters

`extension`	is the replacement file extension.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
PathCheck fn=ntcgicheck extension=pl
```

See Also

`init-cgi`, `send-cgi`, `send-wincgi`, `send-shellcgi`

require-auth

Applicable in `PathCheck`-class directives.

The `require-auth` function allows access to resources only if the user or group is authorized. Before this function is called, an authorization function (such as `basic-auth`) must be called in an `AuthTrans` directive.

If a user was authorized in an `AuthTrans` directive, and the `auth-user` parameter is provided, then the user's name must match the `auth-user` wildcard value. Also, if the `auth-group` parameter is provided, the authorized user must belong to an authorized group which must match the `auth-user` wildcard value.

Parameters

`path`	(optional) is a wildcard local file system path on which this function should operate. If no path is provided, the function applies to all paths.
`auth-type`	is the type of HTTP authorization used and must match the auth-type from the previous authorization function in AuthTrans. Currently, `basic` is the only authorization type defined.
`realm`	is a string sent to the browser indicating the secure area (or realm) for which a user name and password are requested.
`auth-user`	(optional) specifies a wildcard list of users who are allowed access. If this parameter is not provided, then any user authorized by the authorization function is allowed access.
`auth-group`	(optional) specifies a wildcard list of groups that are allowed access.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
PathCheck fn=require-auth auth-type=basic realm="Marketing Plans"
auth-group=mktg auth-user=(jdoe|johnd|janed)
```

See Also

`basic-auth, basic-ncsa`

set-virtual-index

Applicable in `PathCheck`-class directives.

The `set-virtual-index` function specifies a virtual index for a directory, which determines the URL forwarding. The index can refer to a LiveWire application, a servlet in its own namespace, a Netscape Application Server applogic, and so on.

`REQ_NOACTION` is returned if none of the URIs listed in the from parameter match the current URI. `REQ_ABORTED` is returned if the file specified by the `virtual-index` parameter is missing or if the current URI cannot be found. `REQ_RESTART` is returned if the current URI matches any one of the URIs mentioned in the `from` parameter or if there is no `from` parameter.

Parameters

`virtual-index`	is the URI of the content generator that acts as an index for the URI the user enters.
`from`	(optional) is a comma-separated list of URIs for which this `virtual-index` is applicable. If `from` is not specified, the `virtual-index` always applies.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
# MyLWApp is a LiveWire application
PathCheck fn=set-virtual-index virtual-index=MyLWApp
```

ssl-check

Applicable in `PathCheck`-class directives.

If a restriction is selected that is not consistent with the current cipher settings under Security Preferences, this function opens a popup dialog which warns that ciphers with larger secret keysizes need to be enabled. This function is designed to be used together with a Client tag to limit access of certain directories to non-exportable browsers.

The function returns REQ_NOACTION if SSL is not enabled, or if the secret-keysize parameter is not specified. If the secret keysize for the current session is less than the specified secret-keysize and the bong-file parameter is not specified, the function returns REQ_ABORTED with a status of PROTOCOL_FORBIDDEN. If the bong file is specified, the function returns REQ_PROCEED, and the path variable is set to the bong filename. Also, when a keysize restriction is not met, the SSL session cache entry for the current session is invalidated, so that a full SSL handshake will occur the next time the same client connects to the server.

Requests that use ssl-check are not cacheable in the accelerator file cache if ssl-check returns something other than REQ_NOACTION.

Parameters

secret-keysize	(optional) is the minimum number of bits required in the secret key.
bong-file	(optional) is the name of a file (not a URI) to be served if the restriction is not met
bucket	optional, common to all obj.conf functions

ssl-logout

Applicable in PathCheck-class directives.

ssl-logout invalidates the current SSL session in the server's SSL session cache. This does not affect the current request, but the next time the client connects, a new SSL session will be created. If SSL is enabled, this function returns REQ_PROCEED after invalidating the session cache entry. If SSL is not enabled, it returns REQ_NOACTION.

Parameters

bucket	optional, common to all obj.conf functions

unix-uri-clean

Applicable in PathCheck-class directives.

Unix Only. The unix-uri-clean function denies access to any resource whose physical path contains /./, /../ or // (these are potential security problems).

Parameters

bucket	optional, common to all obj.conf functions
dotdirok	If present, "//" sequences are allowed.

Examples

```
PathCheck fn=unix-uri-clean
```

See Also

nt-uri-clean

ObjectType Stage

ObjectType directives determine the MIME type of the file to send to the client in response to a request. MIME attributes currently sent are type, encoding, and language. The MIME type sent to the client as the value of the content-type header.

ObjectType directives also set the type parameter, which is used by Service directives to determine how to process the request according to what kind of content is being requested.

If there is more than one ObjectType directive in an object, all the directives are applied in the order they appear. If a directive sets an attribute and later directives try to set that attribute to something else, the first setting is used and the subsequent ones ignored.

The obj.conf file almost always has an ObjectType directive that calls the type-by-extension function. This function instructs the server to look in a particular file (the MIME types file) to deduce the content type from the extension of the requested resource.

The following ObjectType-class functions are described in detail in this section:

- force-type sets the content-type header for the response to a specific type.
- set-default-type allows you to define a default charset, content-encoding, and content-language for the response being sent back to the client.
- shtml-hacktype requests that .htm and .html files are parsed for server-parsed html commands.
- type-by-exp sets the content-type header for the response based on the requested path.
- type-by-extension sets the content-type header for the response based on the files extension and the MIME types database.

force-type

Applicable in ObjectType-class directives.

The force-type function assigns a type to requests that do not already have a MIME type. This is used to specify a default object type.

Make sure that the directive that calls this function comes last in the list of ObjectType directives so that all other ObjectType directives have a chance to set the MIME type first. If there is more than one ObjectType directive in an object, all the directives are applied in the order they appear. If a directive sets an attribute and later directives try to set that attribute to something else, the first setting is used and the subsequent ones ignored.

Parameters

type	(optional) is the type assigned to a matching request (the content-type header).
enc	(optional) is the encoding assigned to a matching request (the content-encoding header).
lang	(optional) is the language assigned to a matching request (the content-language header).
charset	(optional) is the character set for the magnus-charset parameter in rq->srvhdrs. If the browser sent the Accept-charset header or its User-agent is mozilla / 1.1 or newer, then append "; charset=*charset*" to content-type, where *charset* is the value of the magnus-charset parameter in rq->srvhdrs.

`bucket`	optional, common to all `obj.conf` functions

Examples

```
ObjectType fn=force-type type=text/plain
ObjectType fn=force-type lang=en_US
```

See Also

`type-by-extension`, `type-by-exp`

set-default-type

Applicable in `ObjectType`-class directives.

This function allows you to define a default `charset`, `content-encoding`, and `content-language` for the response being sent back to the client.

If the `charset`, `content-encoding`, and `content-language` have not been set for a response, then just before the headers are sent the defaults defined by `set-default-type` are used. Note that by placing this function in different objects in `obj.conf`, you can define different defaults for different parts of the document tree.

Parameters

`enc`	(optional) is the encoding assigned to a matching request (the `content-encoding` header).
`lang`	(optional) is the language assigned to a matching request (the `content-language` header).
`charset`	(optional) is the character set for the `magnus-charset` parameter in `rq->srvhdrs`. If the browser sent the `Accept-charset` header or its `User-agent` is mozilla/1.1 or newer, then append "`; charset=`*charset*" to content-type, where *charset* is the value of the `magnus-charset` parameter in `rq->srvhdrs`.
`bucket`	optional, common to all `obj.conf` functions

Example

```
ObjectType fn="set-default-type" charset="iso_8859-1"
```

shtml-hacktype

Applicable in `ObjectType`-class directives.

The `shtml-hacktype` function changes the content-type of any `.htm` or `.html` file to `magnus-internal/parsed-html` and returns `REQ_PROCEED`. This provides backward compatibility with server-side includes for files with `.htm` or `.html` extensions. The function may also check the execute bit for the file on Unix systems. The use of this function is not recommended.

Parameters

`exec-hack`	(Unix only, optional) tells the function to change the content-type only if the execute bit is enabled. The value of the parameter is not important. It need only be provided. You may use `exec-hack=true`.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
ObjectType fn=shtml-hacktype exec-hack=true
```

type-by-exp

Applicable in `ObjectType`-class directives.

The `type-by-exp` function matches the current path with a wildcard expression. If the two match, the `type` parameter information is applied to the file. This is the same as `type-by-extension`, except you use wildcard patterns for the files or directories specified in the URLs.

Parameters

`exp`	is the wildcard pattern of paths for which this function is applied.
`type`	(optional) is the type assigned to a matching request (the `content-type` header).
`enc`	(optional) is the encoding assigned to a matching request (the `content-encoding` header).
`lang`	(optional) is the language assigned to a matching request (the `content-language` header).
`charset`	(optional) is the character set for the `magnus-charset` parameter in `rq->srvhdrs`. If the browser sent the `Accept-charset` header or its `User-agent` is mozilla / 1.1 or newer, then append "`; charset=`*charset*" to content-type, where *charset* is the value of the `magnus-charset` parameter in `rq->srvhdrs`.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
ObjectType fn=type-by-exp exp=*.test type=application/html
```

See Also

`type-by-extension`, `force-type`

type-by-extension

Applicable in `ObjectType`-class directives.

This function instructs the server to look in a table of MIME type mappings to find the MIME type of the requested resource according to the extension of the requested resource. The MIME type is added to the `content-type` header sent back to the client.

The table of MIME type mappings is created by a `MIME` element in the `server.xml` file, which loads a MIME types file or list and creates the mappings. For more information about `server.xml`, see Chapter 8, "Virtual Server Configuration Files." For more information about MIME types files, see Appendix B, "MIME Types."

For example, the following two lines are part of a MIME types file:

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

If the extension of the requested resource is `htm` or `html`, the `type-by-extension` file sets the type to `text/html`. If the extension is `.txt`, the function sets the type to `text/plain`.

Parameters

`bucket`	optional, common to all `obj.conf` functions

Examples

```
ObjectType fn=type-by-extension
```

See Also

`type-by-exp, force-type`

Service Stage

The `Service` class of functions sends the response data to the client.

Every `Service` directive has the following optional parameters to determine whether the function is executed. All the optional parameters must match the current request for the function to be executed.

- `type`

(optional) specifies a wildcard pattern of MIME types for which this function will be executed. The `magnus-internal/*` MIME types are used only to select a `Service` function to execute.

- `method`

  (optional) specifies a wildcard pattern of HTTP methods for which this function will be executed. Common HTTP methods are `GET`, `HEAD`, and `POST`.

- `query`

  (optional) specifies a wildcard pattern of query strings for which this function will be executed.

- `UseOutputStreamSize`

  (optional) determines the default output stream buffer size, in bytes, for data sent to the client. If this parameter is not specified, the default is `8192` bytes.

NOTE The `UseOutputStreamSize` parameter can be set to zero in the `obj.conf` file to disable output stream buffering. For the `magnus.conf` file, setting `UseOutputStreamSize` to zero has no effect.

- `flushTimer`

  (optional) determines the maximum number of milliseconds between write operations in which buffering is enabled. If the interval between subsequent write operations is greater than the `flushTimer` value for an application, further buffering is disabled. This is necessary for status monitoring CGI applications that run continuously and generate periodic status update reports. If this parameter is not specified, the default is `3000` milliseconds.

- `ChunkedRequestBufferSize`

  (optional) determines the default buffer size, in bytes, for “un-chunking” request data. If this parameter is not specified, the default is `8192` bytes.

- `ChunkedRequestTimeout`

  (optional) determines the default timeout, in seconds, for “un-chunking” request data. If this parameter is not specified, the default is `60` seconds.

If there is more than one `Service`-class function, the first one matching the optional wildcard parameters (`type`, `method`, and `query`) is executed.

For more information about the `UseOutputStreamSize`, `flushTimer`, `ChunkedRequestBufferSize`, and `ChunkedRequestTimeout` parameters, see “Buffered Streams,” on page 324. The `UseOutputStreamSize`, `ChunkedRequestBufferSize`, and `ChunkedRequestTimeout` parameters also have equivalent `magnus.conf` directives; see “Chunked Encoding,” on page 281. The `obj.conf` parameters override the `magnus.conf` directives.

By default, the server sends the requested file to the client by calling the `send-file` function. The directive that sets the default is:

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

This directive usually comes last in the set of `Service`-class directives to give all other Service directives a chance to be invoked. This directive is invoked if the method of the request is `GET`, `HEAD`, or `POST`, and the type does *not* start with `magnus-internal/`. Note here that the pattern `*~` means “does not match.” For a list of characters that can be used in patterns, see Appendix C, “Wildcard Patterns.”

The following Service-class functions are described in detail in this section:

- `add-footer` appends a footer specified by a filename or URL to a an HTML file.
- `add-header` prepends a header specified by a filename or URL to an HTML file.
- `append-trailer` appends text to the end of an HTML file.
- `imagemap` handles server-side image maps.
- `index-common` generates a fancy list of the files and directories in a requested directory.
- `index-simple` generates a simple list of files and directories in a requested directory.
- `key-toosmall` indicates to the client that the provided certificate key size is too small to accept.
- `list-dir` lists the contents of a directory.
- `make-dir` creates a directory.
- `query-handler` handles the HTML ISINDEX tag.
- `remove-dir` deletes an empty directory.
- `remove-file` deletes a file.
- `rename-file` renames a file.

- `send-cgi` sets up environment variables, launches a CGI program, and sends the response to the client.
- `send-file` sends a local file to the client.
- `send-range` sends a range of bytes of a file to the client.
- `send-shellcgi` sets up environment variables, launches a shell CGI program, and sends the response to the client.
- `send-wincgi` sets up environment variables, launches a WinCGI program, and sends the response to the client.
- `service-dump` creates a performance report based on collected performance bucket data.
- `shtml_send` parses an HTML file for server-parsed html commands.
- `stats-xml` creates a performance report in XML format.
- `upload-file` uploads and saves a file.

add-footer

Applicable in `Service`-class directives.

This function appends a footer to an HTML file that is sent to the client. The footer is specified either as a filename or a URI -- thus the footer can be dynamically generated. To specify static text as a footer, use the `append-trailer` function.

Parameters

`file`	(optional) The pathname to the file containing the footer. Specify either `file` or `uri`. By default the pathname is relative. If the pathname is absolute, pass the `NSIntAbsFilePath` parameter as `yes`.
`uri`	(optional) A URI pointing to the resource containing the footer. Specify either `file` or `uri`.
`NSIntAbsFilePath`	(optional) if the file parameter is specified, the `NSIntAbsFilePath` parameter determines whether the file name is absolute or relative. The default is relative. Set the value to `yes` to indicate an absolute file path.
`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions

`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service type=text/html method=GET fn=add-footer
file="footers/footer1.html"
Service type=text/html method=GET fn=add-footer
file="D:/netscape/server4/footers/footer1.html"
NSIntAbsFilePath="yes"
```

See Also

`append-trailer`, `add-header`

add-header

Applicable in `Service`-class directives.

This function prepends a header to an HTML file that is sent to the client. The header is specified either as a filename or a URI -- thus the header can be dynamically generated.

Parameters

`file`	(optional) The pathname to the file containing the header. Specify either `file` or `uri`. By default the pathname is relative. If the pathname is absolute, pass the `NSIntAbsFilePath` parameter as `yes`.
`uri`	(optional) A URI pointing to the resource containing the header. Specify either `file` or `uri`.

NSIntAbsFilePath	(optional) if the file parameter is specified, the NSIntAbsFilePath parameter determines whether the file name is absolute or relative. The default is relative. Set the value to yes to indicate an absolute file path.
type	optional, common to all Service-class functions
method	optional, common to all Service-class functions
query	optional, common to all Service-class functions
UseOutputStreamSize	optional, common to all Service-class functions
flushTimer	optional, common to all Service-class functions
ChunkedRequestBufferSize	optional, common to all Service-class functions
ChunkedRequestTimeout	optional, common to all Service-class functions
bucket	optional, common to all obj.conf functions

Examples

```
Service type=text/html method=GET fn=add-header
file="headers/header1.html"
Service type=text/html method=GET fn=add-footer
file="D:/netscape/server4/headers/header1.html"
NSIntAbsFilePath="yes"
```

See Also

add-footer, append-trailer

append-trailer

Applicable in Service-class directives.

The append-trailer function sends an HTML file and appends text to the end. It only appends text to HTML files. This is typically used for author information and copyright text. The date the file was last modified can be inserted.

Returns REQ_ABORTED if a required parameter is missing, if there is extra path information after the file name in the URL, or if the file cannot be opened for read-only access.

Parameters

`trailer`	is the text to append to HTML documents. The string is unescaped with `util_uri_unescape` before being sent. The text can contain HTML tags and can be up to 512 characters long after unescaping and inserting the date. If you use the string `:LASTMOD:`, which is replaced by the date the file was last modified; you must also specify a time format with `timefmt`.
`timefmt`	(optional) is a time format string for `:LASTMOD:`. For details about time formats refer to Appendix D, "Time Formats." If `timefmt` is not provided, `:LASTMOD:` will not be replaced with the time.
`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service type=text/html method=GET fn=append-trailer
trailer="<hr><img src=/logo.gif> Copyright 1999"
# Add a trailer with the date in the format: MM/DD/YY
Service type=text/html method=GET fn=append-trailer timefmt="%D"
trailer="<HR>File last updated on: :LASTMOD:"
```

See Also

`add-footer`, `add-header`

imagemap

Applicable in `Service`-class directives.

The `imagemap` function responds to requests for imagemaps. Imagemaps are images which are divided into multiple areas that each have an associated URL. The information about which URL is associated with which area is stored in a mapping file.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service type=magnus-internal/imagemap method=(GET|HEAD)
fn=imagemap
```

index-common

Applicable in `Service`-class directives.

The `index-common` function generates a fancy (or common) list of files in the requested directory. The list is sorted alphabetically. Files beginning with a period (.) are not displayed. Each item appears as an HTML link. This function displays more information than `index-simple` including the size, date last modified, and an icon for each file. It may also include a header and/or readme file into the listing.

The `Init`-class function `cindex-init` in `magnus.conf` specifies the format for the index list, including where to look for the images.

If `obj.conf` contains a call to `index-common` in the `Service` stage, `magnus.conf` must initialize fancy (or common) indexing by invoking `cindex-init` during the `Init` stage.

Indexing occurs when the requested resource is a directory that does not contain an index file or a home page, or no index file or home page has been specified by the functions `find-index` or `home-page`.

The icons displayed are `.gif` files dependent on the `content-type` of the file:

`"text/*"`	`text.gif`
`"image/*"`	`image.gif`
`"audio/*"`	`sound.gif`
`"video/*"`	`movie.gif`
`"application/octet-stream"`	`binary.gif`
directory	`menu.gif`
all others	`unknown.gif`

Parameters

`header`	(optional) is the path (relative to the directory being indexed) and name of a file (HTML or plain text) which is included at the beginning of the directory listing to introduce the contents of the directory. The file is first tried with `.html` added to the end. If found, it is incorporated near the top of the directory list as HTML. If the file is not found, then it is tried without the `.html` and incorporated as preformatted plain text (bracketed by `<PRE>` and).
`readme`	(optional) is the path (relative to the directory being indexed) and name of a file (HTML or plain text) to append to the directory listing. This file might give more information about the contents of the directory, indicate copyrights, authors, or other information. The file is first tried with `.html` added to the end. If found, it is incorporated at the bottom of the directory list as HTML. If the file is not found, then it is tried without the `.html` and incorporated as preformatted plain text (enclosed by `<PRE>` and `</PRE>`).
`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions

`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service fn=index-common type=magnus-internal/directory
method=(GET|HEAD) header=hdr readme=rdme.txt
```

See Also

`cindex-init`, `index-simple`, `find-index`, `home-page`

index-simple

Applicable in `Service`-class directives.

The `index-simple` function generates a simple index of the files in the requested directory. It scans a directory and returns an HTML page to the browser displaying a bulleted list of the files and directories in the directory. The list is sorted alphabetically. Files beginning with a period (.) are not displayed. Each item appears as an HTML link.

Indexing occurs when the requested resource is a directory that does not contain either an index file or a home page, or no index file or home page has been specified by the functions `find-index` or `home-page`.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions

`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service type=magnus-internal/directory fn=index-simple
```

See Also

`cindex-init`, `index-common`

key-toosmall

Applicable in `Service`-class directives.

NOTE	This function is provided for backward compatibility only and was deprecated in iPlanet Web Server 4.*x*. It is replaced by the PathCheck-class SAF `ssl-check`.

The `key-toosmall` function returns a message to the client specifying that the secret key size for SSL communications is too small. This function is designed to be used together with a `Client` tag to limit access of certain directories to non-exportable browsers.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
<Object ppath=/mydocs/secret/*>
Service fn=key-toosmall
</Object>
```

list-dir

Applicable in `Service`-class directives.

The `list-dir` function returns a sequence of text lines to the client in response to a request whose method is INDEX. The format of the returned lines is:

name type size mimetype

The *name* field is the name of the file or directory. It is relative to the directory being indexed. It is URL-encoded, that is, any character might be represented by `%xx`, where `xx` is the hexadecimal representation of the character's ASCII number.

The *type* field is a MIME type such as `text/html`. Directories will be of type `directory`. A file for which the server doesn't have a type will be of type `unknown`.

The *size* field is the size of the file, in bytes.

The *mtime* field is the numerical representation of the date of last modification of the file. The number is the number of seconds since the epoch (Jan 1, 1970 00:00 UTC) since the last modification of the file.

When remote file manipulation is enabled in the server, the `obj.conf` file contains a `Service`-class function that calls `list-dir` for requests whose method is `INDEX`.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service fn=list-dir method="INDEX"
```

make-dir

Applicable in Service-class directives.

The make-dir function creates a directory when the client sends a request whose method is MKDIR. The function can fail if the server can't write to that directory.

When remote file manipulation is enabled in the server, the obj.conf file contains a Service-class function that invokes make-dir when the request method is MKDIR.

Parameters

type	optional, common to all Service-class functions
method	optional, common to all Service-class functions
query	optional, common to all Service-class functions
UseOutputStreamSize	optional, common to all Service-class functions
flushTimer	optional, common to all Service-class functions
ChunkedRequestBufferSize	optional, common to all Service-class functions
ChunkedRequestTimeout	optional, common to all Service-class functions
bucket	optional, common to all obj.conf functions

Examples

```
Service fn="make-dir" method="MKDIR"
```

query-handler

Applicable in Service-class directives.

NOTE This function is provided for backward compatibility only and is used mainly to support the obsolete ISINDEX tag. If possible, use an HTML form instead.

The `query-handler` function runs a CGI program instead of referencing the path requested.

Parameters

`path`	is the full path and file name of the CGI program to run.
`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service query=* fn=query-handler path=/http/cgi/do-grep
Service query=* fn=query-handler path=/http/cgi/proc-info
```

remove-dir

Applicable in `Service`-class directives.

The `remove-dir` function removes a directory when the client sends an request whose method is `RMDIR`. The directory must be empty (have no files in it). The function will fail if the directory is not empty or if the server doesn't have the privileges to remove the directory.

When remote file manipulation is enabled in the server, the `obj.conf` file contains a `Service`-class function that invokes `remove-dir` when the request method is `RMDIR`.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service fn="remove-dir" method="RMDIR"
```

remove-file

Applicable in `Service`-class directives.

The `remove-file` function deletes a file when the client sends a request whose method is DELETE. It deletes the file indicated by the URL if the user is authorized and the server has the needed file system privileges.

When remote file manipulation is enabled in the server, the `obj.conf` file contains a `Service`-class function that invokes `remove-file` when the request method is `DELETE`.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions

`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service fn="remove-file" method="DELETE"
```

rename-file

Applicable in `Service`-class directives.

The `rename-file` function renames a file when the client sends a request with a `New-URL` header whose method is `MOVE`. It renames the file indicated by the URL to `New-URL` within the same directory if the user is authorized and the server has the needed file system privileges.

When remote file manipulation is enabled in the server, the `obj.conf` file contains a `Service`-class function that invokes `rename-file` when the request method is `MOVE`.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service fn="rename-file" method="MOVE"
```

send-cgi

Applicable in `Service`-class directives.

The `send-cgi` function sets up the CGI environment variables, runs a file as a CGI program in a new process, and sends the results to the client.

For details about the CGI environment variables and their NSAPI equivalents, refer to section "CGI to NSAPI Conversion," on page 135.

For additional information about CGI, see the *iPlanet Web Server Administrator's Guide* and the *Programmer's Guide for iPlanet Web Server.*

There are three ways to change the timing used to flush the CGI buffer:

- Adjust the interval between flushes using the `flushTimer` parameter
- Adjust the buffer size using the `UseOutputStreamSize` parameter
- Force iPlanet Web Server to flush its buffer by forcing spaces into the buffer in the CGI script

For more information about `flushTimer` and `UseOutputStreamSize`, see "Buffered Streams," on page 324.

Parameters

`user`	(Unix only) Specifies the name of the user to execute CGI programs as.
`group`	(Unix only) Specifies the name of the group to execute CGI programs as.
`chroot`	(Unix only) Specifies the directory to chroot to before execution begins. This is relative to the `chroot` defined in `magnus.conf`.
`dir`	(Unix only) Specifies the directory to chdir to after chroot but before execution begins.

`rlimit_as`	(Unix only) Specifies the maximum CGI program address space in bytes. You can supply both current (soft) and maximum (hard) limits, separated by a comma. The soft limit must be listed first. If only one limit is specified, both limits are set to this value.
`rlimit_core`	(Unix only) Specifies the maximum CGI program core file size. A value of 0 disables writing cores. You can supply both current (soft) and maximum (hard) limits, separated by a comma. The soft limit must be listed first. If only one limit is specified, both limits are set to this value.
`rlimit_nofile`	(Unix only) Specifies the maximum number of file descriptors for the CGI program. You can supply both current (soft) and maximum (hard) limits, separated by a comma. The soft limit must be listed first. If only one limit is specified, both limits are set to this value.
`nice`	(Unix only) Accepts an increment that determines the CGI program's priority relative to the server. Typically, the server is run with a nice value of 0 and the nice increment would be between 0 (the CGI program runs at same priority as server) and 19 (the CGI program runs at much lower priority than server). While it is possible to increase the priority of the CGI program above that of the server by specifying a nice increment of -1, this is not recommended.
`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

The following example uses variables defined in the `server.xml` file for the `send-cgi` parameters. For more information about defining variables, see Chapter 8, "Virtual Server Configuration Files."

```
<Object name="default">
...
NameTrans fn="pfx2dir" from="/cgi-bin"
dir="/home/foo.com/public_html/cgi-bin" name="cgi"
...
</Object>

<Object name="cgi">
ObjectType fn="force-type" type="magnus-internal/cgi"
Service fn="send-cgi" user="$user" group="$group" dir="$dir"
chroot="$chroot" nice="$nice"
</Object>
```

send-file

Applicable in `Service`-class directives.

The `send-file` function sends the contents of the requested file to the client. It provides the `content-type`, `content-length`, and `last-modified` headers.

Most requests are handled by this function using the following directive (which usually comes last in the list of `Service`-class directives in the default object so that it acts as a default)

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

This directive is invoked if the method of the request is `GET`, `HEAD`, or `POST`, and the type does *not* start with `magnus-internal/`. Note here that the pattern `*~` means "does not match." For a list of characters that can be used in patterns, see Appendix C, "Wildcard Patterns."

Parameters

`nocache`	(optional) prevents the server from caching responses to static file requests. For example, you can specify that files in a particular directory are not to be cached, which is useful for directories where the files change frequently. The value you assign to this parameter is ignored. If you do not wish to use this parameter, leave it out.
`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions

`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

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

In the following example, the server does not cache static files from `/export/somedir/` when requested by the URL prefix `/myurl`.

```
<Object name=default>
...
NameTrans fn="pfx2dir" from="/myurl" dir="/export/mydir",
name="myname"
...
Service method=(GET|HEAD|POST) type=*~magnus-internal/*
fn=send-file
...
</Object>
<Object name="myname">
Service method=(GET|HEAD) type=*~magnus-internal/* fn=send-file
nocache=""
</Object>
```

send-range

Applicable in `Service`-class directives.

When the client requests a portion of a document, by specifying HTTP byte ranges, the `send-range` function returns that portion.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service fn=send-range
```

send-shellcgi

Applicable in `Service`-class directives.

Windows NT only. The `send-shellcgi` function runs a file as a shell CGI program and sends the results to the client. Shell CGI is a server configuration that lets you run CGI applications using the file associations set in Windows NT. For information about shell CGI programs, consult the *iPlanet Web Server Administrator's Guide*.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions

`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service fn=send-shellcgi
Service type=magnus-internal/cgi fn=send-shellcgi
```

send-wincgi

Applicable in `Service`-class directives.

Windows NT only. The `send-wincgi` function runs a file as a Windows CGI program and sends the results to the client. For information about Windows CGI programs, consult the *iPlanet Web Server Administrator's Guide*.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service fn=send-wincgi
Service type=magnus-internal/cgi fn=send-wincgi
```

service-dump

Applicable in `Service`-class directives.

The `service-dump` function creates a performance report based on collected performance bucket data (see “The bucket Parameter,” on page 49”).

To read the report, point the browser here:

`http://`*server_id*`:`*port*`/.perf`

Parameters

`type`	must be `perf` for this function
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
<Object name=default>
NameTrans fn="assign-name" from="/.perf" name="perf"
...
</Object>

<Object name=perf>
Service fn="service-dump"
</Object>
```

See Also

`stats-xml`

shtml_send

Applicable in `Service`-class directives.

The `shtml_send` function parses an HTML document, scanning for embedded commands. These commands may provide information from the server, include the contents of other files, or execute a CGI program. The `shtml_send` function is only available when the Shtml plugin (`libShtml.so` on Unix `libShtml.dll` on Windows NT) is loaded. Refer to the *Programmer's Guide for iPlanet Web Server* for server-parsed HTML commands.

Parameters

`ShtmlMaxDepth`	maximum depth of include nesting allowed. The default value is 10.
`addCgiInitVars`	(Unix only) if present and equal to `yes` (the default is `no`), adds the environment variables defined in the `init-cgi` SAF to the environment of any command executed through the SHTML `exec` tag.
`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service type=magnus-internal/shtml_send method=(GET|HEAD)
fn=shtml_send
```

stats-xml

Applicable in `Service`-class directives.

The `stats-xml` function creates a performance report in XML format. If performance buckets have been defined, this performance report includes them.

However, you do need to initialize this function using the `stats-init` function in `magnus.conf`, then use a `NameTrans` function to direct requests to the `stats-xml` function. See the examples below.

The report is generated here:

`http://`*server_id*`:`*port*`/stats-xml/iwsstats.xml`

The associated DTD file is here:

`http://`*server_id*`:`*port*`/stats-xml/iwsstats.dtd`

For more information about the format of the `iwsstats.xml` file, see the *Performance Tuning, Sizing, and Scaling Guide for iPlanet Web Server*.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

in `magnus.conf`:

```
Init fn="stats-init" update-interval="5" virtual-servers="2000"
profiling="yes"
```

in `obj.conf`:

```
<Object name="default">
...
NameTrans fn="assign-name" from="/stats-xml/*" name="stats-xml"
...
</Object>
...
<Object name="stats-xml">
Service fn="stats-xml"
</Object>
```

See Also

`service-dump`, `stats-init`

upload-file

Applicable in `Service`-class directives.

The `upload-file` function uploads and saves a new file when the client sends a request whose method is `PUT` if the user is authorized and the server has the needed file system privileges.

When remote file manipulation is enabled in the server, the `obj.conf` file contains a `Service`-class function that invokes `upload-file` when the request method is `PUT`.

Parameters

`type`	optional, common to all Service-class functions
`method`	optional, common to all Service-class functions
`query`	optional, common to all Service-class functions
`UseOutputStreamSize`	optional, common to all Service-class functions
`flushTimer`	optional, common to all Service-class functions
`ChunkedRequestBufferSize`	optional, common to all Service-class functions
`ChunkedRequestTimeout`	optional, common to all Service-class functions
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Service fn=upload-file
```

AddLog Stage

After the server has responded to the request, the AddLog directives are executed to record information about the transaction.

If there is more than one `AddLog` directive, all are executed.

The following AddLog-class functions are described in detail in this section:

- `common-log` records information about the request in the common log format.
- `flex-log` records information about the request in a flexible, configurable format.
- `record-useragent` records the client's ip address and user-agent header.

common-log

Applicable in `AddLog`-class directives.

This function records request-specific data in the common log format (used by most HTTP servers). There is a log analyzer in the `/extras/log_anly` directory for iPlanet Web Server.

The common log must have been initialized previously by the `init-clf` function. For information about rotating logs, see `flex-rotate-init`.

There are also a number of free statistics generators for the common log format.

Parameters

`name`	(optional) gives the name of a log file, which must have been given as a parameter to the `init-clf` function in `magnus.conf`. If no name is given, the entry is recorded in the global log file.
`iponly`	(optional) instructs the server to log the IP address of the remote client rather than looking up and logging the DNS name. This will improve performance if DNS is off in the `magnus.conf` file. The value of `iponly` has no significance, as long as it exists; you may use `iponly=1`.

`bucket`	optional, common to all `obj.conf` functions

Examples

```
# Log all accesses to the global log file
AddLog fn=common-log
# Log accesses from outside our subnet (198.93.5.*) to
# nonlocallog
<Client ip="*~198.93.5.*">
AddLog fn=common-log name=nonlocallog
</Client>
```

See Also

`flex-init`, `init-clf`, `record-useragent`, `flex-log`, `flex-rotate-init`

flex-log

Applicable in `AddLog`-class directives.

This function records request-specific data in a flexible log format. It may also record requests in the common log format. There is a log analyzer in the `/extras/flexanlg` directory for iPlanet Web Server.

There are also a number of free statistics generators for the common log format.

The log format is specified by the `flex-init` function call. For information about rotating logs, see `flex-rotate-init`.

Parameters

`name`	(optional) gives the name of a log file, which must have been given as a parameter to the `flex-init` function in `magnus.conf`. If no name is given, the entry is recorded in the global log file.
`iponly`	(optional) instructs the server to log the IP address of the remote client rather than looking up and logging the DNS name. This will improve performance if DNS is off in the `magnus.conf` file. The value of `iponly` has no significance, as long as it exists; you may use `iponly=1`.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
# Log all accesses to the global log file
AddLog fn=flex-log
# Log accesses from outside our subnet (198.93.5.*) to
# nonlocallog
<Client ip="*~198.93.5.*">
AddLog fn=flex-log name=nonlocallog
</Client>
```

See Also

`flex-init`, `init-clf`, `common-log`, `record-useragent`, `flex-rotate-init`

record-useragent

Applicable in `AddLog`-class directives.

The `record-useragent` function records the IP address of the client, followed by its User-Agent HTTP header. This indicates what version of Netscape Navigator (or other client) was used for this transaction.

For information about rotating logs, see `flex-rotate-init`.

Parameters

`name`	(optional) gives the name of a log file, which must have been given as a parameter to the `init-clf` function in `magnus.conf`. If no name is given, the entry is recorded in the global log file.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
# Record the client ip address and user-agent to browserlog
AddLog fn=record-useragent name=browserlog
```

See Also

flex-init, init-clf, common-log, flex-log, flex-rotate-init

Error Stage

If a server application function results in an error, it sets the HTTP response status code and returns the value REQ_ABORTED. When this happens, the server stops processing the request. Instead, it searches for an Error directive matching the HTTP response status code or its associated reason phrase, and executes the directive's function. If the server does not find a matching Error directive, it returns the response status code to the client.

The following Error-class functions are described in detail in this section:

- send-error sends an HTML file to the client in place of a specific HTTP response status.
- qos-error returns an error page stating which quality of service limits caused the error and what the value of the QOS statistic was.

send-error

Applicable in Error-class directives.

The send-error function sends an HTML file to the client in place of a specific HTTP response status. This allows the server to present a friendly message describing the problem. The HTML page may contain images and links to the server's home page or other pages.

Parameters

path	specifies the full file system path of an HTML file to send to the client. The file is sent as text/html regardless of its name or actual type. If the file does not exist, the server sends a simple default error page.
reason	(optional) is the text of one of the reason strings (such as "Unauthorized" or "Forbidden"). The string is not case sensitive.

`code`	(optional) is a three-digit number representing the HTTP response status code, such as 401 or 407. This can be any HTTP response status code or reason phrase according to the HTTP specification. The following is a list of common HTTP response status codes and reason strings. • `401 Unauthorized`. • `403 Forbidden`. • `404 Not Found`. • `500 Server Error`.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Error fn=send-error code=401
path=/netscape/server4/docs/errors/401.html
```

qos-error

Applicable in `Error`-class directives.

The `qos-error` function returns an error page stating which quality of service limits caused the error and what the value of the QOS statistic was.

The code for this SAF is one of the examples in Chapter 6, "Examples of Custom SAFs."

For more information, see the performance chapter of the *iPlanet Web Server Administrator's Guide*.

Parameters

`code`	(optional) is a three-digit number representing the HTTP response status code, such as 401 or 407. The recommended value is 503. This can be any HTTP response status code or reason phrase according to the HTTP specification. The following is a list of common HTTP response status codes and reason strings. • `401 Unauthorized`. • `403 Forbidden`. • `404 Not Found`. • `500 Server Error`.
`bucket`	optional, common to all `obj.conf` functions

Examples

```
Error fn=qos-error code=503
```

See Also

`qos-handler`

Chapter 4

Creating Custom SAFs

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 iPlanet 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.

The sections in this chapter are:

- 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 Chapter 1, "Basics of Server Operation." Also, before writing a custom SAF, check if a built-in SAF already accomplishes the tasks you have in mind.

See Chapter 7, "Syntax and Use of magnus.conf," for a list of the pre-defined Init SAFs. See Chapter 3, "Predefined SAFs and the Request Handling Process," for a list of the rest of the pre-defined SAFs.

For a complete list of the NSAPI routines for implementing custom SAFs, see Chapter 5, "NSAPI Function Reference."

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:

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

The next section discusses the parameters in detail.

The SAF returns a result code which 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 the section "Result Codes," on page 119 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.
- `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-nsca` function might look like:

```
AuthTrans fn=basic-ncsa auth-type=basic
dbm=/netscape/server4/userdb/rs
```

In this case, the `pb` parameter passed to `basic-ncsa` contains name/value pairs that correspond to `auth-type=basic` and `dbm=/netscape/server4/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 128 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 5, "NSAPI Function Reference" for information about NSAPI routines for manipulating the `Session` data structure):

- `sn->client`

  is a 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`

  is a 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 5, “NSAPI Function Reference,” for information about NSAPI routines for manipulating the `Request` data structure).

- `rq->vars`

  is a 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`

  is a pointer to a `pblock` containing elements of the HTTP request. This includes the HTTP method (GET, POST, ...), 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`

  is a pointer to a `pblock` containing all the request headers (such as User-Agent, If-Modified-Since, ...) received from the client in the HTTP request. See Appendix E, “HyperText Transfer Protocol,” for more information about request headers. This `pblock` does not normally change throughout the request-response process.

- `rq->srvhdrs`

  is a pointer to a `pblock` containing the response headers (such as Server, Date, Content-type, Content-length,...) to be sent to the client in the HTTP response. See Appendix E, “HyperText Transfer Protocol” for more information about response headers.

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` which were previously inserted by an 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 (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 (PathCheck, ObjectType, and AddLog), the server proceeds to the next SAF in the current step.

- `REQ_NOACTION`

  indicates 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 `REQ_ABORTED` should also set the HTTP response status code. If the server finds an `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 `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 `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 `obj.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. Reconfigure 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 the section "Overview of NSAPI C Functions," on page 127. Chapter 5, "NSAPI Function Reference," provides information about all of the routines available.

For examples of custom SAFs, see `nsapi/examples/` in the server root directory and also see Chapter 6, "Examples of Custom SAFs."

The signature for all SAFs is:

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

For more details on the parameters, see the section "SAF Parameters," on page 116.

The iPlanet 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 the HTTP requests.

Keep these things 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. Also log them so that 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:

```
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 128.

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 do particular things, 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 the section "Required Behavior of SAFs for Each Directive," on page 132.

Compile and Link

Compile and link your code with the native compiler for the target platform. For UNIX, use the `gmake` command. For Windows NT, use the `nmake` command. For Windows NT, 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.

Follow these guidelines for compiling and linking.

Include Directory and nsapi.h File

Add the *server_root*`/plugins/include` (UNIX) or *server_root*`\plugins\include` (Windows NT) 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 NT) library directory to your linker command.

Table 4-1 lists the library that you need to link to.

Table 4-1 Libraries

Platform	Library
Windows NT	`ns-httpd40.dll` (in addition to the standard Windows libraries)
HPUX	`libns-httpd40.sl`
All other UNIX platforms	`libns-httpd40.so`

Linker Commands and Options for Generating a Shared Object

To generate a shared library, use the commands and options listed in Table 4-2.

Table 4-2 Linker commands and options

Platform	Options
Solaris Sparc	`ld -G` or `cc -G`
Windows NT	`link -LD`
HPUX	`cc +Z -b -Wl,+s -Wl,-B,symbolic`
AIX	`cc -p 0 -berok -blibpath:$(LD_RPATH)`
Compaq	`cc -shared`
Linux	`gcc -shared`
IRIX	`cc -shared`

Additional Linker Flags

Use the linker flags in Table 4-3 to specify which directories should be searched for shared objects during runtime to resolve symbols.

Table 4-3 Linker flags

Platform	Flags
Solaris Sparc	`-R` *dir*`:`*dir*
Windows NT	(no flags, but the `ns-httpd40.dll` file must be in the system PATH variable)

Table 4-3 Linker flags

Platform	Flags
HPUX	`-Wl,+b,`*dir*`,`*dir*
AIX	`-blibpath:`*dir*`:`*dir*
Compaq	`-rpath` *dir*`:`*dir*
Linux	`-Wl,-rpath,`*dir*`:`*dir*
IRIX	`-Wl,-rpath,`*dir*`:`*dir*

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

Table 4-4 lists the flags and defines that you need to use for compilation of your source code.

Table 4-4 Compiler flags and defines

Platform	Flags/Defines
Solaris Sparc	`-DXP_UNIX -D_REENTRANT -KPIC -DSOLARIS`
Windows NT	`-DXP_WIN32 -DWIN32 /MD`
HP-UX	`-DXP_UNIX -D_REENTRANT -DHPUX`
AIX	`-DXP_UNIX -D_REENTRANT -DAIX  $(DEBUG)`
Compaq	`-DXP_UNIX -KPIC`
Linux	`-DLINUX -D_REENTRANT -fPIC`
IRIX	`-o32 -exceptions -DXP_UNIX -KPIC`
All Platforms	`-MCC_HTTPD -NET_SSL`

Table 4-5 lists the optional flags and defines you can use.

Table 4-5 Optional flags and defines

Flag/Define	Platforms	Description
`-DSPAPI20`	All	Needed for the proxy utilities function include file `putil.h`

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.

iPlanet 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 Netscape Enterprise Server, however, were built on AIX 4.1, which did not support native runtime-linking. Enterprise Server had specific additional software (provided by IBM AIX development to Netscape) 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 iPlanet Web Server 4.*x* and 6.*x* versions as they are.

However, they can easily be relinked to work with iPlanet 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 iPlanet 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 iPlanet 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 (-) in place of 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 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:/Netscape/server4/docs/animations/small, while all files to be treated as full screen animations reside in the directory D:/Netscape/server4/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 pathnames and also assign a name to the request.

```
NameTrans fn=pfx2dir from="/animations/small"
dir="D:/Netscape/server4/docs/animations/small" name="small_anim"
NameTrans fn=pfx2dir from="/animations/fullscreen"
dir="D:/Netscape/server4/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>
```

Reconfigure the Server

After modifying `obj.conf`, you need to reconfigure the server. See "Dynamic Reconfiguration," on page 22 for details.

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 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 Netscape 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 the public routines are detailed in Chapter 5, "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 include:

- `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 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 7, "Syntax and Use of magnus.conf."

- `MALLOC`
- `FREE`
- `STRDUP`
- `REALLOC`
- `CALLOC`
- `PERM_MALLOC`
- `PERM_FREE`
- `PERM_STRDUP`
- `PERM_REALLOC`
- `PERM_CALLOC`

File I/O

The file I/O functions provides 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_write` writes to the network socket.

Threads

Thread functions include functions for creating your own threads which 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.
- `util_getline` gets the next line (up to a LF or CRLF) from a buffer.
- `util_hostname` gets the local hostname as a fully qualified domain name.
- `util_later_than` compares two dates.
- `util_sprintf` same as standard library routine `sprintf()`.
- `util_strftime` same as 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 plug-in 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` which 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
- Service SAFs
- Error SAFs
- AddLog SAFs

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`.
- 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` which 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).
- 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

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`.

CGI to NSAPI Conversion

You may have a need to convert a CGI variable into a 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 which are thread-safe. Also, you should use the NSAPI memory management and other routines for speed and platform independence.

Table 4-6

CGI getenv()	NSAPI
`AUTH_TYPE`	`pblock_findval("auth-type", rq->vars);`

Table 4-6

CGI getenv()	NSAPI
AUTH_USER	pblock_findval("auth-user", rq->vars);
CONTENT_LENGTH	pblock_findval("content-length", rq->headers);
CONTENT_TYPE	pblock_findval("content-type", rq->headers);
GATEWAY_INTERFACE	"CGI/1.1"
HTTP_*	pblock_findval("*", rq->headers); (* is lower-case, dash replaces underscore)
PATH_INFO	pblock_findval("path-info", rq->vars);
PATH_TRANSLATED	pblock_findval("path-translated", rq->vars);
QUERY_STRING	pblock_findval("query", rq->reqpb); (GET only, POST puts query string in body data)
REMOTE_ADDR	pblock_findval("ip", sn->client);
REMOTE_HOST	session_dns(sn) ? session_dns(sn) : pblock_findval("ip", sn->client);
REMOTE_IDENT	pblock_findval("from", rq->headers); (not usually available)
REMOTE_USER	pblock_findval("auth-user", rq->vars);
REQUEST_METHOD	pblock_findval("method", req->reqpb);
SCRIPT_NAME	pblock_findval("uri", rq->reqpb);
SERVER_NAME	char *util_hostname();
SERVER_PORT	conf_getglobals()->Vport; (as a string)
SERVER_PROTOCOL	pblock_findval("protocol", rq->reqpb);
SERVER_SOFTWARE	MAGNUS_VERSION_STRING
Netscape specific:	
CLIENT_CERT	pblock_findval("auth-cert", rq->vars)
HOST	char *session_maxdns(sn); (may be null)
HTTPS	security_active ? "ON" : "OFF";
HTTPS_KEYSIZE	pblock_findval("keysize", sn->client);
HTTPS_SECRETKEYSIZE	pblock_findval("secret-keysize", sn->client);

Table 4-6

CGI getenv()	NSAPI
QUERY	pblock_findval(query", rq->reqpb); (GET only, POST puts query string in entity-body data)
SERVER_URL	http_uri2url_dynamic("","", sn, rq);

Chapter 5

NSAPI Function Reference

This chapter lists all 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 7, “Syntax and Use of magnus.conf,” for a list of the pre-defined Init SAFs. See Chapter 3, “Predefined SAFs and the Request Handling Process,” for a list of the rest of the pre-defined SAFs.

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 Appendix A, “Data Structure Reference,” and also look in the `nsapi.h` header file in the `include` directory in the build for iPlanet Web Server 6.0.

NSAPI Functions (in Alphabetical Order)

For an alphabetical list of function names, see Appendix G, “Alphabetical List of NSAPI Functions and Macros.”

C D F L M N P R S U V

C

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

```
void *CALLOC(int num, int size)
```

Returns

A void pointer to a block of memory.

Parameters

int num is the number of elements to allocate.

int size is the size in bytes of each element.

Example

```
/* Allocate space for an array of 100 char pointers */
char *name;
name = (char *) CALLOC(100, sizeof(char *));
```

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

```
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

```
void condvar_terminate(CONDVAR cv);
```

Returns

void

Parameters

CONDVAR cv is a condition variable.

See also

condvar_init, condvar_notify, condvar_wait, crit_init, crit_enter, crit_exit, crit_terminate.

condvar_wait

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

```
void condvar_wait(CONDVAR cv);
```

Returns

`void`

Parameters

`CONDVAR cv` is a condition variable.

See also

`condvar_init`, `condvar_notify`, `condvar_terminate`, `crit_init`, `crit_enter`, `crit_exit`, `crit_terminate`.

crit_enter

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

```
void crit_enter(CRITICAL crvar);
```

Returns

`void`

Parameters

`CRITICAL crvar` is a critical-section variable.

See also

`crit_init`, `crit_exit`, `crit_terminate`.

crit_exit

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

```
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

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

```
CRITICAL crit_init(void);
```

Returns

A newly allocated critical-section variable (`CRITICAL`)

Parameters

none.

See also

`crit_enter`, `crit_exit`, `crit_terminate`.

crit_terminate

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`.

D

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 pathname of an exisiting file. The file is opened in Read Mode only. The application calling this function should not modify the currency of the file pointed 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);
```

Returns

Pointer to PRFileDesc, 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` 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);

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 which 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

```
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, filebuf_open_nostat`

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

```
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

```
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

```
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 which 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

```
filebuf *buf = filebuf_open_nostat(fd, FILE_BUFFERSIZE, &finfo);
if (!buf) {
      system_fclose(fd);
}
```

See also

`filebuf_close, filebuf_open, filebuf_getc, filebuf_buf2sd`

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

`MALLOC`, `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

```
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

```
/* 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`

L

log_error

The `log_error` function creates an entry in an error log, recording the date, the severity, and a specified text.

Syntax

```
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 non-recoverable 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.

The Session and Request parameters are the same as the ones passed into your SAF.

`char *fmt` specifies the format for the `printf` function that delivers the message.

`...` represents a sequence of parameters for the `printf` function.

Example

```
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

```
void *MALLOC(int size)
```

Returns

A void pointer to a block of memory.

Parameters

`int size` is the number of bytes to allocate.

Example

```
/* 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`

N

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 7, "Syntax and Use of magnus.conf."

Syntax

```
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 (Internet Protocol) address as a character string in dotted-decimal notation: `nnn.nnn.nnn.nnn`

`int verify`, if non-zero, 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_write

The `net_write` function writes a specified number of bytes to a specified socket from a specified buffer. It returns the number of bytes written.

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);
```

Returns

`void`

Parameters

`netbuf *buf` is the buffer to close.

See also

`netbuf_buf2sd`, `netbuf_getc`, `netbuf_grab`, `netbuf_open`

netbuf_getc

The `netbuf_getc` function retrieves a character from the cursor position of the network buffer specified by `b`.

Syntax

```
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_close`, `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

```
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_getc`, `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

P

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

```
pb_param *param_create(char *name, char *value);
```

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_find`, `pblock_remove`, `pblock_nvinsert`

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

```
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

```
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_nninsert`, `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

```
char **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 non-heap space for the string if needed.

Use this function to stream the `pblock` for archival and other purposes.

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 in order to deallocate the memory used by the pb_param structure.

Syntax

```
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

```
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 back slashes (\) 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 num*size bytes 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_CALLOC and CALLOC both obtain their memory from the system heap.

Syntax

```
void *PERM_CALLOC(int num, int size)
```

Returns

A void pointer to a block of memory

Parameters

int num is the number of elements to allocate.

int size is the size in bytes of each element.

Example

```
/* Allocate 256 bytes for a name */
char **name;
name = (char **) PERM_CALLOC(100, sizeof(char *));
```

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

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(vod *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);
```

See also

PERM_MALLOC, PERM_FREE, PERM_CALLOC, PERM_STRDUP, MALLOC, FREE, STRDUP, CALLOC, REALLOC

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

```
newstr = (char *) PERM_MALLOC(strlen(str) + 1);
strcpy(newstr, str);
```

A string created with PERM_STRDUP should be disposed with PERM_FREE.

Syntax

`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.

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

```
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.

The Session and Request parameters are the same as the ones passed into your SAF.

`stat *finfo` is the stat structure for the file.

The `stat` structure contains the information about the file from the file system. You can get the `stat` structure info using `request_stat_path`.

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

```
int protocol_start_response(Session *sn, Request *rq);
```

Returns

The constant `REQ_PROCEED` if the operation succeeded, in which case you should send the data you were preparing to send.

The constant `REQ_NOACTION` if the operation succeeded, but the request method was HEAD in which case no data should be sent to the client.

The constant `REQ_ABORTED` if the operation did not succeed.

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.

Example

```
/* 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;
}
```

See also

`protocol_status`

protocol_status

The `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 `protocol_start_response`.

For the complete list of valid status code constants, please refer to the file "`nsapi.h`" in the server distribution

Syntax

```
void protocol_status(Session *sn, Request *rq, int n, char *r);
```

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

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 that 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`

R

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;
name = (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`

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

```
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`

S

session_dns

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 7, "Syntax and Use of magnus.conf."

Syntax

```
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 DNS 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 7, "Syntax and Use of magnus.conf."

Syntax

```
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 pre-validated 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 like `*.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);
```

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 pre-validated shell expression (wildcard pattern) to compare against.

See also

`shexp_casecmp`, `shexp_cmp`, `shexp_valid`

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

```
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:

```
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

```
char *name1 = "MyName";
char *name2 = STRDUP(name1);
```

See also

MALLOC, FREE, CALLOC, REALLOC, PERM_MALLOC, PERM_FREE, PERM_CALOC, 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 using 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 that your program will append data to.

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.

See also

`system_errmsg`, `system_fopenRO`, `system_fopenRW`, `system_lseek`, `system_fread`, `system_fwrite`, `system_fwrite_atomic`, `system_flock`, `system_ulock`, `system_fclose`

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

```
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_fopenWA`, `system_lseek`, `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

```
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));
```

See also

`system_errmsg`, `system_fopenRO`, `system_fopenRW`, `system_fopenWA`, `system_lseek`, `system_fread`, `system_fwrite`, `system_flock`, `system_ulock`, `system_fclose`

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 a 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

```
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

```
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 pathname to a local file system pathname. 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 like Windows NT. You can use system_unix2local to convert the Unix file name into the format that Windows NT 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 pathname to be converted.

`char *lp` is the local pathname.

You must allocate the parameter `lp`, and it must contain enough space to hold the local pathname.

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.

See also

`systhread_current`, `systhread_getdata`, `systhread_init`, `systhread_newkey`, `systhread_setdata`, `systhread_sleep`, `systhread_start`, `systhread_timerset`

systhread_current

The `systhread_current` function returns a pointer to the current thread.

Syntax

```
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

```
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.

See also

systhread_current, systhread_newkey, systhread_setdata, systhread_sleep, systhread_start, systhread_timerset

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

```
int systhread_newkey(void);
```

Returns

An integer key.

Parameters

`none.`

See also

`systhread_current`, `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

```
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`.

See also

`systhread_current`, `systhread_getdata`, `systhread_newkey`, `systhread_sleep`, `systhread_start`, `systhread_timerset`

systhread_sleep

The `systhread_sleep` function puts the calling thread to sleep for a given time.

Syntax

```
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_setdata`, `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

```
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, 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`

U

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 to see 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

```
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 NT, 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 paths.

Syntax

```
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 NT, 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 paths.

Syntax

```
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

```
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 non alphanumeric 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

```
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 `l`.

`char *l` 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

```
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 like 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

`char *ua` is the user-agent string from the request headers.

`char *major` is the major release number (to the left of the decimal point).

`char *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);
```

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

```
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 run-time library of your compiler.

Syntax

```
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 run-time 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 alpha-numeric 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 run-time library of your compiler. Refer to Appendix D, "Time Formats" for details on time formats.

`const struct tm *t` is a pointer to a calendar time (`tm`) struct, 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 alpha-numeric 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

```
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.

See also

util_strcasecmp

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

```
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 NT./) at the end of the URI. Use this function to see if a URI requested by the client is insecure.

Syntax

```
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 plug-in doesn't work.

Syntax

```
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 run-time library of your compiler.

Syntax

```
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 run-time library of your compiler.

Syntax

```
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

```
int vs_alloc_slot(void);
```

Returns

A slot number on success, or -1 on failure.

See also

vs_get_data, vs_set_data

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);
```

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

```
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

```
void* vs_set_data(const VirtualServer* vs, int* slot, void* data);
```

Returns

Data on success, 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

```
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.

Chapter 6

Examples of Custom SAFs

This chapter discusses examples of custom Sever Application Functions (SAFs) for each directive in the request-response process. You may wish to use these examples as the basis for implementing your own custom SAFs. For more information about creating your own custom SAFs, see Chapter 4, “Creating Custom SAFs.”

Before writing custom SAFs, you should be familiar with the request-response process (discussed in Chapter 1, “Basics of Server Operation”) and the role of the configuration file `obj.conf` (discussed in Chapter 2, “Syntax and Use of obj.conf”).

Before writing your own SAF, check if an existing SAF serves your purpose. The pre-defined SAFs are discussed in Chapter 3, “Predefined SAFs and the Request Handling Process.”

For a list of the NSAPI functions for creating new SAFs, see Chapter 5, “NSAPI Function Reference.”

This chapter has the following sections:

- Examples in the Build
- AuthTrans Example
- NameTrans Example
- PathCheck Example
- ObjectType Example
- Service Example
- AddLog Example
- Quality of Service Examples

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 the example files.

To test an example, load the `examples` shared library into the iPlanet 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 iPlanet Web Server to load the changes, since `Init` directives are only applied during server initialization.

AuthTrans Example

This simple example of an `AuthTrans` function demonstrate how to use your own custom ways of verifying that the username and password that a remote client provided is accurate. This program uses a hard coded table of usernames 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 username 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 username and password from the headers associated with the request. When a client initially makes a request, the username and password are unknown so the `AuthTrans` function and `PathCheck` function work together to reject the request, since they can't validate the username and password. When the client receives the rejection, the usual response is for it to pop up a dialog box asking the user for their username and password, and then the client submits the request again, this time including the username and password in the headers.

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

Installing the Example

To install the function on the iPlanet 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 also:

```
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 */
            /* user again */
            return REQ_NOACTION;
        }

        /* If we return REQ_PROCEED, the username will be accepted */
        return REQ_PROCEED;
    }

    /* No match, have it ask them again */
    log_error(LOG_SECURITY, "hardcoded-auth", sn, rq,
        "unknown user %s", user);
    return REQ_NOACTION;
}
```

NameTrans Example

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 The main thing that a `NameTrans` function usually does is to convert the logical URL in `ppath` in `rq->vars` to a physical pathname. However, the example discussed here, `explicit_pathinfo`, does not translate the URL into a physical pathname, 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 pathname.

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 on the end.

Normally `NameTrans` directives 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 pathname.

Installing the Example

To install the function on the iPlanet 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);
        return REQ_PROCEED;
    }
    /* Else do nothing */
    return REQ_NOACTION;
}
```

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:

`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`).

`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:

`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.

```
#include "nsapi.h"

/*  Set to NULL to prevent problems with people not calling
    acf-init */
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 */

/* The longest line we'll allow in an access control file */
#define MAX_ACF_LINE 256

/* Used to free static array on restart */
#ifdef __cplusplus
extern "C"
#endif

NSAPI_PUBLIC void acf_free(void *unused)
{
    register int x;

    for(x = 0; hosts[x]; ++x)
        FREE(hosts[x]);
    FREE(hosts);
    hosts = NULL;
}

#ifdef __cplusplus
extern "C"
#endif

NSAPI_PUBLIC int acf_init(pblock *pb, Session *sn, Request *rq)
{
    /* Parameter */
    char *acf_file = pblock_findval("file", pb);

    /* Working variables */
    int num_hosts;
    FILE *f;
    char err[MAGNUS_ERROR_LEN];
    char buf[MAX_ACF_LINE];
```

```
    /* Check usage. Note that Init functions have special
       error logging */
    if(!acf_file) {
        util_sprintf(err, "missing parameter to acf_init
            (need file)");
        pblock_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());
        pblock_nvinsert("error", err, pb);
        return REQ_ABORTED;
    }

    /* Initialize hosts array */
    num_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 main thing an `ObjectType` directive needs to do 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:

```
/* Set the content-type to magnus-internal/parsed-html */
pblock_nvinsert("content-type", "magnus-internal/parsed-html",
    rq->srvhdrs);
```

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.

```
#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->vars);
    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;
}
```

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.""More Complex Service Example," on page 231.

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

```
#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
    * 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 which 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 filename 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)
{
    /* 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 Examples

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

For more information, see the *Performance Tuning, Sizing, and Scaling Guide for iPlanet Web Server*.

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 "frame/log.h"
#include "frame/http.h"
#include "safs/qos.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
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(pblock *pb, Session *sn, Request *rq)
{
    char error[1024] = "";

    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
%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 */
        pblock_nvreplace("content-type", "text/html", rq->srvhdrs);

        protocol_start_response(sn, rq);
        net_write(sn->csd, error, strlen(error));
        return REQ_PROCEED;
    }
    else
    {
    /* this 503 didn't come from a QOS SAF failure, let someone
    else handle it */
    return REQ_PROCEED;
    };
}

/*------------------------------------------------------------------------------
 qos_handler

 This is an NSAPI AuthTrans function

It examines the QOS values in the request and compare 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(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);

    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, PROTOCOL_SERVICE_UNAVAILABLE, NULL);
    return REQ_ABORTED;
    };
}
```

Chapter 7

Syntax and Use of magnus.conf

When the iPlanet 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. iPlanet Web Server executes all the directives defined in `magnus.conf`.

Except for the `Init` SAFs, the directives in `magnus.conf` specify a variable and a value, for example:

```
ServerID https-boots.mcom.com
#ServerRoot d:/netscape/server4/https-boots.mcom.com
```

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 global settings that can be specified in `magnus.conf` in iPlanet Web Server 6.0.

The categories are:

- Init SAFs
- Server Information
- Language Issues
- DNS Lookup
- Threads, Processes and Connections

- Native Thread Pools
- CGI
- Error Logging and Statistic Collection
- ACL
- Security
- Chunked Encoding
- Miscellaneous

For an alphabetical list of directives, see Appendix H, "Alphabetical List of Directives in magnus.conf."

NOTE Much of the functionality of the file cache is controlled by a configuration file called `nsfc.conf`. For information about `nsfc.conf`, see the *Performance Tuning, Sizing, and Scaling Guide for iPlanet Web Server.*

Init SAFs

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.

The following `Init`-class functions are described in detail in this section:

- `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-uhome` loads user home directory information.
- `load-modules` loads shared libraries into the server.
- `nt-console-init` enables the NT 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:

`opts` (optional) is a 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.

`widths` (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 non-zero) 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 non-zero value for a column which is less than the length of its title, the width defaults to the minimum required to display the title.

`timezone` (optional) This 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`.

`format` (optional) This 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`.

`ignore` (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 (.).

`icon-uri` (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.

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
```

See Also

`index-common`, `find-index`, `home-page`

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` see "The bucket Parameter," on page 49 and the `service-dump` function). This function works only if the `perf-init` function is enabled.

For more information about performance buckets, see the *Performance Tuning, Sizing, and Scaling Guide for iPlanet Web Server*.

Parameters

`name`	A name for the bucket, for example `cgi-bucket`.
`description`	A description of what the bucket measures, for example `CGI Stats`.

Example:

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

See Also

`perf-init`

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

`cache-size`	(optional) specifies how many entries are contained in the cache. Acceptable values are 32 to 32768; the default value is 1024.
`expire`	(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).

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 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.

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

logFileName	The 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-servern ame/logs/access"`
	`mylogfile = "log1"`
	You will use the log file name later, as a parameter to the `flex-log` function.

`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.

`buffer-size`	Specifies the size of the global log buffer. The default is `8192`. See the third `flex-init` example below.
`num-buffers`	Specifies the maximum number of logging buffers to use. The default is `1000`. See the third `flex-init` example below.

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 4, “Creating Custom SAFs” for more information about parameter blocks and Chapter 5, “NSAPI Function Reference,” for functions to manipulate pblocks.)

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 Table 7-1. 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 7-1 Typical components of flex-init formatting

Flex-log option	Component
Client Host name (unless `iponly` is specified in flex-log or DNS name is not available) or IP address	`%Ses->client.ip%`
Client DNS name	`%Ses->client.dns%`
System date	`%SYSDATE%`
Full HTTP request line	`\"%Req->reqpb.clf-request%\"`

Table 7-1 Typical components of flex-init formatting

Flex-log option	Component
Status	`%Req->srvhdrs.clf-status%`
Response content length	`%Req->srvhdrs.content-length%`
Response content type	`%Req->srvhdrs.content-type%`
Referer header	`\"%Req->headers.referer%\"`
User-agent header	`\"%Req->headers.user-agent%\"`
HTTP Method	`%Req->reqpb.method%`
HTTP URI	`%Req->reqpb.uri%`
HTTP query string	`%Req->reqpb.query%`
HTTP protocol version	`%Req->reqpb.protocol%`
Accept header	`%Req->headers.accept%`
Date header	`%Req->headers.date%`
If-Modified-Since header	`%Req->headers.if-modified-since%`
Authorization header	`%Req->headers.authorization%`
Any header value	`%Req->headers.`*headername*`%`
Name of authorized user	`%Req->vars.auth-user%`
Value of a cookie	`%Req->headers.cookie.`*name*`%`
Value of any variable in `Req->vars`	`%Req->vars.`*varname*`%`
Virtual Server ID	`%vsid%`

Examples

The first example below initializes flexible logging into the file `/usr/netscape/server4/https-servername/logs/access`.

```
Init fn=flex-init
access="/usr/netscape/server4/https-servername/logs/access"
format.access="%Ses->client.ip% - %Req->vars.auth-user%
[%SYSDATE%] \"%Req->reqpb.clf-request%\" %Req->srvhdrs.clf-status%
%Req->srvhdrs.content-length%"
```

This will record the following items

- ip or hostname, followed by the three characters " - "
- the user name, followed by the two characters " ["
- the system date, followed by the two characters "] "
- the full HTTP request in quotes, followed by a single space
- the HTTP result status in quotes, followed by a single space
- the 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 `/user/netscape/server4/https-servername/logs/extended`.

```
Init fn=flex-init
extended="/usr/netscape/server4/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 7, "Syntax and Use of magnus.conf") frees the request handling threads to quickly write the log data.

```
Init fn=flex-init buffer-size=16384 num-buffers=2000
access="/usr/netscape/server4/https-servername/logs/access"
format.access="%Ses->client.ip% - %Req->vars.auth-user%
[%SYSDATE%] \"%Req->reqpb.clf-request%\" %Req->srvhdrs.clf-status%
%Req->srvhdrs.content-length%"
```

See Also

`flex-rotate-init`, `flex-log`

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 up disk space.

By default, log rotation is disabled.

Parameters

`rotate-start`	Indicates the time to start rotation. This value is a 4 digit string indicating the time in 24 hour format, for example, 0900 indicates 9 am while 1800 indicates 9 pm.
`rotate-interval`	Indicates the number of minutes to elapse between each log rotation.
`rotate-access`	(optional) determines whether `common-log`, `flex-log`, and `record-useragent` logs are rotated. Values are `yes` (the default) and `no`.

`rotate-error`	(optional) determines whether error logs are rotated. Values are `yes` (the default) and `no`.
`rotate-callback`	(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.

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`, `common-log`, `flex-log`, `record-useragent`

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

`timeout`	(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.

`cgistub-path`	(optional) specifies the path to the CGI stub binary. If not specified, iPlanet 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 which 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 iPlanet 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 *iPlanet Web Server Programmer's Guide*.
env-variable	(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.

Example

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

See Also

`send-cgi`, `send-wincgi`, `send-shellcgi`

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

logFileName	The 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 `common-log` function.

Examples

```
Init fn=init-clf
access=/usr/netscape/server4/https-boots/logs/access
Init fn=init-clf templog=/tmp/mytemplog templog2=/tmp/mytemplog2
```

See Also

`common-log`, `record-useragent`, `flex-rotate-init`

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

pwfile	(optional) specifies the full file system path to a file other than /etc/passwd. If not provided, the default Unix path (/etc/passwd) is used.

Examples

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

See Also

unix-home, find-links

load-modules

Applicable in Init-class directives.

The load-modules function loads a shared library or Dynamic Link Library 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 Server Application Functions, you get the server to load them by using the load-modules function and specifying the shared library or dll to load.

Parameters

shlib	specifies either the full path to the shared library or dynamic link library or a file name relative to the server configuration directory.

`funcs`	is a comma separated list of the names of the functions in the shared library or dynamic link library 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.
`NativeThread`	(optional) specifies which threading model to use. `no` causes the routines in the library to use user-level threading. `yes` enables kernel-level threading. The default is `yes`.
pool	the name of a custom thread pool, as specified in `thread-pool-init`.

Examples

```
Init fn=load-modules shlib="C:/mysrvfns/corpfns.dll"
funcs="moveit"
Init fn=load-modules shlib="/mysrvfns/corpfns.so"
funcs="myinit,myservice"
Init fn=myinit
```

nt-console-init

Applicable in `Init`-class directives.

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

Parameters

`stderr`	Directs error messages to the NT console. The required and only value is `console`.
`stdout`	Directs output to the NT console. The required and only value is `console`.

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 *Performance Tuning, Sizing, and Scaling Guide for iPlanet Web Server*.

Parameters

`disable`	flag to disable the use of system performance measurement via performance buckets. Should have a value of true or false. Default value is true.

Example

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

See Also

`define-perf-bucket`

pool-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).

NOTE Any memory you allocate from Init-class functions will be allocated as persistent memory, even if you use MALLOC. The server cleans up only the memory that is allocated while processing a request, and because Init-class functions are run before processing any requests, their memory is allocated globally.

Parameters

free-size	(optional) maximum size in bytes of free block list. May not be greater than 1048576.
disable	(optional) flag to disable the use of pooled memory. Should have a value of true or false. Default value is false.

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

Parameters

`methods`	is a comma separated list of the names of the methods you are registering.

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.

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

Parameters

`update-interval`	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.
`virtual-servers`	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.
`profiling`	enables NSAPI performance profiling using buckets if set to `yes`. This can also be enabled through the `perf-init` Init SAF. The default is `no`, which results in slightly better server performance.

Example

```
Init fn="stats-init" update-interval="5" virtual-servers="2000"
profiling="yes"
```

See also

`stats-xml`

thread-pool-init

Applicable in `Init`-class directives.

This function creates a new pool of user threads. A pool must be declared before it's 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 plug-ins.

On Windows NT, the default native pool is always being used and iPlanet Web Server uses fibers (user-scheduled threads) for initial request processing. Using custom additional pools on Windows NT 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," on page 273 in Chapter 7, "Syntax and Use of magnus.conf."

Parameters

`name`	name of the thread pool.
`maxthreads`	maximum number of threads in the pool.
`minthreads`	minimum number of threads in the pool.

`queueSize`	size of the queue for the pool. If all the 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.
`stackSize`	stack size of each thread in the native (kernel) thread pool.

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`

Server Information

This sub-section lists the directives in `magnus.conf` that specify information about the server. They are:

- ExtraPath
- MtaHost
- NetSiteRoot
- ServerConfigurationFile
- ServerID
- ServerRoot
- TempDir
- TempDirSecurity

- User

ExtraPath

Appends the specified directory name to the `PATH` environment variable. This is used for configuring Java on Windows NT. There is no default value; you must specify a value.

Syntax

`ExtraPath` *path*

MtaHost

Specifies the name of the SMTP mail server used by the server's agents. This value must be specified before reports can be sent to a mailing address.

NetSiteRoot

Specifies the absolute pathname to the top-level directory under which server instances can be found. This directive is used by the Administration Server. There is no default value; you must specify a value.

Syntax

`NetSiteRoot` *path*

ServerConfigurationFile

Specifies the location of the virtual server configuration file.

Syntax

`ServerConfigurationFile` *path*

Default

`ServerConfigurationFile` *server_root*/*server_id*`/config/server.xml`

ServerID

Specifies the server ID, such as `https-boots.mcom.com`.

ServerRoot

Specifies the server root. This directive is set during installation and is commented out. Unlike other directives, the server expects this directive to start with `#`. Do not change this directive. If you do, the Server Manager may not function properly.

Syntax

`#ServerRoot` *path*

Example

`#ServerRoot d:/netscape/server4/https-boots.mcom.com`

TempDir

Specifies the directory on the local volume that the server uses for its temporary files. On Unix, this directory must be owned by, and writable by, the user the server runs as. See also the directives User and TempDirSecurity.

Syntax

`TempDir` *path*

Default

`/tmp` (Unix)

`TEMP` (environment variable for Windows NT)

TempDirSecurity

Determines whether the server checks if the TempDir directory is secure. On Unix, specifying `TempDirSecurity off` allows the server to use `/tmp` as a temporary directory.

CAUTION Specifying `TempDirSecurity off` or using `/tmp` as a temporary directory on Unix is highly discouraged. Using `/tmp` as a temporary directory opens a number of potential security risks.

Syntax

`TempDirSecurity [on|off]`

Default

`on`

User

Windows NT: The `User` directive specifies the user account the server runs with. By using a specific user account (other than LocalSystem), you can restrict or enable system features for the server. For example, you can use a user account that can mount files from another machine.

Unix: The `User` directive specifies the Unix user account for the server. If the server is started by the superuser or root user, the server binds to the Port you specify and then switches its user ID to the user account specified with the `User` directive. This directive is ignored if the server isn't started as `root`. The user account you specify should have *read* permission to the server's root and subdirectories. The user account should have write access to the `logs` directory and execute permissions to any CGI programs. The user account should not have write access to the configuration files. This ensures that in the unlikely event that someone compromises the server, they won't be able to change configuration files and gain broader access to your machine. Although you can use the `nobody` user, it isn't recommended.

Syntax

`User` *name*

`name` is the 8-character (or less) login name for the user account.

Default

If there is no `User` directive, the server runs with the user account it was started with.

Examples

```
User http
```

```
User server
```

```
User nobody
```

Language Issues

This section lists the directives in `magnus.conf` related to language issues. The directives are:

- AdminLanguage
- ClientLanguage
- DefaultCharSet
- DefaultLanguage

AdminLanguage

For an international version of the server, this directive specifies the language for the Server Manager. Values are `en` (English), `fr` (French), `de` (German) or `ja` (Japanese).

Default

The default is `en`.

ClientLanguage

For an international version of the server, this directive specifies the language for client messages (such as File Not Found). Values are `en` (English), `fr` (French), `de` (German) or `ja` (Japanese).

Default

The default is `en`.

DefaultCharSet

For an international version of the server, this directive specifies the default character set for the server. The default character set is used for both the client responses and administration.

Default

The default is `iso-8859-1`.

DefaultLanguage

For an international version of the server, this directive specifies the default language for the server. The default language is used for both the client responses and administration. Values are `en` (English), `fr` (French), `de` (German) or `ja` (Japanese).

Default

The default is `en`.

DNS Lookup

This section lists the directives in `magnus.conf` that affect DNS lookup. The directives are:

- AsyncDNS

- DNS

AsyncDNS

Specifies whether asynchronous DNS is allowed. The `DNS` directive must be set to on for this directive to take effect. The value is either on or `off`. If DNS is enabled, enabling asynchronous DNS improves server performance.

Default

The default is `off`.

DNS

The `DNS` directive specifies whether the server performs DNS lookups on clients that access the server. When a client connects to your server, the server knows the client's IP address but not its host name (for example, it knows the client as 198.95.251.30, rather than its host name `www.a.com`). The server will resolve the client's IP address into a host name for operations like access control, CGI, error reporting, and access logging.

If your server responds to many requests per day, you might want (or need) to stop host name resolution; doing so can reduce the load on the DNS or NIS server.

Syntax

```
DNS [on|off]
```

Default

DNS host name resolution is on as a default.

Example

```
DNS on
```

Threads, Processes and Connections

In iPlanet Web Server 6.0, acceptor threads on a listen socket accept connections and put them onto a connection queue. Session threads then pick up connections from the queue and service the requests. The session threads post more session threads if required at the end of the request. The policy for adding new threads is based on the connection queue state:

- Each time a new connection is returned, the number of connections waiting in the queue (the backlog of connections) is compared to the number of session threads already created. If it is greater than the number of threads, more threads are scheduled to be added the next time a request completes.
- The previous backlog is tracked, so that if it is seen to be increasing over time, and if the increase is greater than the `ThreadIncrement` value, and the number of session threads minus the backlog is less than the `ThreadIncrement` value, then another `ThreadIncrement` number of threads are scheduled to be added.
- The process of adding new session threads is strictly limited by the `RqThrottle` value.
- To avoid creating too many threads when the backlog increases suddenly (such as the startup of benchmark loads), the decision whether more threads are needed is made only once every 16 or 32 times a connection is made based on how many session threads already exist.

This subsection lists the directives in `magnus.conf` that affect the number and timeout of threads, processes, and connections. They are:

- ConnQueueSize
- HeaderBufferSize
- IOTimeout
- KeepAliveThreads
- KeepAliveTimeout
- KernelThreads
- ListenQ
- MaxKeepAliveConnections
- MaxProcs (Unix Only)
- PostThreadsEarly
- RcvBufSize
- RqThrottle
- RqThrottleMin
- SndBufSize
- StackSize
- StrictHttpHeaders

- TerminateTimeout
- ThreadIncrement
- UseNativePoll (Unix only)

Also see the section “Native Thread Pools,” on page 273 for directives for controlling the pool of native kernel threads.

ConnQueueSize

Specifies the number of outstanding (yet to be serviced) connections that the web server can have. It is recommended that this value always be greater than the operating system limit for the maximum number of open file descriptors per process.

Default

The default value is 5000.

HeaderBufferSize

The size (in bytes) of the buffer used by each of the request processing threads for reading the request data from the client. The maximum number of request processing threads is controlled by the RqThrottle setting.

Default

The default value is `8192` (8 KB).

IOTimeout

Specifies the number of seconds the server waits for data to arrive from the client. If data does not arrive before the timeout expires then the connection is closed. By setting it to less than the default 30 seconds, you can free up threads sooner. However, you may also disconnect users with slower connections.

Syntax

IOTimeout *seconds*

Default

30 seconds for servers that don't use hardware encryption devices and 300 seconds for those that do.

KeepAliveThreads

This directive determines the number of threads in the keep-alive subsystem. It is recommended that this number be a small multiple of the number of processors on the system. (for example, a 2 CPU system should have 2 or 4 keep alive threads). The maximum number of keep-alive connections allowed (`MaxKeepAliveConnections`) should also be taken into consideration when choosing a value for this setting.

Default

1

KeepAliveTimeout

This directive determines the maximum time that the server holds open an HTTP Keep-Alive connection or a persistent connection between the client and the server. The Keep-Alive feature for earlier versions of the server allows the client/server connection to stay open while the server processes the client request. The default connection is a persistent connection that remains open until the server closes it or the connection has been open for longer than the time allowed by `KeepAliveTimeout`.

The timeout countdown starts when the connection is handed over to the keep-alive subsystem. If there is no activity on the connection when the timeout expires, the connection is closed.

Default

The default value is `30` seconds. The maximum value is `300` seconds (5 minutes).

KernelThreads

iPlanet Web Server can support both kernel-level and user-level threads whenever the operating system supports kernel-level threads. Local threads are scheduled by NSPR within the process whereas kernel threads are scheduled by the host operating system. Usually, the standard debugger and compiler are intended for use with kernel-level threads. By setting `KernelThreads` to `1` (on), you ensure that the server uses only kernel-level threads, not user-level threads. By setting `KernelThreads` to `0` (off), you ensure that the server uses only user-level threads, which may improve performance.

Default

The default is 0 (off).

ListenQ

Specifies the maximum number of pending connections on a listen socket. Connections that time out on a listen socket whose backlog queue is full will fail.

Default

The default value is platform-specific: 4096 (AIX), 200 (NT), 128 (all others).

MaxKeepAliveConnections

Specifies the maximum number of Keep-Alive and persistent connections that the server can have open simultaneously. Values range from 0 to 32768.

Default

MaxProcs (Unix Only)

Specifies the maximum number of processes that the server can have running simultaneously. If you don't include `MaxProcs` in your `magnus.conf` file, the server defaults to running a single process.

One process per processor is recommended if you are running in multi-process mode. In iPlanet Web Server 6.0, there is always a primordial process in addition to the number of active processes specified by this setting.

There is additional discussion of this and other server configuration and performance tuning issues in the *Performance Tuning, Sizing, and Scaling Guide for iPlanet Web Server*.

Default

1

PostThreadsEarly

If this directive is set to `1` (on), the server checks the whether the minimum number of threads are available at a listen socket after accepting a connection but before sending the response to the request. Use this directive when the server will be handling requests that take a long time to handle, such as those that do long database connections.

Default

0 (off)

RcvBufSize

Specifies the size (in bytes) of the receive buffer used by sockets. Allowed values are determined by the operating system.

Default

The default value is determined by the operating system. Typical defaults are 4096 (4K), 8192 (8K).

RqThrottle

Specifies the maximum number of request processing threads that the server can handle simultaneously. Each request runs in its own thread.

There is additional discussion of this and other server configuration and performance tuning issues in the *Performance Tuning, Sizing, and Scaling Guide for iPlanet Web Server*.

Default

RqThrottleMin

Specifies the number of request processing threads that are created when the server is started. As the load on the server increases, more request processing threads are created (up to a maximum of `RqThrottle` threads).

Default

SndBufSize

Specifies the size (in bytes) of the send buffer used by sockets.

Default

The default value is determined by the operating system. Typical defaults are `4096` (4K), `8192` (8K).

StackSize

Determines the maximum stack size for each request handling thread.

Default

The most favorable machine-specific stack size.

StrictHttpHeaders

Controls strict HTTP header checking. If strict HTTP header checking is on, the server rejects connections that include inappropriately duplicated headers.

Syntax

StrictHttpHeaders `[on|off]`

Default

`on`

TerminateTimeout

Specifies the time that the server waits for all existing connections to terminate before it shuts down.

Default

30 seconds

ThreadIncrement

The number of additional or new request processing threads created to handle an increase in the load on the server, for example when the number of pending connections (in the request processing queue) exceeds the number of idle request processing threads.

When a server starts up, it creates `RqThrottleMin` number of request processing threads. As the load increases, it creates `ThreadIncrement` additional request processing threads until `RqThrottle` request processing threads have been created.

Default

The default value is 10.

UseNativePoll (Unix only)

Uses a platform-specific poll interface when set to 1(on). Uses the NSPR poll interface in the KeepAlive subsystem when set to 0 (off).

Default

1 (on)

Native Thread Pools

This section lists the directives for controlling the size of the native kernel thread pool. You can also control the native thread pool by setting the system variables `NSCP_POOL_STACKSIZE`, `NSCP_POOL_THREADMAX`, and `NSCP_POOL_WORKQUEUEMAX`. If you have set these values as environment variables and also in `magnus.conf`, the environment variable values will take precedence.

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 plug-ins.

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

The directives are:

- NativePoolStackSize
- NativePoolMaxThreads
- NativePoolMinThreads
- NativePoolQueueSize

NativePoolStackSize

Determines the stack size of each thread in the native (kernel) thread pool.

Default

0

NativePoolMaxThreads

Determines the maximum number of threads in the native (kernel) thread pool.

Default

NativePoolMinThreads

Determines the minimum number of threads in the native (kernel) thread pool.

Default

1

NativePoolQueueSize

Determines the number of threads that can wait in the queue for the thread pool. If all threads in the pool are busy, then the next request-handling thread that needs to use a thread in the native pool must wait in the queue. If the queue is full, the next request-handling thread that tries to get in the queue is rejected, with the result that it returns a busy response to the client. It is then free to handle another incoming request instead of being tied up waiting in the queue.

Default

0

CGI

This section lists the directives in `magnus.conf` that affect requests for CGI programs. The directives are:

- CGIExpirationTimeout
- CGIStubIdleTimeout
- CGIWaitPid (UNIX Only)
- MaxCGIStubs
- MinCGIStubs

CGIExpirationTimeout

This directive specifies the maximum time in seconds that CGI processes are allowed to run before being killed.

The value of `CGIExpirationTimeout` should not be set too low - 300 seconds (5 minutes) would be a good value for most interactive CGIs; but if you have CGIs that are expected to take longer without misbehaving, then you should set it to the maximum duration you expect a CGI program to run normally. A value of `0` disables CGI expiration, which means that there is no time limit for CGI processes.

Note that on Windows NT platforms `init-cgi` time-out does not work, so you must use `CGIExpirationTimeout`.

Default

`0`

CGIStubIdleTimeout

This directive causes the server to kill any CGIStub processes that have been idle for the number of seconds set by this directive. Once the number of processes is at `MinCGIStubs`, the server does not kill any more processes.

Default

`30`

CGIWaitPid (UNIX Only)

For UNIX platforms, when `CGIWaitPid` is set to on, the action for the SIGCHLD signal is the system default action for the signal. If a NSAPI plugin fork/execs a child process, it should call `waitpid` with its child process `pid` when `CGIWaitPid` is enabled to avoid leaving "defunct" processes when its child process terminates. When `CGIWaitPid` is enabled, the SHTML engine waits explicitly on its exec cmd child processes. Note that this directive has no effect on CGI.

Default

`on`

MaxCGIStubs

Controls the maximum number of CGIStub processes the server can spawn. This is the maximum concurrent CGIStub processes in execution, not the maximum number of pending requests. The default value should be adequate for most systems. Setting this too high may actually reduce throughput.

Default

`10`

MinCGIStubs

Controls the number of processes that are started by default. The first CGIStub process is not started until a CGI program has been accessed. Note that if you have an `init-cgi` directive in the `magnus.conf` file, the minimum number of CGIStub processes are spawned at startup. The value must be less than the `MaxCGIStubs` value.

Default

`2`

WincgiTimeout

WinCGI processes that take longer than this value are terminated when this timeout (in seconds) expires.

Default

`60`

Error Logging and Statistic Collection

This section lists the directives in `magnus.conf` that affect error logging and the collection of server statistics. They are:

- ErrorLog
- ErrorLogDateFormat
- LogFlushInterval
- LogVerbose
- LogVsId
- PidLog

ErrorLog

The `ErrorLog` directive specifies the directory where the server logs its errors. If errors are reported to a file, then the file and directory in which the log is kept must be writable by whatever user account the server runs as.

Unix: You can also use the `syslog` facility.

Syntax

`ErrorLog` *logfile*

The *logfile* can be either a full path or file name.

On Unix systems, it can be the keyword `SYSLOG` (it must be in all capital letters).

Default

There is no default error log.

Examples

Windows NT:

```
ErrorLog C:\Netscape\ns-home\Logs\Errors
```

Unix:

```
ErrorLog /var/ns-server/logs/errors
ErrorLog SYSLOG
```

ErrorLogDateFormat

The `ErrorLogDateFormat` directive specifies the date format that the server logs use.

Syntax

`ErrorLogDateFormat` *format*

The *format* can be any format valid for the C library function `strftime`. See Appendix D, "Time Formats."

Default

`%d/%b/%Y:%H:%M:%S`

LogFlushInterval

This directive determines the log flush interval, in seconds, of the log flush thread.

Default

30

LogVerbose

This directive determines whether verbose logging occurs or not. If the value is on, the server logs all server messages including those that are not logged by default.

Default

off

LogVsId

This directive determines whether or not virtual server IDs are displayed in the error log. You should enable `LogVsId` when multiple virtual servers share the same log file.

Default

off

PidLog

`PidLog` specifies a file in which to record the process ID (pid) of the base server process. Some of the server support programs assume that this log is in the server root, in `logs/pid`.

To shut down your server, kill the base server process listed in the pid log file by using a `-TERM` signal. To tell your server to reread its configuration files and reopen its log files, use `kill` with the `-HUP` signal.

If the `PidLog` file isn't writable by the user account that the server uses, the server does not log its process ID anywhere. The server won't start if it can't log the process ID.

Syntax

`PidLog` *file*

The *file* is the full path name and file name where the process ID is stored.

Default

There is no default.

Examples

```
PidLog /var/ns-server/logs/pid
```

```
PidLog /tmp/ns-server.pid
```

ACL

This section lists the directives in `magnus.conf` relevant to access control lists (ACLs). They are:

- ACLCacheLifetime
- ACLUserCacheSize
- ACLGroupCacheSize

ACLCacheLifetime

`ACLCacheLifetime` determines the number of seconds before cache entries expire. Each time an entry in the cache is referenced, its age is calculated and checked against `ACLCacheLifetime`. The entry is not used if its age is greater than or equal to the `ACLCacheLifetime`. If this value is set to 0, the cache is turned off.

If you use a large number for this value, you may need to restart the iPlanet Web Server when you make changes to the LDAP entries. For example, if this value is set to 120 seconds, the iPlanet Web Server might be out of sync with the LDAP server for as long as two minutes. If your LDAP is not likely to change often, use a large number.

Default

120

ACLUserCacheSize

`ACLUserCacheSize` determines the number of users in the User Cache.

Default

200

ACLGroupCacheSize

`ACLGroupCacheSize` determines how many group IDs can be cached for a single UID/cache entry.

Default

4

Security

This section lists the directives in `magnus.conf` that affect server access and security issues for iPlanet Web Server. They are:

- Security
- SSLCacheEntries
- SSLClientAuthDataLimit
- SSLClientAuthTimeout
- SSLSessionTimeout
- SSL3SessionTimeout

Security

The Security directive globally enables or disables SSL by making certificates available to the server instance. It must be on for virtual servers to use SSL. If enabled, the user is prompted for the administrator password (in order to access certificates, and so on).

NOTE	When you create a secure listen socket through the Server Manager, security is automatically turned on globally in `magnus.conf`. When you create a secure listen socket manually in `server.xml`, security must be turned on by editing `magnus.conf`.

For more information about enabling SSL for individual virtual servers, see Chapter 8, "Virtual Server Configuration Files."

Syntax

```
Security [on|off]
```

Default

`off`

Example

```
Security off
```

SSLCacheEntries

Specifies the number of SSL sessions that can be cached. There is no upper limit.

Syntax

`SSLCacheEntries` *number*

If the *number* is 0, the default value, which is 10000, is used.

SSLClientAuthDataLimit

Specifies the maximum amount of application data, in bytes, that is buffered during the client certificate handshake phase.

Default

The default value is `1048576` (1 MB).

SSLClientAuthTimeout

Specifies the number of seconds after which the client certificate handshake phase times out.

Default

`60`

SSLSessionTimeout

The `SSLSessionTimeout` directive controls SSL2 session caching.

Syntax

`SSLSessionTimeout` *seconds*

The *seconds* value is the number of seconds until a cached SSL2 session becomes invalid. If the `SSLSessionTimeout` directive is specified, the value of seconds is silently constrained to be between 5 and 100 seconds.

Default

The default value is 100.

SSL3SessionTimeout

The `SSL3SessionTimeout` directive controls SSL3 session caching.

Syntax

`SSL3SessionTimeout` *seconds*

The *seconds* value is the number of seconds until a cached SSL3 session becomes invalid. The default value is 86400 (24 hours). If the `SSL3SessionTimeout` directive is specified, the value of seconds is silently constrained to be between 5 and 86400 seconds.

Chunked Encoding

This section lists directives that control chunked encoding. For more information, see “Buffered Streams,” on page 324.

- UseOutputStreamSize
- ChunkedRequestBufferSize
- ChunkedRequestTimeout

These directives have equivalent Service SAF parameters in `obj.conf`. The `obj.conf` parameters override these directives. For more information, see "Service Stage," on page 82.

UseOutputStreamSize

The `UseOutputStreamSize` directive determines the default output stream buffer size for the `net_read` and `netbuf_grab` NSAPI functions.

NOTE The `UseOutputStreamSize` parameter can be set to zero in the `obj.conf` file to disable output stream buffering. For the `magnus.conf` file, setting `UseOutputStreamSize` to zero has no effect.

Syntax

`UseOutputStreamSize` *size*

The *size* value is the number of bytes.

Default

The default value is `8192` (8 KB).

ChunkedRequestBufferSize

The `ChunkedRequestBufferSize` directive determines the default buffer size for "un-chunking" request data.

Syntax

`ChunkedRequestBufferSize` *size*

The *size* value is the number of bytes.

Default

The default value is 8192.

ChunkedRequestTimeout

The `ChunkedRequestTimeout` directive determines the default timeout for "un-chunking" request data.

Syntax

`ChunkedRequestTimeout` *seconds*

The *seconds* value is the number of seconds.

Default
The default value is 60 (1 minute).

Miscellaneous

This section lists miscellaneous other directives in `magnus.conf`.

- ChildRestartCallback
- HTTPVersion
- MaxRqHeaders
- Umask (UNIX only)

NOTE	Directives noted with boolean values have the following equivalent values: `on/yes/true` and `off/no/false`.

ChildRestartCallback

This directive forces the callback of NSAPI functions that were registered using the `daemon_atrestart` function when the server is restarting or shutting down. Values are `on`, `off`, `yes`, `no`, `true`, or `false`.

Default
`no`

HTTPVersion

The current HTTP version used by the server in the form *m.n*, where *m* is the major version number and *n* the minor version number.

Default
The default value is `1.1`.

MaxRqHeaders

Specifies the maximum number of header lines in a request. Values range from 0 to 32.

Default

32

Umask (UNIX only)

This directive specifies the umask value used by the NSAPI functions `System_fopenWA()` and `System_fopenRW()` to open files in different modes. Valid values for this directive are standard UNIX umask values.

For more information on these functions, see `system_fopenWA` and `system_fopenRW` in Chapter 5, "NSAPI Function Reference."

Chapter 8

Virtual Server Configuration Files

The `server.xml` file configures virtual servers. A master file, `server.dtd`, determines the format and content of the `server.xml` file. This chapter describes both these files and contains the following sections:

- The server.dtd File
- The server.xml File
- Elements in server.dtd and server.xml
- Virtual Server Selection for Request Processing
- User Database Selection
- The iPlanet LDAP Schema

The server.dtd File

The `server.dtd` file defines the various elements that the `server.xml` file can contain and the attributes these elements can have. The `server.dtd` file is located in the *server_root*/*server_id*/`config` directory.

NOTE	Do not edit the `server.dtd` file. Its contents change only with new versions of iPlanet Web Server.

For example, the following code defines the `VSCLASS` (or virtual server class) element. The first line specifies that a `VS` element can contain `VARS`, `VS`, or `QOSPARAMS` elements (if this element could not contain other elements, you would see `EMPTY` instead of a list of element names in parentheses). The remaining lines specify that a `VSCLASS` element can contain `id`, `connections`, `objectfile`, or `rootobject` attributes, but only the `id` attribute is required.

```
<!ELEMENT VSCLASS (VARS?,VS*,QOSPARAMS?)>
<!ATTLIST VSCLASS
   id ID #REQUIRED
   objectfile CDATA #IMPLIED
   rootobject CDATA #IMPLIED
   acceptlanguage (yes|no|on|off|1|0) #IMPLIED
```

Labels such as `ID` and `CDATA` are XML data types. For more information about XML, see the XML specification at:

`http://www.w3.org/TR/REC-xml`

The server.xml File

The `server.xml` file configures the addresses and ports that the server listens on and assigns virtual servers to these listen sockets. The encoding is UTF-8 to maintain compatibility with regular UNIX text editors. The `server.xml` file is located in the *server_root*/`https-`*server_id*/`config` directory.

Here is a simple `server.xml` file. It contains two listen sockets (`LS`), two virtual server classes (`VSCLASS`), and three virtual servers (`VS`).

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- declare any variables to be used in the server.xml file in the
ATTLIST below -->
<!DOCTYPE SERVER SYSTEM "server.dtd" [
<!ATTLIST VARS
   docroot CDATA #IMPLIED
   adminusers CDATA #IMPLIED
   webapps_file CDATA #IMPLIED
   webapps_enable CDATA #IMPLIED
   accesslog CDATA #IMPLIED
   user CDATA #IMPLIED
   group CDATA #IMPLIED
   chroot CDATA #IMPLIED
   dir CDATA #IMPLIED
```

```
    nice CDATA #IMPLIED
 >
]>

<SERVER legacyls="ls1">
    <VARS accesslog="/iws60/https-server.iplanet.com/logs/access"/>
    <LS id="ls1" ip="1.1.1.1" port="80" security="off"
    acceptorthreads="1">
        <CONNECTIONGROUP id="group1" matchingip="default"
        servername="server.iplanet.com"
        defaultvs="server.iplanet.com"/>
    </LS>
    <LS id="ls2" ip="any" port="80" security="off"
    acceptorthreads="1">
        <CONNECTIONGROUP id="group2" matchingip="default"
        servername="server2.iplanet.com"
        defaultvs="server2.iplanet.com"/>
    </LS>
    <MIME id="mime1" file="mime.types" />
    <ACLFILE id="acl1"
    file="/iws60/httpacl/generated.https-server.iplanet.com.acl" />
    <VSCLASS id="defaultclass" objectfile="obj.conf"
    rootobject="default" >
        <VARS docroot="/iws60/docs" />
        <VS id="server.iplanet.com" connections="group1" mime="mime1"
        aclids="acl1">
            <VARS webapps_file="web-apps.xml" webapps_enable="on" />
            <USERDB id="default" database="default" />
        </VS>
    </VSCLASS>
    <VSCLASS id="class2" objectfile="class2.obj.conf"
    rootobject="default" >
        <VARS docroot="/iws60/docs/class2" />
        <VS id="server2.iplanet.com" connections="group2"
        mime="mime1" aclids="acl1">
            <VARS webapps_file="web-apps.xml" webapps_enable="on" />
            <USERDB id="default" database="default" />
        </VS>
        <VS id="acme.com" connections="group2"
        mime="mime1" aclids="acl1">
            <VARS docroot="/iws60/docs/class2/acme"
            webapps_file="web-apps.xml" webapps_enable="on" />
            <USERDB id="default" database="default" />
        </VS>
    </VSCLASS>
</SERVER>
```

If no virtual server (VS) can be found that matches an IP address or `Host` header, requests are processed using the default VS defined in the `CONNECTIONGROUP`. This VS could be made to output a customized error message, or even handle the request using a special document root.

Variables

Defining variables for use in the `obj.conf` file is not required, but it is sometimes useful. The following code defines and uses a `docroot` variable:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- declare any variables to be used in the server.xml file in the
ATTLIST below -->
<!DOCTYPE SERVER SYSTEM "server.dtd" [
<!ATTLIST VARS
   docroot CDATA #IMPLIED
   ...
 >
]>
...
      <VS id="acme.com" connections="group2"
      mime="mime1" aclids="acl1">
         <VARS docroot="/iws60/docs/class2/acme"
         webapps_file="web-apps.xml" webapps_enable="on" />
         <USERDB id="default" database="default" />
      </VS>
...
```

This variable allows different document root directories to be assigned for different virtual servers. The variable can then be used in the `obj.conf` file. For example:

```
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 iPlanet Web Server configuration files. Any variable referenced in an `obj.conf` file must be defined in the `server.xml` file at the `SERVER`, `VSCLASS`, or `VS` level. Defining variables with default values at the `SERVER` or `VSCLASS` level and overriding them in the `VS` is recommended.

Format of a Variable

A variable is found when the following regular expression matches:

\$[A-Za-z][A-Za-z0-9_]*

This expression represents a $ followed by one or more alphanumeric characters. A delimited version ("${VARS}") is not supported. To get a regular $ character, use $$ in files to have variable substitution.

The id Variable

A special variable, `id`, is always available within a `VS` element and refers to the value of the `id` attribute. It is predefined and cannot be overridden. The `id` attribute uniquely identifies a virtual server. For example:

```
<VARS
   docroot="/export/$id"
/>
```

If the `id` attribute of the containing `VS` is `myserver`, the `docroot` variable is set to the value `/export/myserver`.

Variables Used in the Interface

The following variables are used by the Administration Server, Server Manager, Class Manager, and Virtual Server Manager. Unlike the `$id` variable, they are not predefined in the server, and they can be overridden.

`$docroot`	The document root of the virtual server. Typically evaluated as the value of the `document-root` parameter in the `obj.conf` file.
`$webapps_file`	The path and name of the web application configuration file, which is usually `web-apps.xml`. For more information about `web-apps.xml`, see the *Programmer's Guide to Servlets for iPlanet Web Server*.
`$webapps_enable`	A flag that indicates whether web applications are enabled for a `VS`. Allowed values are `on` and `off`. If the `webapps_file` variable has a value for a `VS`, this variable need not be defined and is assumed to be `on`.
`$accesslog`	The log file for a virtual server.
`$user`	The value of the `user` parameter of the `send-cgi` SAF.
`$group`	The value of the `group` parameter of the `send-cgi` SAF.
`$chroot`	The value of the `chroot` parameter of the `send-cgi` SAF.

`$dir`	The value of the `dir` parameter of the `send-cgi` SAF.
`$nice`	The value of the `nice` parameter of the `send-cgi` SAF.

Variable Evaluation

Variables are evaluated when generating specific objset for individual virtual servers.

Evaluation is recursive: variable values can contain other variables. For example:

```
...
<SERVER>
   <VARS docrootbase = "/export" />
   ...
   <VSCLASS ...>
      <VARS docroot = "$docrootbase/nonjava/$id" />
      ...
   </VSCLASS>
   <VSCLASS ...>
      <VARS docroot = "$docrootbase/java/$id" />
      ...
   </VSCLASS>
   ...
</SERVER>
...
```

Variables lower in the tree override variables from above. For example, it is possible to set a variable for a class of virtual servers and override it with a definition of the same variable in an individual virtual server.

Using the Server Manager and Class Manager

You can add virtual server classes and virtual servers to iPlanet Web Server through the Server Manager and Class Manager interface, as described in the *iPlanet Web Server Administrator's Guide*.

Elements in server.dtd and server.xml

This section describes the XML elements in the `server.dtd` and `server.xml` files. Subelements must be defined in the order in which they are listed.

SERVER

Defines a server. There can only be one of this element in a `server.xml` file.

Subelements: VARS, LS, MIME, ACLFILE, VSCLASS, QOSPARAMS

Attributes:

`qosactive`	Enables quality of service features, which let you set limits on server entities or view server statistics for bandwidth and connections. Allowed values are `yes`, `no`, `on`, `off`, `1`, `0`. The default is `no`.
`qosmetricsinterval`	(optional) The interval in seconds during which the traffic is measured. The default is `30`.
`qosrecomputeinterval`	(optional) The period in milliseconds in which the bandwidth gets recomputed for all server entities. The default is `100` ms.
`legacyls`	The `id` attribute of the listen socket for legacy (4.*x*) applications. This `LS` should contain only one `CONNECTIONGROUP`, which should be configured to only one `VS`, its `defaultvs`. All legacy applications must run on this virtual server, which is the default virtual server for the entire server.

VARS

Defines variables that can be given values in `server.xml` and referenced in `obj.conf`. “Variables,” on page 288. For a list of variables commonly defined in server.xml, see “Variables Used in the Interface,” on page 289.“Variables Used in the Interface” on page 289

Subelements: none

Attributes: none

LS (Listen Socket)

Defines a listen socket.

NOTE	When you create a secure listen socket through the Server Manager, security is automatically turned on globally in `magnus.conf.` When you create a secure listen socket manually in `server.xml`, security must be turned on by editing `magnus.conf`.

Subelements: `CONNECTIONGROUP`

Attributes:

`id`	(optional) The socket family type. A socket family type cannot begin with a number. When you create a secure listen socket in the `server.xml` file, `Security` must be turned on in `magnus.conf`. When you create a secure listen socket in the Server Manager, security is automatically turned on globally in `magnus.conf`. A listen socket name cannot begin with a number.
`ip`	IP address of the listen socket. Can be in dotted-pair or IPv6 notation. Can also be `any` for `INADDR_ANY`. Configuring a listen socket to listen on `any` is required if more than one `CONNECTIONGROUP` is configured to it.
`port`	Port number to create the listen socket on. Legal values are `1 - 65535`. On Unix, creating sockets that listen on ports `1 - 1024` requires superuser privileges. Configuring an SSL listen socket to listen on port 443 is recommended. Two different IP addresses can't use the same port.
`security`	(optional) Determines whether the listen socket runs SSL. Legal values are `on`, `off`, `yes`, `no`, `1`, `0`. The default is `no`. You can turn SSL2 or SSL3 on or off and set ciphers using an `SSLPARAMS` object in a `CONNECTIONGROUP` object. The `Security` setting in the `magnus.conf` file globally enables or disables SSL by making certificates available to the server instance. Therefore, `Security` in `magnus.conf` must be `on` or `security` in `server.xml` does not work. For more information, see Chapter 7, "Syntax and Use of magnus.conf."

`acceptorthreads`	(optional) Number of acceptor threads for the listen socket. The recommended value is the number of processors in the machine. The default is `1`, legal values are `1 - 1024`.
`family`	(optional) The socket family type. The default is `inet`, legal values are `inet, inet6, and nca`. Use the value `inet6` for IPv6 listen sockets. When using the value of `inet6,` IPv4 addresses will be prefixed with `::ffff:` in the log file. Specify nca to make use of the Solaris Network Cache and Accelerator.
`blocking`	(optional) Determines whether the listen socket and the accepted socket are put in to blocking mode. Use of blocking mode may improve benchmark scores. Legal values are `on`, `off`, `yes`, `no`, `1`, `0`. The default is `no`.

CAUTION Blocking mode sockets should not be used in real world deployments. Use of blocking mode sockets precludes dynamic reconfiguration and exposes the server to denial of service attacks.

CONNECTIONGROUP

Defines a group of connection properties to which you can assign virtual servers. See "Virtual Server Selection for Request Processing," on page 299 for more information.

Subelements: `SSLPARAMS`

Attributes:

`id`	Internal name for the connection group. A CONNECTIONGROUP name cannot begin with a number.
`matchingip`	IP address that the associated virtual servers use or the value `default`. Can be in dotted-pair or IPv6 notation. Cannot be `any` for `INADDR_ANY`. Must be `default` if the containing `LS` does not have `ip=any`. If the containing `LS` has `ip=any`, can be a specific IP address or `default`. In this case, `default` means any IP addresses not specified in other `LS` or `CONNECTIONGROUP` elements.
`defaultvs`	The `id` attribute of the default virtual server for this particular connection group.

`servername`	Tells the server what to put in the host name section of any URLs it sends to the client. This affects URLs the server automatically generates; it doesn't affect the URLs for directories and files stored in the server. This name should be the alias name if your server uses an alias. If you append a colon and port number, that port will be used in URLs the server sends to the client.

SSLPARAMS

Defines SSL parameters of a connection group.

An `SSLPARAMS` element is required inside, and only allowed inside, a `CONNECTIONGROUP` element contained by a listen socket that has its `security` attribute set to `on`.

Subelements: none

Attributes:

`servercertnickname`	The nickname of the server certificate in the certificate database or the PKCS#11 token. In the certificate, the name format is *tokenname*`:`*nickname*. Including the *tokenname*`:` part of the name in this attribute is optional.
`ssl2`	(optional) Determines whether SSL2 is enabled. Legal values are `on`, `off`, `yes`, `no`, `1`, `0`. The default is `no`. If both SSL2 and SSL3 are enabled for a virtual server, the server tries SSL3 encryption first. If that fails, the server tries SSL2 encryption.
`ssl2ciphers`	(optional) A space-separated list of the SSL2 ciphers used, with the prefix `+` to enable or `-` to disable, for example `+rc4`. Allowed values are `rc4export`, `rc2export`, `idea`, `des`.
`ssl3`	(optional) Determines whether SSL3 is enabled. Legal values are `on`, `off`, `yes`, `no`, `1`, `0`. The default is `yes`. If both SSL2 and SSL3 are enabled for a virtual server, the server tries SSL3 encryption first. If that fails, the server tries SSL2 encryption.

`ssl3tlsciphers`	(optional) A space-separated list of the SSL3 ciphers used, with the prefix `+` to enable or `-` to disable, for example `+rsa_des_sha`. Allowed SSL3 values are `rsa_des_sha`, `rsa_rc4_40_md5`, `rsa_rc2_40_md5`, `rsa_null_md5`. Allowed TLS values are `rsa_des_56_sha`, `rsa_rc4_56_sha`.
`tls`	(optional) Determines whether TLS is enabled. Legal values are `on`, `off`, `yes`, `no`, `1`, `0`. The default is `on`.
`tlsrollback`	(optional) Determines whether TLS rollback is enabled. Legal values are `on`, `off`, `yes`, `no`, `1`, `0`. The default is `on`. TLS rollback should be enabled for Microsoft Internet Explorer 5.0 and 5.5. For more information, see the *iPlanet Web Server Administrator's Guide*.
`clientauth`	(optional) Determines whether SSL3 client authentication is performed on every request, independent of ACL-based access control. Legal values are `on`, `off`, `yes`, `no`, `1`, `0`. The default is `no`.

MIME

Defines MIME types.

The most common way that the server determines the MIME type of a requested resource is by invoking the `type-by-extension` directive in the `ObjectType` section of the `obj.conf` file. The `type-by-extension` function does not work if no `MIME` element has been defined in the `SERVER` element.

Subelements: none

Attributes:

`id`	Internal name for the MIME types listing. Used in a `VS` element to define the MIME types used by the virtual server. The MIME types name cannot begin with a number.
`file`	The name of a MIME types file. For information about the format of this file, see Appendix B, "MIME Types."

ACLFILE

References one or more ACL files.

Subelements: none

Attributes:

`id`	Internal name for the ACL file listing. Used in a `VS` element to define the ACL file used by the virtual server. An ACL file listing name cannot begin with a number.
`file`	A space-separated list of ACL files. Each ACL file must have a unique name. For information about the format of an ACL file, see the *iPlanet Web Server Administrator's Guide*. The name of the default ACL file is `generated.https-`*server_id*`.acl`, and the file resides in the *server_root*`/`*server_id*`/httpacl` directory. To use this file, you must reference it in `server.xml`.

VSCLASS

Defines a virtual server class.

Subelements: `VARS`, `VS`, `QOSPARAMS`

Attributes:

`id`	Virtual server class ID. This is a unique ID that allows lookup of a specific virtual server class. A virtual server class ID cannot begin with a number.
`objectfile`	The file name of the `obj.conf` file for this class of virtual servers. Cannot be overridden in a `VS` element.
`rootobject`	(optional) Tells the server which object loaded from an `obj.conf` file is the default. The default object is expected to have all the name translation (`NameTrans`) directives for the virtual server; any server behavior that is configured in the default object affects the entire server. The default value is `default`. If you specify an object that doesn't exist, the server doesn't report an error until a client tries to retrieve a document. The Server Manager assumes the default to be the object named `default`. Don't deviate from this convention if you use (or plan to use) the Server Manager.

acceptlanguage	(optional) If on, the server parses the Accept-Language header and sends an appropriate language version based on which language the client can accept. You should set this value to on only if the server supports multiple languages. The default is off. Can be overridden in a VS element.

VS (Virtual Server)

Defines a virtual server.

Subelements: VARS, QOSPARAMS, USERDB

Attributes:

id	Virtual server ID. This is a unique ID that allows lookup of a specific virtual server. Can also be referred to as the variable $id in an obj.conf file. A virtual server ID cannot begin with a number.
connections	(optional) A space-separated list of CONNECTIONGROUP ids that specify the connection(s) the virtual server uses. Required only for a VS that is not the defaultvs of a CONNECTIONGROUP.
urlhosts	A space-separated list of values allowed in the Host request header to select the current virtual server. Each VS that is configured to the same CONNECTIONGROUP must have a unique urlhosts value for that group.
mime	The id of the MIME element used by the virtual server.
state	(optional) Determines whether a VS is active (on) or inactive (off, disable). The default is on (active). When inactive, a VS does not service requests. If a VS is disable, only the global server administrator can turn it on.
aclids	(optional) One or more id attributes of ACLFILE elements, separated by spaces. Specifies the ACL file(s) used by the virtual server.
errorlog	(optional) Specifies a log file for virtual-server-specific error messages.

`acceptlanguage`	(optional) If `on`, the server parses the `Accept-Language` header and sends an appropriate language version based on which language the client can accept. You should set this value to `on` only if the server supports multiple languages. The default is `off`.

QOSPARAMS

Defines quality of service parameters of a SERVER, VSCLASS, or VS.

Attributes of the SERVER element activate the quality of service features. In addition, the `qos-handler` and `qos-error` SAFs must be included in the `obj.conf` file.

For more information, see the *Performance Tuning, Sizing, and Scaling Guide for iPlanet Web Server*.

Subelements: none

Attributes:

`maxbps`	(optional) The maximum bandwidth limit for the SERVER, VSCLASS, or VS in bytes per second.
`enforcebandwidth`	(optional) Specifies whether the bandwidth limit should be enforced or not. Allowed values are `yes`, `no`, `on`, `off`, `1`, `0`. The default is `no`.
`maxconn`	(optional) The maximum number of concurrent connections for the SERVER, VSCLASS, or VS.
`enforceconnections`	(optional) Specifies whether the connection limit should be enforced or not. Allowed values are `yes`, `no`, `on`, `off`, `1`, `0`. The default is `no`.

USERDB

Defines the user database used by the virtual server.

See “User Database Selection,” on page 300 for more information about how a user database is selected for a given virtual server.

Subelements: none

Attributes:

`id`	The user database name in the virtual server's ACL file. A user database name cannot begin with a number.
`database`	The user database name in the `dbswitch.conf` file.
`basedn`	(optional) Overrides the base DN lookup in the `dbswitch.conf` file. However, the `basedn` value is still relative to the base DN value from the `dbswitch.conf` entry.
`certmaps`	(optional) Specifies which certificate to LDAP entry mappings (defined in `certmap.conf`) to use. If not present, all mappings are used. All lookups based on mappings in `certmap.conf` are relative to the final base DN of the VS.

Virtual Server Selection for Request Processing

Before the server can process a request, it must accept the request via a listen socket, then direct the request to the correct connection group and virtual server. This section discusses how the virtual server is determined.

After the virtual server is determined, the server executes the `obj.conf` file for the virtual server class to which the virtual server belongs. For details about how the server decides which directives to execute in `obj.conf`, see "Flow of Control in obj.conf," on page 34.

A connection group is first selected as follows:

- If the listen socket is configured to listen on a particular IP address, it can contain only one connection group, and that group is selected.
- If the listen socket is configured to listen on `any`, the IP address to which the client connected is matched to the `matchingip` attribute of a connection group contained by that listen socket. If no `matchingip` attribute matches, the connection group with `matchingip=default` is selected.

A virtual server is then selected as follows:

- If the connection group is configured to only a default virtual server, that virtual server is selected.

- If the connection group has more than one virtual server configured to it, the request `Host` header is matched to the `urlhosts` attribute of a virtual server. If no `Host` header is present or no `urlhosts` attribute matches, the default virtual server for the connection group is selected.

If a virtual server is configured to an SSL listen socket, its `urlhosts` attribute is checked against the subject pattern of the certificate at server startup, and a warning is generated and written to the error log if they don't match.

User Database Selection

A USERDB object selects a user database for the containing virtual server. How this selection occurs depends on the virtual server's ACL file and the `dbswitch.conf` file.

The ACL file format is unchanged from previous iPlanet Web Server versions. However, the following changes have been made in iPlanet Web Server 6.0:

- Virtual servers in `server.xml` reference ACL files. The `magnus.conf` file no longer references ACL files.
- The ACL file's `database` attribute does not map to a `dbswitch.conf` entry directly, but instead maps to an `id` attribute of a USERDB element. The `database` attribute of the USERDB element then maps to a `dbswitch.conf` entry. This extra layer between the ACL file and the `dbswitch.conf` file gives the server administrator full control over which databases virtual server administrators and users have access to.

iPlanet Web Server 6.0 introduces the following changes to the `dbswitch.conf` file and LDAP databases:

- The base DN in the LDAP URL in `dbswitch.conf` defines a root object for all further DN specifications. So, for most new installations, it can be empty, because the final base DN is determined in other ways -- either through a DC tree lookup or an explicit "`basedn`" value in the `USERDB` tag.
- A new `dbswitch.conf` attribute for LDAP databases, `dcsuffix`, defines the root of the DC tree. This root is relative to the base DN in the LDAP URL. You can use `dcsuffix` if the database is *schema compliant*. Requirements for schema compliance are listed in "The iPlanet LDAP Schema," on page 301.

A user database is selected for a virtual server as follows:

- If a `VS` has no `USERDB` subelement, user- or group-based ACLs fail.

- When no `database` attribute is present in a virtual server's ACL definition, the `VS` must have a `USERDB` subelement with an `id` attribute of `default`. The `database` attribute of the `USERDB` then points to a database in `dbswitch.conf`. If no "`database`" attribute is present, "`default`" is used.
- If an LDAP database is schema compliant, the base DN of the access is computed using a DC tree lookup of the `servername` attribute of the CONNECTIONGROUP. The DC tree lookup is based at the `dcsuffix` DN. The result must contain an `inetDomainBaseDN` attribute that contains the base DN. This base DN is taken as is and is not relative to any of the base DN values.
- If the `basedn` attribute of the USERDB element is not present and the database is not schema compliant, the accesses happen relative to the base DN in the `dbswitch.conf` entry, as in previous iPlanet Web Server versions.

The iPlanet LDAP Schema

You can use the `dcsuffix` attribute in the `dbswitch.conf` file if your LDAP database meets the requirements outlined in this section.

The subtree rooted at an ISP entry (for example, `o=isp`) is called the *convergence tree*. It contains all the directory data related to organizations (customers) served by an ISP.

The subtree rooted at `o=internet` is called the *domain component tree* or *dc tree*. It contains a sparse DNS tree with entries for the customer domains served. These entries are links to the appropriate location in the convergence tree where the data for that domain is located.

The directory tree may be single rooted, which is recommended (for example, `o=root` may have `o=isp` and `o=internet` under it), or have two separate roots, one for the convergence tree and one for the dc tree.

The Convergence Tree

The top level of the convergence tree must have one organization entry for each customer (or organization), and one for the ISP itself.

Underneath each organization, there must be two `organizationalUnit` entries: `ou=People` and `ou=Groups`. A third, `ou=Devices`, can be present if device data is to be stored for the organization.

Each user entry must have a unique uid value within a given organization. The namespace under this subtree can be partitioned into various ou entries that aggregate user entries in convenient groups (for example, ou=eng, ou=corp). User uid values must still be unique within the entire People subtree.

User entries in the convergence tree are of type inetOrgPerson. The cn, sn, and uid attributes must be present. The uid attribute must be a valid e-mail name (specifically, it must be a valid local-part as defined in RFC822). It is recommended that the cn contain *name initial sn*. It is recommended that the RDN of the user entry be the uid value. User entries must contain the auxiliary class inetUser if they are to be considered enabled for service or valid.

User entries can also contain the auxiliary class inetSubscriber, which is used for account management purposes. If an inetUserStatus attribute is present in an entry and has a value of inactive or deleted, the entry is ignored.

Groups are located under the Groups subtree and consist of LDAP entries of type groupOfUniqueNames.

The Domain Component (dc)Tree

The dc tree contains hierarchical domain entries, each of which is a DNS name component.

Entries that represent the domain name of a customer are overlaid with the LDAP auxiliary class inetDomain. For example, the two LDAP entries dc=customer1,dc=com,o=Internet,o=root and dc=customer2,dc=com,o=Internet,o=root contain the inetDomain class, but dc=com,o=Internet,o=root does not. The latter is present only to provide structure to the tree.

Entries with an inetDomain attribute are called virtual domains. These must have the attribute inetDomainBaseDN filled with the DN of the top level organization entry where the data of this domain is stored in the convergence tree. For example, the virtual domain entry in dc=cust2,dc=com,o=Internet,o=root would contain the attribute inetDomainBaseDN with value o=Cust2,o=isp,o=root.

If an inetDomainStatus attribute is present in an entry and has a value of inactive or deleted, the entry is ignored.

Appendix A

Data Structure Reference

NSAPI uses many data structures which are defined in the `nsapi.h` header file, which is in the directory *server-root*/`plugins/include`.

The NSAPI functions described in Chapter 5, “NSAPI Function Reference,” provide access to most of the data structures and data fields. Before directly accessing a data structure in `naspi.h`, check if an accessor function exists for it.

For information about the privatization of some data structures in iPlanet Web Server 4.*x*, see “Privatization of Some Data Structures,” on page 304.

The rest of this chapter describes some of the frequently used public data structures in `nsapi.h` for your convenience. Note that only the most commonly used fields are documented here for each data structure; for complete details look in `nsapi.h`.

- `session`
- `pblock`
- `pb_entry`
- `pb_param`
- `Session->client`
- `request`
- `stat`
- `shmem_s`
- `cinfo`

Privatization of Some Data Structures

In iPlanet 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 In iPlanet Web Server 4.*x* and 6.*x*, that is, it will be necessary to `#include "nsapi_pvt.h"` in order to build such plugins from source. There is also a small chance that these programs will not be binary compatible with iPlanet 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 iPlanet Web Server 6.0.

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:

```
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;
```

pblock

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 5, “NSAPI Function Reference.” You should not need to write code that access `pblock` data fields directly.

```
typedef struct {
    int hsize;
    struct pb_entry **ht;
} pblock;
```

pb_entry

The pb_entry is a single element in the parameter block.

```
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.

```
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:

```
/*
* session_dns returns the DNS host name of the client for this
* session and inserts it into the client pblock. Returns NULL if
* unavailable.
*/
char *session_dns(Session *sn);
```

request

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;
    pblock *headers;

    /* Server's response headers */
    pblock *srvhdrs;

    /* The object set constructed to fulfill this request */
    httpd_objset *os;

    /* The stat last returned by request_stat_path */
    char *statpath;
    struct stat *finfo;
} Request;
```

stat

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:

```
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 plug-in API activities are `st_size`, `st_atime`, `st_mtime`, and `st_ctime`.

shmem_s

```
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;
```

cinfo

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;
```

Appendix B

MIME Types

This appendix discusses the MIME types file. The sections are:

- Introduction
- Determining the MIME Type
- How the Type Affects the Response
- What Does the Client Do with the MIME Type?
- Syntax of the MIME Types File
- Sample MIME Types File

Introduction

The MIME types file in the `config` directory contains mappings between MIME (Multipurpose Internet Mail Extensions) types and file extensions. For example, the MIME types file maps the extensions `.html` and `.htm` to the type `text/html`:

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

When the iPlanet Web Server receives a request for a resource from a client, it uses the MIME type mappings to determine what kind of resource is being requested.

MIME types are defined by three attributes: language (`lang`), encoding (`enc`), and content-type (`type`). At least one of these attributes must be present for each type. The most commonly used attribute is `type`. The server frequently considers the `type` when deciding how to generate the response to the client. (The `enc` and `lang` attributes are rarely used.)

The default MIME types file is called `mime.types`.

Determining the MIME Type

During the ObjectType step in the request handling process, the server determines the MIME type attributes of the resource requested by the client. Several different server application functions (SAFs) can be used to determine the MIME type, but the most commonly used one is type-by-extension. This function tells the server to look up the MIME type according to the requested resource's file extension in the MIME types table.

The directive in obj.conf that tells the server to look up the MIME type according to the extension is:

```
ObjectType fn=type-by-extension
```

If the server uses a different SAF, such as force-type, to determine the type, then the MIME types table is not used for that particular request.

For more details of the ObjectType step, see Chapter 2, "Syntax and Use of obj.conf."

How the Type Affects the Response

The server considers the value of the type attribute when deciding which Service directive in obj.conf to use to generate the response to the client.

By default, if the type does not start with magnus-internal/, the server just sends the requested file to the client. The directive in obj.conf that contains this instruction is:

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

Note here the use of the special characters *~ to mean "does not match." See Appendix C, "Wildcard Patterns" for details of special characters.

By convention, all values of type that require the server to do something other than just send the requested resource to the client start with magnus-internal/.

For example, if the requested resource's file extension is .map, the type is mapped to magnus-internal/imagemap. If the extension is .cgi, .exe, or .bat, the type is set to magnus-internal/cgi:

```
type=magnus-internal/imagemap       exts=map
type=magnus-internal/cgi            exts=cgi,exe,bat
```

If the type starts with magnus-internal/, the server executes whichever Service directive in obj.conf matches the specified type. For example, if the type is magnus-internal/imagemap, the server uses the imagemap function to generate the response to the client, as indicated by the following directive:

```
Service method=(GET|HEAD) type=magnus-internal/imagemap fn=imagemap
```

If the type is magnus-internal/servlet, the server uses the NSServletService function to generate the response to the client, as indicated by the following directive:

```
Service type="magnus-internal/servlet" fn="NSServletService"
```

What Does the Client Do with the MIME Type?

The Service function generates the data and sends it to the client that made the request. When the server sends the data to the client, it also sends headers. These headers include whichever MIME type attributes are known (which is usually type).

When the client receives the data, it uses the MIME type to decide what to do with the data. For browser clients, the usual thing is to display the data in the browser window.

If the requested resource cannot be displayed in a browser but needs to be handled by another application, its type starts with application/, for example application/octet-stream (for .bin file extensions) or application/x-maker (for .fm file extensions). The client has its own set of user-editable mappings that tells it which application to use to handle which types of data.

For example, if the type is application/x-maker, the client usually handles it by opening Adobe FrameMaker to display the file.

Syntax of the MIME Types File

The first line in the MIME types file identifies the file format and must read:

```
#--Netscape Communications Corporation MIME Information
```

Other non-comment lines have the following format:

```
type=type/subtype exts=[file extensions]
```

- type/subtype is the type and subtype.

- `exts` are the file extensions associated with this type.

Sample MIME Types File

Here is an example of a MIME types file:

```
#--Netscape Communications Corporation MIME Information
# Do not delete the above line. It is used to identify the file type.
type=application/octet-stream      exts=bin,exe
type=application/oda               exts=oda
type=application/pdf               exts=pdf
type=application/postscript        exts=ai,eps,ps
type=application/rtf               exts=rtf
type=application/x-mif             exts=mif,fm
type=application/x-gtar            exts=gtar
type=application/x-shar            exts=shar
type=application/x-tar             exts=tar
type=application/mac-binhex40      exts=hqx

type=audio/basic                   exts=au,snd
type=audio/x-aiff                  exts=aif,aiff,aifc
type=audio/x-wav                   exts=wav

type=image/gif                     exts=gif
type=image/ief                     exts=ief
type=image/jpeg                    exts=jpeg,jpg,jpe
type=image/tiff                    exts=tiff,tif
type=image/x-rgb                   exts=rgb
type=image/x-xbitmap               exts=xbm
type=image/x-xpixmap               exts=xpm
type=image/x-xwindowdump           exts=xwd

type=text/html                     exts=htm,html
type=text/plain                    exts=txt
type=text/richtext                 exts=rtx
type=text/tab-separated-values     exts=tsv
type=text/x-setext                 exts=etx

type=video/mpeg                    exts=mpeg,mpg,mpe
type=video/quicktime               exts=qt,mov
type=video/x-msvideo               exts=avi

enc=x-gzip                         exts=gz
enc=x-compress                     exts=z
enc=x-uuencode                     exts=uu,uue

type=magnus-internal/imagemap      exts=map
type=magnus-internal/parsed-html   exts=shtml
type=magnus-internal/cgi           exts=cgi,exe,bat
type=magnus-internal/jsp           exts=jsp
```

Appendix C

Wildcard Patterns

This appendix describes the format of wildcard patterns used by the iPlanet Web Server.

These wildcards are used in:

- directives in the configuration file `obj.conf` (see Chapter 2, "Syntax and Use of obj.conf")
- various built-in SAFs (see Chapter 3, "Predefined SAFs and the Request Handling Process")
- some NSAPI functions (see Chapter 5, "NSAPI Function Reference")

Wildcard patterns use special characters. If you want to use one of these characters without the special meaning, precede it with a backslash (\) character.

Wildcard Patterns

Table C-1 Wildcard patterns

Pattern	Use
`*`	Match zero or more characters.
`?`	Match exactly one occurrence of any character.
`|`	An or expression. The substrings used with this operator can contain other special characters such as * or $. The substrings must be enclosed in parentheses, for example, (a | b | c), but the parentheses cannot be nested.
`$`	Match the end of the string. This is useful in or expressions.
`[abc]`	Match one occurrence of the characters a, b, or c. Within these expressions, the only character that needs to be treated as a special character is]; all others are not special.

Table C-1 Wildcard patterns

Pattern	Use
`[a-z]`	Match one occurrence of a character between a and z.
`[^az]`	Match any character except a or z.
`*~`	This expression, followed by another expression, removes any pattern matching the second expression.

Wildcard Examples

Table C-2 Wildcard examples

Pattern	Result
`*.netscape.com`	Matches any string ending with the characters `.netscape.com`.
`(quark|energy).netscape.com`	Matches either `quark.netscape.com` or `energy.netscape.com`.
`198.93.9[23].???`	Matches a numeric string starting with either `198.93.92` or `198.93.93` and ending with any 3 characters.
`*.*`	Matches any string with a period in it.
`*~netscape-*`	Matches any string except those starting with `netscape-`.
`*.netscape.com~quark.netscape.com`	Matches any host from domain `netscape.com` except for a single host `quark.netscape.com`.
`*.netscape.com~(quark|energy|neutrino).netscape.com`	Matches any host from domain `.netscape.com` except for hosts `quark.netscape.com`, `energy.netscape.com`, and `neutrino.netscape.com`.
`*.com~*.netscape.com`	Matches any host from domain `.com` except for hosts from subdomain `netscape.com`.
`type=*~magnus-internal/*`	Matches any type that does not start with `magnus-internal/`. This wildcard pattern is used in the file `obj.conf` in the catch-all `Service` directive.

Appendix D

Time Formats

This appendix 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.

Table D-1

Symbol	Meaning
%a	Abbreviated weekday name (3 chars)
%d	Day of month as decimal number (01-31)
%S	Second as decimal number (00-59)
%M	Minute as decimal number (00-59)
%H	Hour in 24-hour format (00-23)
%Y	Year with century, as decimal number, up to 2099
%b	Abbreviated month name (3 chars)
%h	Abbreviated month name (3 chars)
%T	Time "HH:MM:SS"
%X	Time "HH:MM:SS"
%A	Full weekday name
%B	Full month name
%C	"%a %b %e %H:%M:%S %Y"
%c	Date & time "%m/%d/%y %H:%M:%S"
%D	Date "%m/%d/%y"

Table D-1

Symbol	Meaning
%e	Day of month as decimal number (1-31) without leading zeros
%I	Hour in 12-hour format (01-12)
%j	Day of year as decimal number (001-366)
%k	Hour in 24-hour format (0-23) without leading zeros
%l	Hour in 12-hour format (1-12) without leading zeros
%m	Month as decimal number (01-12)
%n	line feed
%p	A.M./P.M. indicator for 12-hour clock
%R	Time "%H:%M"
%r	Time "%I:%M:%S %p"
%t	tab
%U	Week of year as decimal number, with Sunday as first day of week (00-51)
%w	Weekday as decimal number (0-6; Sunday is 0)
%W	Week of year as decimal number, with Monday as first day of week (00-51)
%x	Date "%m/%d/%y"
%y	Year without century, as decimal number (00-99)
%%	Percent sign

Appendix E

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 appendix 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 appendix has the following sections:

- Compliance
- Requests
- Responses
- Buffered Streams

Compliance

iPlanet Web Server 6.0 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.

For more information on the criteria for being conditionally compliant, see the Hypertext Transfer Protocol—HTTP / 1.1 specification (RFC 2068) at:

`http://www.ietf.org/rfc/rfc2068.txt?number=2068`

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. Some commonly used request headers are shown in Table E-1.

Table E-1 Common request headers

Request header	Description
`Accept`	The file types the browser can accept.
`Authorization`	Used if the browser wants to authenticate itself with a server; information such as the username and password are included.

Table E-1 Common request headers

Request header	Description
`User-agent`	The name and version of the browser software.
`Referer`	The URL of the document where the user clicked on the link.
`Host`	The Internet host and port number of the resource being requested.

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.

Some common status codes are shown in Table E-2.

Table E-2 Common HTTP status codes

Status code	Meaning
200	OK; request has succeeded for the method used (GET, POST, HEAD).
201	The request has resulted in the creation of a new resource reference by the returned URI.
206	The server has sent a response to byte range requests.
302	Found. Redirection to a new URL. The original URL has moved. This is not an error; most browsers will get the new page.
304	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.
400	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.
401	Unauthorized. The user requested a document but didn’t provide a valid username or password.
403	Forbidden. Access to this URL is forbidden.
404	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.
408	If the client starts a request but does not complete it within the keep-alive timeout configured in the server, then this reponse will be sent and the connection closed. The request can be repeated with another open connection.
411	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.
413	Some applications (eg. certain NSAPI plug-ins) cannot handle very large amounts of data, so they will return this code.
414	The URI is longer than the maximum the web server is willing to serve.
416	Data was requested outside the range of a file.

Table E-2 Common HTTP status codes (Continued)

Status code	Meaning
`500`	Server error. A server-related error occurred. The server administrator should check the server's error log to see what happened.
`503`	Sent if the quality of service mechanism was enabled and bandwidth or connection limits were attained. The server will then serve requests with that code. See the "quality of service" section.

Response Headers

The response headers contain information about the server and the response data. Common response headers are shown in Table E-3.

.

Table E-3 Common response headers

Response header	Description
`Server`	The name and version of the web server.
`Date`	The current date (in Greenwich Mean Time).
`Last-modified`	The date when the document was last modified.
`Expires`	The date when the document expires.
`Content-length`	The length of the data that follows (in bytes).
`Content-type`	The MIME type of the following data.
`WWW-authenticate`	Used during authentication and includes information that tells the browser software what is necessary for authentication (such as username and password).

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 following features to the iPlanet 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 client can detect the end of the response and re-use 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 5, “NSAPI Function Reference.”). You can specify the following Service SAF parameters that affect stream buffering, which are described in detail in Chapter 3, “Predefined SAFs and the Request Handling Process.”

- `UseOutputStreamSize`
- `flushTimer`

- `ChunkedRequestBufferSize`
- `ChunkedRequestTimeout`

The `UseOutputStreamSize`, `ChunkedRequestBufferSize`, and `ChunkedRequestTimeout` parameters also have equivalent `magnus.conf` directives; see “Chunked Encoding,” on page 281. The `obj.conf` parameters override the `magnus.conf` directives.

NOTE The `UseOutputStreamSize` parameter can be set to zero 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
```

Dynamic Results Caching Functions

The functions described in this appendix allow you to write a results caching plugin for iPlanet 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
- RefershInterval

The RefershInterval value must be a PrIntervalTime type. For more information, see the NSPR reference at:

http://www.mozilla.org/projects/nspr/reference/html/index.html

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 appendix lists the most important functions a results caching plugin can use. For more information, see the following file:

server_root/plugins/include/drnsapi.h

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

```
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

DrHdl hdl is a pointer to an unallocated handle.

RefreshFunc_t ref is a pointer to a cache refresh function. This can be NULL; see the DR_CHECK flag and DR_EXPIR return value for dr_net_write.

FreeFunc_t fre is a pointer to a function that frees an entry.

CompareFunc_t cmp is is a pointer to a key comparator function.

PRUint32 maxEntriesp is the maximum number of entries possible in the cache for a given hdl.

PRIntervalTime maxAgep is the maximum amount of time that an entry is valid. If 0, the cache never expires.

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 a 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

DrHdl hdl is a persistent handle created by the dr_cache_init function.

const char *key is the key to cache, search, or refresh.

PRUint32 klen is the length of the key in bytes.

PRIntervalTime timeout is the expiration time of this entry. If a value of 0 is passed, the maxAge value passed to dr_cache_init is used.

Entry *entry is the not NULL entry to be cached.

Request *rq is a pointer to the request.

Session *sn is a pointer to the session.

Example

```
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_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, `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.

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

```
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

`DrHdl hdl` is a persistent handle created by the `dr_cache_init` function.

`const char *key` is the key to cache, search, or refresh.

`PRUint32 klen` is the length of the key in bytes.

`const char *hdr` is any header data (which can be NULL).

`const char *ftr` is any footer data (which can be NULL).

PRUint32 hlen is the length of the header data in bytes (which can be 0).

PRUint32 flen is the length of the footer data in bytes (which can be 0).

PRIntervalTime timeout is the timeout before this function aborts.

PRUint32 flags is ORed directives for this function (see Flags).

Request *rq is a pointer to the request.

Session *sn is a pointer to the session.

Flags

DR_NONE specifies that no cache is used, so the function works as net_write does; DrHdl can be NULL.

DR_FORCE forces the cache to refresh even if it has not expired.

DR_CHECK 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.

DR_IGNORE ignores cache expiration and sends out the cache entry even if it has expired.

DR_CNTLEN supplies the Content-length header and does a PROTOCOL_START_RESPONSE.

DR_PROTO does a PROTOCOL_START_RESPONSE.

Example

```
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

const char *fileName is the file to be inserted.

const char *hdr is any header data (which can be NULL).

const char *ftr is any footer data (which can be NULL).

PRUint32 hlen is the length of the header data in bytes (which can be 0).

PRUint32 flen is the length of the footer data in bytes (which can be 0).

PRUint32 flags is ORed directives for this function (see Flags).

PRIntervalTime timeout is the timeout before this function aborts.

Request *rq is a pointer to the request.

Session *sn is a pointer to the session.

Flags

FC_CNTLEN supplies the Content-length header and does a PROTOCOL_START_RESPONSE.

FC_PROTO does a PROTOCOL_START_RESPONSE.

Example

```
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;
}
```

Alphabetical List of NSAPI Functions and Macros

C

D

F

L

M

N

P

R

S

U

V

Alphabetical List of Directives in magnus.conf

A

C

D

E

F

H

I

K

L

M

N

P

R

S

T

U

W

Alphabetical List of Pre-defined SAFs

For `Init` SAFs, see Appendix H, "Alphabetical List of Directives in magnus.conf."

A

B

C

D

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W