Developer's Guide to NSAPI

Sun™ ONE Application Server

Version 7

817-2176-10
March 2003

Contents

About This Guide

This book discusses how to use Netscape Server Application Programmer's Interface (NSAPI) to build plugins that use Server Application Functions (SAFs) to extend and modify Sun™ Open Net Environment (Sun ONE) Application Server 7. It also provides a reference of the NSAPI functions you can use to define new SAFs.

This preface contains information about the following topics:

- NSAPI Plugins and J2EE Web Applications
- Who Should Use This Guide
- Using the Documentation
- How This Guide Is Organized
- Documentation Conventions
- Product Support

NOTE	The NSAPI interface is Unstable. An unstable interface may be experimental or transitional, and hence may change incompatibly, be removed, or be replaced by a more stable interface in the next release.

NSAPI Plugins and J2EE Web Applications

In Sun ONE Application Server, NSAPI plugins cannot interoperate with J2EE web applications. Specifically:

- Do not place NSAPI plugins within web application context roots.
- Do not include the output of NSAPI plugins in servlets or JSPs.

- Do not forward requests to NSAPI plugins from servlets or JSPs.
- If you use `security-constraint` and `filter-mapping` features in the default web application, NSAPI features may not work as expected.

Who Should Use This Guide

The intended audience for this guide is the person who develops, assembles, and deploys NSAPI plugins in a corporate enterprise.

This guide assumes you are familiar with the following topics:

- HTTP
- HTML
- NSAPI
- C programming
- Software development processes, including debugging and source code control

Using the Documentation

The Sun ONE Application Server manuals are available as online files in Portable Document Format (PDF) and Hypertext Markup Language (HTML) formats, at:

`http://docs.sun.com/`

The following table lists tasks and concepts described in the Sun ONE Application Server manuals. The left column lists the tasks and concepts, and the right column lists the corresponding manuals.

Table 1 Sun ONE Application Server Documentation Roadmap

For information about	See the following
Late-breaking information about the software and the documentation	Release Notes
Supported platforms and environments	Platform Summary

Table 1 Sun ONE Application Server Documentation Roadmap *(Continued)*

For information about	See the following
Introduction to the application server, including new features, evaluation installation information, and architectural overview.	*Getting Started Guide*
Installing Sun ONE Application Server and its various components (sample applications, Administration interface, Sun ONE Message Queue).	*Installation Guide*
Creating and implementing J2EE applications that follow the open Java standards model on the Sun ONE Application Server 7. Includes general information about application design, developer tools, security, assembly, deployment, debugging, and creating lifecycle modules.	*Developer's Guide*
Creating and implementing J2EE applications that follow the open Java standards model for web applications on the Sun ONE Application Server 7. Discusses web application programming concepts and tasks, and provides sample code, implementation tips, and reference material.	*Developer's Guide to Web Applications*
Creating and implementing J2EE applications that follow the open Java standards model for enterprise beans on the Sun ONE Application Server 7. Discusses EJB programming concepts and tasks, and provides sample code, implementation tips, and reference material.	*Developer's Guide to Enterprise JavaBeans Technology*
Creating clients that access J2EE applications on the Sun ONE Application Server 7	*Developer's Guide to Clients*
Creating web services	*Developer's Guide to Web Services*
J2EE features such as JDBC, JNDI, JTS, JMS, JavaMail, resources, and connectors	*Developer's Guide to J2EE Features and Services*
Creating custom NSAPI plugins	*Developer's Guide to NSAPI*

Table 1 Sun ONE Application Server Documentation Roadmap *(Continued)*

For information about	See the following
Performing the following administration tasks: • Using the Administration interface and the command line interface • Configuring server preferences • Using administrative domains • Using server instances • Monitoring and logging server activity • Configuring the web server plugin • Configuring the Java Messaging Service • Using J2EE features • Configuring support for CORBA-based clients • Configuring database connectivity • Configuring transaction management • Configuring the web container • Deploying applications • Managing virtual servers	*Administrator's Guide*
Editing server configuration files	*Administrator's Configuration File Reference*
Configuring and administering security for the Sun ONE Application Server 7 operational environment. Includes information on general security, certificates, and SSL/TLS encryption. HTTP server-based security is also addressed.	*Administrator's Guide to Security*
Configuring and administering service provider implementation for J2EE CA connectors for the Sun ONE Application Server 7. Includes information about the Administration Tool, DTDs and provides sample XML files.	J2EE CA Service Provider Implementation Administrator's Guide
Migrating your applications to the new Sun ONE Application Server 7 programming model from the Netscape Application Server version 2.1, including a sample migration of an Online Bank application provided with Sun ONE Application Server	*Migration Guide*

Table 1 Sun ONE Application Server Documentation Roadmap *(Continued)*

For information about	See the following
Using Sun ONE Message Queue.	The Sun ONE Message Queue documentation at: http://docs.sun.com/?p=/coll/S1_MessageQueue_30

How This Guide Is Organized

This book has the following chapters and appendices:

- Chapter 1, "Syntax and Use of obj.conf"

  This chapter describes the configuration file obj.conf. The chapter discusses the syntax and use of directives in this file, which instruct the server how to process HTTP requests.

- Chapter 2, "Predefined SAFs and the Request Handling Process"

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

- Chapter 3, "SAFs in the init.conf File"

  This chapter discusses the SAFs you can set in the configuration file `init.conf` to configure the Sun ONE Application server during initialization.

- 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, "Examples of Custom SAFs"

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

- Chapter 6, "NSAPI Function Reference"

  This chapter presents a reference of the NSAPI functions. You use NSAPI functions to define SAFs.

- Chapter 7, "Creating Custom Server-Parsed HTML Tags"

  This chapter explains how to create custom server-parsed HTML tags.

- Appendix A, "Data Structure Reference"

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

- Appendix B, "Wildcard Patterns"

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

- Appendix C, "Time Formats"

  This appendix lists time formats.

- Appendix D, "Dynamic Results Caching Functions"

  This appendix explains how to create a results caching plugin.

- Appendix E, "HyperText Transfer Protocol"

  This appendix gives an overview of HTTP.

- Appendix F, "Alphabetical List of Pre-defined SAFs"
  Appendix G, "Alphabetical List of NSAPI Functions and Macros"

  These appendices provide alphabetical lists for easy lookup of predefined SAFs and NSAPI functions.

Documentation Conventions

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

- General Conventions
- Conventions Referring to Directories

General Conventions

The following general conventions are used in this guide:

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

- **URLs** are given in the format:

  http://*server.domain*/*path*/*file*.html

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

- **Font conventions** include:
  - The `monospace` font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.
  - *Italic* type is used for code variables.
  - *Italic* type is also used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
  - **Bold** type is used as either a paragraph lead-in or to indicate words used in the literal sense.
- **Installation root directories** for most platforms are indicated by *install_dir* in this document. Exceptions are noted in "Conventions Referring to Directories" on page 20.

  By default, the location of *install_dir* on **most** platforms is:
  - Solaris 8 non-package-based Evaluation installations:

    user's home directory`/sun/appserver7`
  - Solaris unbundled, non-evaluation installations:

    `/opt/SUNWappserver7`
  - Windows, all installations:

    `C:\Sun\AppServer7`

  For the platforms listed above, *default_config_dir* and *install_config_dir* are identical to *install_dir*. See "Conventions Referring to Directories" on page 20 for exceptions and additional information.
- **Instance root directories** are indicated by *instance_dir* in this document, which is an abbreviation for the following:

  default_config_dir`/domains/`*domain*`/`*instance*
- **UNIX-specific descriptions** throughout this manual apply to the Linux operating system as well, except where Linux is specifically mentioned.

Conventions Referring to Directories

By default, when using the Solaris 8 and 9 package-based installation and the Solaris 9 bundled installation, the application server files are spread across several root directories. These directories are described in this section.

- **For Solaris 9 bundled installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:
  - *install_dir* refers to `/usr/appserver/`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.
  - *default_config_dir* refers to `/var/appserver/domains`, which is the default location for any domains that are created.
  - *install_config_dir* refers to `/etc/appserver/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.
- **For Solaris 8 and 9 package-based, non-evaluation, unbundled installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:
  - *install_dir* refers to `/opt/SUNWappserver7`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.
  - *default_config_dir* refers to `/var/opt/SUNWappserver7/domains`which is the default location for any domains that are created.
  - *install_config_dir* refers to `/etc/opt/SUNWappserver7/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.

Product Support

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

- The online support web site at:

  `http://www.sun.com/supportraining/`

- The telephone dispatch number associated with your maintenance contract

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

- Description of the problem, including the situation where the problem occurs and its impact on your operation
- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem
- Detailed steps on the methods you have used to reproduce the problem
- Any error logs or core dumps

Chapter 1

Syntax and Use of obj.conf

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

All `obj.conf` files are located in the *instance_dir*/`config` directory. There is one `obj.conf` file for each virtual server, unless several virtual servers are configured to share an `obj.conf` file. Whenever this guide refers to "the `obj.conf` file," it refers to all `obj.conf` files or to the `obj.conf` file for the virtual server being described.

The file named `obj.conf` that lacks a prefix is a template that Sun ONE Application Server uses to create `obj.conf` files for each virtual server. Editing this file does not affect any existing virtual servers, but does affect any subsequently created virtual servers.

By default, each active `obj.conf` file is named *virtual_server_name*-`obj.conf`. Because the default virtual server for a server instance is named after the instance, when you first create a server instance, its `obj.conf` file is named *instance_name*-`obj.conf`. Editing one of these files directly or through the Administration interface changes the configuration of a virtual server.

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 passthrough plugin; the syntax rules for editing `obj.conf`; and a note about example directives.

The sections in this chapter are:

- How the Server Handles HTTP Requests
- 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

NOTE The `obj.conf` interface is Unstable. An unstable interface may be experimental or transitional, and hence may change incompatibly, be removed, or be replaced by a more stable interface in the next release.

How the Server Handles HTTP Requests

Sun ONE Application Server is an application server that accepts and responds to HyperText Transfer Protocol (HTTP) requests. Browsers communicate using several protocols including HTTP, FTP, and gopher. The Sun ONE Application Server handles HTTP and HTTPS specifically.

For more information about the HTTP protocol refer to Appendix E, “HyperText Transfer Protocol”, and 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 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: Standard/7.0
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 HTTP Request Handling Process

When the server first starts up it performs some initialization and then waits for an HTTP request from a client (such as a browser). When it receives a request, it first selects a virtual server. For details about how the virtual server is determined, see the description of the `server.xml` file in the *Sun ONE Application Server Administrator's Configuration File Reference.*

After the virtual server is selected, the `obj.conf` file for the virtual server 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.

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:/Sun/AppServer7/domains/domain1/server1/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.

Sun ONE Application 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 2, "Predefined SAFs and the Request Handling Process". You can also define new SAFs.

The `init.conf` file contains `Init` directive SAFs that initialize the server. For more information, see Chapter 3, "SAFs in the init.conf File".

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 33 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:/Sun/AppServer7/domains/domain1/server1/docs"
```

  This example calls the `document-root` function with a `root` argument of `D:/Sun/AppServer7/domains/domain1/server1/docs`. The function `document-root` function translates the `http://`*hostname*`/` part of the requested URL to the document root, which in this case is `D:/Sun/AppServer7/domains/domain1/server1/docs`. Thus a request for `http://`*hostname*`/doc1.html` is translated to `D:/Sun/AppServer7/domains/domain1/server1/docs/doc1.html`.

- `PathCheck`

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

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

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

- `ObjectType`

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

  The resulting type may be:

  - A common document type such as `text/html` or `image/gif` (for example, the file name extension `.gif` translates to the MIME type `image/gif`).

  - An internal server type. Internal types always begin with `magnus-internal`.

  For example:

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

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

- `Service`

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

  The default `Service` directive is:

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

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

  Another example is:

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

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

- `AddLog`

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

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

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

- `Error`

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

For example:

```
Error fn="send-error" reason="Unauthorized"
path="D:/Sun/AppServer7/domains/domain1/server1/errors/unauthorized.html"
```

In this example, the server sends the file in:

```
D:/Sun/AppServer7/domains/domain1/server1/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="/Sun/AppServer7/domains/domain1/server1/docs/private/*">
```

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

- The successful `NameTrans` directive specifies a `name` argument
- the physical 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 33.

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://`*hostname*`/cgi/`.

```
<Object name="default">
NameTrans fn="pfx2dir" from="/cgi"
dir="D:/Sun/AppServer7/domains/domain1/server1/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://`*hostname/* part of the requested URL to `D:/Sun/AppServer7/domains/domain1/server1/docs/` (which is the document root directory).

```
<Object name="default">
NameTrans fn="document-root"
root="D:/Sun/AppServer7/domains/domain1/server1/docs"
...
</Object>
```

The URL `http://`*hostname/*`internalplan1.html` would be translated to:

```
D:/Sun/AppServer7/domains/domain1/server1/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:/Sun/AppServer7/domains/domain1/server1/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 a variable called `docroot`:

```
<property name=docroot
value="/Sun/AppServer7/domains/domain1/server1/docs/class2/acme" />
```

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

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

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

NOTE Variable substitution is allowed only in an `obj.conf` file. It is not allowed in any other Sun ONE Application Server configuration files.

Any variable referenced in an `obj.conf` file must be defined in the `server.xml` file.

For more information about defining variables, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

Flow of Control in obj.conf

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

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

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://`*hostname*`/` part of the requested URL to a physical directory that has been designated as the server's document root. For example:

```
NameTrans fn="document-root"
root="D:/Sun/AppServer7/domains/domain1/server1/docs"
```

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

The `pfx2dir` (prefix to directory) function is used to set up additional mappings between URLs and directories. For example, the following directive translates the URL `http://`*hostname*`/cgi/` into the directory pathname `D:/Sun/AppServer7/domains/domain1/server1/docs/mycgi/`:

```
NameTrans fn="pfx2dir" from="/cgi"
dir="D:/Sun/AppServer7/domains/domain1/server1/docs/mycgi"
```

Notice that if this directive appeared *after* the one that calls `document-root`, it would never be executed, with the result that the resultant directory pathname would be `D:/Sun/AppServer7/domains/domain1/server1/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://`*hostname*`/cgi/`.

```
<Object name="default">
...
NameTrans fn="pfx2dir" from="/cgi"
dir="D:/Sun/AppServer7/domains/domain1/server1/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:/Sun/AppServer7/domains/domain1/server1/docs/internalplan1.html
```

In this case, the partial path *internal* matches the path:

```
D:/Sun/AppServer7/domains/domain1/server1/docs/internalplan1.html
```

So now the server would start processing the directives in this object before processing the remaining directives in the default object.

PathCheck

After converting the logical URL of the requested resource to a physical 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 the *Sun ONE Application Server Administrator's Configuration File Reference.*

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://`*hostname*`/cgi/` are translated into requests for resources in the directory `D:/Sun/AppServer7/domains/domain1/server1/docs/mycgi/`. Since a `name` is assigned to the request, the server processes `ObjectType` directives in the object named `cgi` before processing the ones in the default object. This object has one `ObjectType` directive, which forces the `type` to be `magnus-internal/cgi`.

```
NameTrans fn="pfx2dir" from="/cgi"
dir="D:/Sun/AppServer7/domains/domain1/server1/docs/mycgi" name="cgi"
<Object name="cgi">
ObjectType fn="force-type" type="magnus-internal/cgi"
Service fn="send-cgi"
</Object>
```

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

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:/`*hostname*`/jos.html`. In this case, all directives executed by the server are in the default object.

- The following `NameTrans` directive translates the requested URL to:

```
D:/Sun/AppServer7/domains/domain1/server1/docs/jos.html
```

```
NameTrans fn="document-root"
root="D:/Sun/AppServer7/domains/domain1/server1/docs"
```

- Assume that the `PathCheck` directives all succeed.
- The following `ObjectType` directive tells the server to look up the resource's MIME type in the MIME types table:

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

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

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

- The server invokes the following `Service` directive. The value of the `type` parameter matches anything that does *not* begin with `magnus-internal/`. (For a list of all wildcard patterns, see Appendix B, "Wildcard Patterns".) This directive sends the requested file, `jos.html`, to the client.

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

Here is an example that involves using another object.

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

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

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

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

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

  `Service fn="index-simple"`

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

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

Default Service Directive

There is usually a `Service` directive that does the default 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 B, “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 init.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 3, “SAFs in the init.conf File”.

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.

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

Order of Directives

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

For `PathCheck` directives, the order within the `PathCheck` section is not so important, since the server executes all `PathCheck` directives. However, 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. Back-slash escapes the next character.

Comments

Comments begin with a pound (#) sign. If you manually add comments to `obj.conf`, then use the Administration interface to make changes to your server, the Administration interface 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:/Sun/AppServer7/domains/domain1/server1/docs/mycgi" name="cgi"
```

Chapter 2

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.

Each SAF has its own arguments, which are passed to it by a 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 that tells the server whether it succeeded, did nothing, or failed.

For a discussion of the use and syntax of `obj.conf`, see the chapter, Chapter 1, "Syntax and Use of obj.conf".

For a list of `Init` (initialization) SAFs, see Chapter 3, "SAFs in the init.conf File".

This chapter includes functions that are part of the core functionality of Sun ONE Application Server. It does not include functions that are available only if additional components, such as 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 F, "Alphabetical List of Pre-defined SAFs".

The following table lists the SAFs that can be used with each directive. The left column lists the directives, and the right column lists the SAFs for each directive.

Table 2-1 Available Server Application Functions (SAFs) Per Directive

Directive	Server Application Functions

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

NameTrans Stage	`assign-name`
	`document-root`
	`home-page`
	`ntrans-j2ee`
	`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`

Table 2-1 Available Server Application Functions (SAFs) Per Directive

Directive	Server Application Functions
ObjectType Stage	check-passthrough force-type set-default-type shtml-hacktype type-by-exp type-by-extension
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 service-j2ee service-passthrough shtml_send upload-file
AddLog Stage	common-log flex-log record-useragent
Error Stage	error-j2ee send-error qos-error

The bucket Parameter

The following performance buckets are predefined in Sun ONE Application 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 SAFs, including those in the `default-bucket`.

You can define additional performance buckets in the `init.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-init` function to generate performance statistics.

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

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 Stage - 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:

- `auth-passthrough` inspects an incoming HTTP (web) request for client information encoded by a `service-passthrough` function running on an intermediate server.
- `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.

auth-passthrough

Applicable in `AuthTrans`-class directives.

The `auth-passthrough` function inspects an incoming HTTP (web) request for client information encoded by a `service-passthrough` function running on an intermediate server. The client information includes:

- The IP address from which the request originated.
- The SSL key size used by the originating client.
- The SSL client certificate presented by the originating client.

When `auth-passthrough` detects encoded client information, it treats the request as if it had arrived directly from the originating client instead of via an intermediate server running `service-passthrough`.

The `auth-passthrough` function is optional on the server instance that receives the request forwarded by `service-passthrough`.

Since `auth-passthrough` makes it possible to override information that may be used for authentication (for example, the IP address of the original request), it is important that only trusted clients and servers be allowed to connect to a server running `auth-passthrough`. As a minimal precaution, only servers behind a corporate firewall should run `auth-passthrough`; no Internet-accessible server should run `auth-passthrough`. Further, if this information about the originating client is not required, `auth-passthrough` should not be used.

Parameters

The following table describes parameters for the `auth-passthrough` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-2 `auth-passthrough` parameters

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

Examples

```
AuthTrans fn=auth-passthrough
```

See Also

`init-passthrough`, `check-passthrough`, `service-passthrough`

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

The following table describes parameters for the `basic-auth` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-3 `basic-auth` parameters

Parameter	Description
`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 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 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 `init.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

The following table describes parameters for the basic-ncsa function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-4 basic-ncsa parameters

Parameter	Description
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.

Table 2-4 `basic-ncsa` parameters

Parameter	Description
`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=/Sun/AppServer7/domains/domain1/server1/userdb/rs

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

AuthTrans fn=basic-ncsa auth-type=basic
userfile=/Sun/AppServer7/domains/domain1/server1/.htpasswd
grpfile=/Sun/AppServer7/domains/domain1/server1/.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

The following table describes parameters for the `get-sslid` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-5 `get-sslid` parameters

Parameter	Description
`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.

For more information, see the *Sun ONE Application Server Performance Tuning, Sizing, and Scaling Guide*.

Parameters

The following table describes parameters for the `qos-handler` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-6 `qos-handler` parameters

Parameter	Description
`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/Sun/AppServer7/domains/domain1/server1/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://`*host_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.
- `ntrans-j2ee` determines whether a request maps to a Java web application context.
- `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://`*hostname*`/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

The following table describes parameters for the `assign-name` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-7 `assign-name` parameters

Parameter	Description
`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.

Table 2-7 `assign-name` parameters

Parameter	Description
`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, 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://`*hostname*`/` part of the path is replace by the physical pathname for the document root.

When the server receives a request for `http://`*hostname*`/somepath/somefile`, the `document-root` function replaces `http://`*hostname*`/` with the value of its `root` parameter. For example, if the document root directory is `/usr/Sun/AppServer7/domains/domain1/server1/docs`, then when the server receives a request for `http://`*hostname*`/a/b/file.html`, the `document-root` function translates the pathname for the requested resource to:

```
/usr/Sun/AppServer7/domains/domain1/server1/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

The following table describes parameters for the `document-root` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-8 `document-root` parameters

Parameter	Description
`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/Sun/AppServer7/domains/domain1/server1/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

The following table describes parameters for the home-page function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-9 home-page parameters

Parameter	Description
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"
```

ntrans-j2ee

Applicable in NameTrans-class directives.

The ntrans-j2ee function determines whether a request maps to a Java web application context.

Parameters

The following table describes parameters for the ntrans-j2ee function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-10 ntrans-j2ee parameters

Parameter	Description
name	is a named object in obj.conf whose directives are applied to requests made to Java web applications.

Table 2-10 `ntrans-j2ee` parameters

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

Examples

```
NameTrans fn="ntrans-j2ee" name="j2ee"
```

See Also

`init-j2ee`, `service-j2ee`, `error-j2ee`

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

The following table describes parameters for the `pfx2dir` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-11 `pfx2dir` parameters

Parameter	Description
`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.

Table 2-11 `pfx2dir` parameters

Parameter	Description
`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 Windows, 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://`*hostname*`/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://`*hostname*`/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://`*hostname*`/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

The following table describes parameters for the `redirect` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-12 `redirect` parameters

Parameter	Description
`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).

Table 2-12 `redirect` parameters

Parameter	Description
`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`. For more information about `util_uri_escape`, see Chapter 6, "NSAPI Function Reference".
`bucket`	optional, common to all `obj.conf` functions

Examples

In the first example, any request for `http://`*hostname*`/`*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://`*hostname*`/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

The following table describes parameters for the `strip-params` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-13 `strip-params` parameters

Parameter	Description
`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 `init.conf`.

Parameters

The following table describes parameters for the `unix-home` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-14 `unix-home` parameters

Parameter	Description
`from`	is the URL prefix to translate, usually "`/~`".
`subdir`	is the subdirectory within the user's home directory that contains their 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.

Table 2-14 `unix-home` parameters

Parameter	Description
`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 that access to the requested resource is forbidden.
- `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 that access to the requested resource is forbidden.

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

The following table describes parameters for the `check-acl` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-15 `check-acl` parameters

Parameter	Description
`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

The following table describes parameters for the deny-existence function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-16 deny-existence parameters

Parameter	Description
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/Sun/AppServer7/domains/domain1/server1/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

The following table describes parameters for the `find-index` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-17 `find-index` parameters

Parameter	Description
`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

The following table describes parameters for the `find-links` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-18 `find-links` parameters

Parameter	Description
`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.
checkFileExistence	check linked file for existence and abort request with `403 (forbidden)` if this check fails.
`bucket`	optional, common to all `obj.conf` functions

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

The following table describes parameters for the find-pathinfo function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-19 find-pathinfo parameters

Parameter	Description
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

The following table describes parameters for the get-client-cert function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-20 `get-client-cert` parameters

Parameter	Description
`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:/Sun/AppServer7/domains/domain1/server1/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:/Sun/AppServer7/domains/domain1/server1/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:/Sun/AppServer7/domains/domain1/server1/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.

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:/Sun/AppServer7/domains/domain1/server1/docs are subject to the access rules in the file my.nsconfig.

```
<Object ppath="D:/Sun/AppServer7/domains/domain1/server1/docs/*">
PathCheck fn="load-config" file="my.nsconfig" descend=1
basedir="D:/Sun/AppServer7/domains/domain1/server1/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

The following table describes parameters for the load-config function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-21 load-config parameters

Parameter	Description
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.

Table 2-21 `load-config` parameters

Parameter	Description
`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://`*hostname*`/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:/Sun/AppServer7/domains/domain1/server1/docs/nikki/` or a subdirectory thereof, it searches for a configuration file called `checkaccess.nsconfig`.

The server starts the search in the directory `D:/Sun/AppServer7/domains/domain1/server1/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:/Sun/AppServer7/domains/domain1/server1/docs/nikki" descend="1"
</Object>
```

nt-uri-clean

Applicable in `PathCheck`-class directives.

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

Parameters

The following table describes parameters for the `nt-uri-clean` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-22 `nt-uri-clean` parameters

Parameter	Description
tildeok	if present, allows tilde"~" characters in URIs. This is a potential security risk on the Windows 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.
`bucket`	optional, common to all `obj.conf` functions

Examples

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

See Also

`unix-uri-clean`

ntcgicheck

Applicable in `PathCheck`-class directives.

Windows 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

The following table describes parameters for the `ntcgicheck` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-23 `ntcgicheck` parameters

Parameter	Description
`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

The following table describes parameters for the `require-auth` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-24 `require-auth` parameters

Parameter	Description
`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.

Table 2-24 `require-auth` parameters

Parameter	Description
`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 servlet in its own namespace, for example.

`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

The following table describes parameters for the `set-virtual-index` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-25 `set-virtual-index` parameters

Parameter	Description
`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

The following table describes parameters for the `ssl-check` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-26 `ssl-check` parameters

Parameter	Description
`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

The following table describes parameters for the `ssl-logout` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-27 `ssl-logout` parameters

Parameter	Description
`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

The following table describes parameters for the unix-uri-clean function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-28 unix-uri-clean parameters

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

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:

- `check-passthrough` checks to see if the requested resource is available on the local server.
- `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.

check-passthrough

Applicable in `ObjectType`-class directives.

The `check-passthrough` function checks to see if the requested resource (for example, the HTML document or GIF image) is available on the local server. If the requested resource does not exist locally, `check-passthrough` sets the type to indicate that the request should be passed to another server for processing by `service-passthrough`.

Parameters

The following table describes parameters for the `check-passthrough` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-29 `check-passthrough` parameters

Parameter	Description
`type`	(optional) is the type assigned when the requested resource does not exist. The default is `magnus-internal/passthrough`.

Table 2-29 `check-passthrough` parameters

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

Example

```
ObjectType fn="check-passthrough"
```

See Also

`init-passthrough`, `auth-passthrough`, `service-passthrough`

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

The following table describes parameters for the `force-type` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-30 `force-type` parameters

Parameter	Description
`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).

Table 2-30 `force-type` parameters

Parameter	Description
`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

The following table describes parameters for the `set-default-type` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-31 `set-default-type` parameters

Parameter	Description
`enc`	(optional) is the encoding assigned to a matching request (the `content-encoding` header).

Table 2-31 `set-default-type` parameters

Parameter	Description
`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

The following table describes parameters for the `shtml-hacktype` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-32 `shtml-hacktype` parameters

Parameter	Description
`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

The following table describes parameters for the `type-by-exp` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-33 `type-by-exp` parameters

Parameter	Description
`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` and MIME types files, see the *Sun ONE Application Server Administrator's Configuration File Reference.*

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

The following table describes parameters for the `type-by-extension` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-34 `type-by-extension` parameters

Parameter	Description
`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 init.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 290. The `UseOutputStreamSize`, `ChunkedRequestBufferSize`, and `ChunkedRequestTimeout` parameters also have equivalent `init.conf` directives; see the *Sun ONE Application Server Administrator's Configuration File Reference*. The `obj.conf` parameters override the `init.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 B, "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.
- `service-j2ee` services requests made to Java web applications.
- `service-passthrough` forwards a request to another server for processing.
- `shtml_send` parses an HTML file for server-parsed html commands.
- `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

The following table describes parameters for the `add-footer` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-35 `add-footer` parameters

Parameter	Description
`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:/Sun/AppServer7/domains/domain1/server1/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

The following table describes parameters for the `add-header` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-36 `add-header` parameters

Parameter	Description
`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

Table 2-36 `add-header` parameters

Parameter	Description
`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:/Sun/AppServer7/domains/domain1/server1/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

The following table describes parameters for the `append-trailer` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-37 `append-trailer` parameters

Parameter	Description
`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 C, "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

The following table describes parameters for the `imagemap` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-38 `imagemap` parameters

Parameter	Description
`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 init.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, init.conf must initialize fancy (or common) indexing by invoking init-cgi 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.

The following table describes which icon is displayed based on the content-type of the file. The left column lists the content-type values, and the right column lists the .gif files for the displayed icons.

Table 2-39 content-type icons

Content Type	Icon
"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

The following table describes parameters for the index-common function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-40 `index-common` parameters

Parameter	Description
`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

The following table describes parameters for the `index-simple` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-41 `index-simple` parameters

Parameter	Description
`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

The following table describes parameters for the `key-toosmall` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-42 `key-toosmall` parameters

Parameter	Description
`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

The following table describes parameters for the `list-dir` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-43 `list-dir` parameters

Parameter	Description
`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

Table 2-43 `list-dir` parameters

Parameter	Description
`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

The following table describes parameters for the `make-dir` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-44 `make-dir` parameters

Parameter	Description
`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

Table 2-44 `make-dir` parameters

Parameter	Description
`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

The following table describes parameters for the `query-handler` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-45 `query-handler` parameters

Parameter	Description
`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

Table 2-45 `query-handler` parameters

Parameter	Description
`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

The following table describes parameters for the `remove-dir` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-46 `remove-dir` parameters

Parameter	Description
`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

Table 2-46 `remove-dir` parameters

Parameter	Description
`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

The following table describes parameters for the `remove-file` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-47 `remove-file` parameters

Parameter	Description
`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

The following table describes parameters for the `rename-file` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-48 `rename-file` parameters

Parameter	Description
`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 "CGI to NSAPI Conversion" on page 170.

For additional information about CGI, see the *Sun ONE Application Server Administrator's Guide* and the *Sun ONE Application Server Developer's Guide to Web Applications*.

You can change the timing used to flush the CGI buffer in these ways:

- Adjust the interval between flushes using the `flushTimer` parameter
- Adjust the buffer size using the `UseOutputStreamSize` parameter
- Force Sun ONE Application 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 290.

Parameters

The following table describes parameters for the `send-cgi` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-49 `send-cgi` parameters

Parameter	Description
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 `init.conf`.
dir	(UNIX only) Specifies the directory to chdir to after chroot but before execution begins.

Table 2-49 `send-cgi` parameters

Parameter	Description
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 the *Sun ONE Application Server Administrator's Configuration File Reference*.

```
<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 B, "Wildcard Patterns".

Parameters

The following table describes parameters for the `send-file` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-50 `send-file` parameters

Parameter	Description
`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

The following table describes parameters for the `send-range` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-51 `send-range` parameters

Parameter	Description
`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 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. For information about shell CGI programs, consult the *Sun ONE Application Server Administrator's Guide*.

Parameters

The following table describes parameters for the `send-shellcgi` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-52 `send-shellcgi` parameters

Parameter	Description
`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 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 *Sun ONE Application Server Administrator's Guide*.

Parameters

The following table describes parameters for the `send-wincgi` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-53 `send-wincgi` parameters

Parameter	Description
`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 47).

To read the report, point the browser here:

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

Parameters

The following table describes parameters for the `service-dump` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-54 `service-dump` parameters

Parameter	Description
`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-init`

service-j2ee

Applicable in `Service`-class directives.

The `service-j2ee` function services requests made to Java web applications.

Parameters

The following table describes parameters for the `service-j2ee` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-55 `service-j2ee` parameters

Parameter	Description
`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="ntrans-j2ee" name="j2ee"
...
</Object>

<Object name=j2ee>
Service fn="service-j2ee"
</Object>
```

See Also

`init-j2ee`, `ntrans-j2ee`, `error-j2ee`

service-passthrough

Applicable in `Service`-class directives.

The `service-passthrough` function forwards a request to another server for processing. This function can be configured to use SSL or non-SSL (HTTPS or HTTP) connections to the remote server independently of the type of connection the original request arrived on.

The `service-passthrough` function encodes information about the originating client that may be decoded by an `auth-passthrough` function running on the remote server. Use of `auth-passthrough` is optional.

When multiple remote servers are configured, `service-passthrough` chooses a single remote server from the list on a request-by-request basis.

Parameters

The following table describes parameters for the `service-passthrough` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-56 `service-passthrough` parameters

Parameter	Description
`servers`	A quoted, space-delimited list of the servers that receive the forwarded requests. Individual server names may optionally be: • Prefixed with `http://` or `https://` to indicate the desired protocol. • Suffixed with a colon and an integer (for example `:8000`) to indicate the desired port.
`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

Here is a stand-alone example of the service-passthrough function:

```
Service fn="service-passthrough" servers="http://server1
http://server2"
```

A service-passthrough function is typically used in combination with other directives in the obj.conf configuration file as follows:

```
<Object name="passthrough">
ObjectType fn="force-type" type="magnus-internal/passthrough"
PathCheck fn="deny-existence" path="*/WEB-INF/*"
Service type="magnus-internal/passthrough" fn="service-passthrough"
servers="http://192.168.1.100:8000 http://192.168.1.101:8000"
Error reason="Bad Gateway" fn="send-error" uri="$docroot/badgateway.html"
</Object>

<Object name="default">
...
NameTrans fn="assign-name" from="(/webapp1|/webapp1/*)" name="passthrough"
...
</Object>
```

This example forwards any request for the web application deployed at the URI /webapp1 to one of the backend application servers at IP addresses 192.168.1.100 and 192.168.1.101. If the backend application server is down, the local HTML file badgateway.html is displayed instead.

If you want the server running service-passthrough to serve files it has access to and forward only those requests it cannot satisfy to the backend application servers, change the ObjectType line as follows:

```
ObjectType fn="check-passthrough"
type="magnus-internal/passthrough"
```

See Also

init-passthrough, auth-passthrough, check-passthrough, assign-name, force-type, deny-existence

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) is loaded. Refer to the *Sun ONE Application Server Developer's Guide to Web Applications* for server-parsed HTML commands.

Parameters

The following table describes parameters for the `shtml_send` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-57 `shtml_send` parameters

Parameter	Description
`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
```

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

The following table describes parameters for the upload-file function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-58 upload-file parameters

Parameter	Description
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 Sun ONE Application 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

The following table describes parameters for the `common-log` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-59 `common-log` parameters

Parameter	Description
`name`	(optional) gives the name of a log file, which must have been given as a parameter to the `init-clf` function in `init.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 `init.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 Sun ONE Application 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

The following table describes parameters for the `flex-log` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-60 `flex-log` parameters

Parameter	Description
`name`	(optional) gives the name of a log file, which must have been given as a parameter to the `flex-init` function in `init.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 `init.conf` file. The value of `iponly` has no significance, as long as it exists; you may use `iponly=1`.

Table 2-60 `flex-log` parameters

Parameter	Description
`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 the client was used for this transaction. For information about rotating logs, see `flex-rotate-init`.

Parameters

The following table describes parameters for the `record-useragent` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-61 `record-useragent` parameters

Parameter	Description
`name`	(optional) gives the name of a log file, which must have been given as a parameter to the `init-clf` function in `init.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:

- `error-j2ee` handles errors that occur during execution of J2EE applications and modules deployed to the Sun ONE Application Server.
- `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.

error-j2ee

Applicable in `Error`-class directives.

The `error-j2ee` function handles errors that occur during execution of web applications deployed to the Sun ONE Application Server individually or as part of full J2EE applications.

Parameters

The following table describes parameters for the `error-j2ee` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-62 `error-j2ee` parameters

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

See Also

`init-j2ee`, `ntrans-j2ee`, `service-j2ee`

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

The following table describes parameters for the `send-error` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-63 `send-error` parameters

Parameter	Description
`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.

Table 2-63 `send-error` parameters

Parameter	Description
`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

Example

```
Error fn=send-error code=401
path=/Sun/AppServer7/domains/domain1/server1/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.

For more information, see the performance chapter of the *Sun ONE Application Server Administrator's Guide*.

Parameters

The following table describes parameters for the `qos-error` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 2-64 `qos-error` parameters

Parameter	Description
`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 3

SAFs in the init.conf File

When the Sun ONE Application Server starts up, it looks in a file called `init.conf` in the *instance_dir*/`config` directory to establish a set of global variable settings that affect the server's behavior and configuration. Sun ONE Application Server executes all the directives defined in `init.conf`. The order of the directives is not important.

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

NOTE	The `init.conf` interface is Unstable. An unstable interface may be experimental or transitional, and hence may change incompatibly, be removed, or be replaced by a more stable interface in the next release.

This chapter lists the `Init` SAFs that can be specified in `init.conf` in Sun ONE Application Server 7. For an alphabetical list of all SAFs, see Appendix F, "Alphabetical List of Pre-defined SAFs". For information about the other, non-SAF directives in `init.conf`, see the *Sun ONE Application Server Administrator's Configuration File Reference.*

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

The `Init` directives are SAFs, like `obj.conf` directives, and have SAF syntax rather than the simpler *variable value* syntax of other `init.conf` directives. They are located in `init.conf` because, like other `init.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-j2ee` initializes the Java subsystem.
- `init-passthrough` initializes the passthrough plugin.
- `init-uhome` loads user home directory information.
- `load-modules` loads shared libraries into the server.
- `perf-init` enables system performance measurement via performance buckets.
- `pool-init` configures pooled memory allocation.
- `register-http-method` lets you extend the HTTP protocol by registering new HTTP methods.
- `stats-init` enables reporting of performance statistics in XML format.

- `thread-pool-init` configures an additional thread pool.

cindex-init

Applicable in `Init`-class directives.

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

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

Parameters

The following table describes parameters for the `cindex-init` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-1 `cindex-init` parameters

Parameter	Description
`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.

Table 3-1 `cindex-init` parameters

Parameter	Description
`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 47 and the `service-dump` function). This function works only if the `perf-init` function is enabled.

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

Parameters

The following table describes parameters for the `define-perf-bucket` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-2 `define-perf-bucket` parameters

Parameter	Description
`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

The following table describes parameters for the `dns-cache-init` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-3 `dns-cache-init` parameters

Parameter	Description
`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

The following table describes parameters for the `flex-init` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-4 `flex-init` parameters

Parameter	Description

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/Sun/AppServer7/domains/domain1/server1/logs/access"`
	
	`mylogfile = "access.log"`
	
	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 functions to manipulate pblocks.)

Any additional text is treated as literal text, so you can add to the line to make it more readable. See the "Typical components of flex-init formatting" table. 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.

The following table shows `flex-init` formatting components. The left column lists `flex-log` options, and the right column lists the equivalent `flex-init` components.

Table 3-5 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%\"`
Status	`%Req->srvhdrs.clf-status%`
Response content length	`%Req->srvhdrs.content-length%`
Response content type	`%Req->srvhdrs.content-type%`
Referer header	`\"%Req->headers.referer%\"`

Table 3-5 Typical components of flex-init formatting

Flex-log option	Component
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/Sun/AppServer7/domains/domain1/server1/logs/access`.

```
Init fn=flex-init
access="/usr/Sun/AppServer7/domains/domain1/server1/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 `/usr/Sun/AppServer7/domains/domain1/server1/logs/extended`.

```
Init fn=flex-init
extended="/usr/Sun/AppServer7/domains/domain1/server1/logs/extend
ed" 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` `init.conf` directive to 4 seconds (see the *Sun ONE Application Server Administrator's Configuration File Reference*) frees the request handling threads to quickly write the log data.

```
Init fn=flex-init buffer-size=16384 num-buffers=2000
access="/usr/Sun/AppServer7/domains/domain1/server1/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 server logs and the `common-log`, `flex-log`, and `record-useragent` `AddLog` SAFs. Call this function in the `Init` section of `init.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

The following table describes parameters for the `flex-rotate-init` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-6 `flex-rotate-init` parameters

Parameter	Description
`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.

Table 3-6 `flex-rotate-init` parameters

Parameter	Description
`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 server 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

The following table describes parameters for the `init-cgi` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-7 `init-cgi` parameters

Parameter	Description
`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, Sun ONE Application Server looks in the following locations, in the following order: • `../private/Cgistub`, relative to the server instance's `config` directory • `../../bin/https/bin/Cgistub`, relative to the server's installation directory 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. If present, the `../private` directory must be owned by the server user and have permissions `d??x------`. This prevents other users (for example, users with shell accounts or CGI access) from using Cgistub to set their uid. For information about installing an suid Cgistub, see the *Sun ONE Application Server Developer's Guide to Web Applications.*
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

The following table describes parameters for the `init-clf` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-8 `init-clf` parameters

Parameter	Description

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/Sun/AppServer7/domains/`
	`domain1/server1/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/Sun/AppServer7/domains/domain1/server1/logs/access
Init fn=init-clf templog=/tmp/mytemplog templog2=/tmp/mytemplog2
```

See Also

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

init-j2ee

Applicable in `Init`-class directives.

The `init-j2ee` function initializes the Java subsystem.

Parameters

This function requires a `LateInit=yes` parameter.

Example

```
Init fn="load-modules" shlib="install_dir/lib/libj2eeplugin.so"
funcs="init-j2ee,ntrans-j2ee,service-j2ee,error-j2ee"
shlib_flags="(global|now)"

Init fn="init-j2ee" LateInit=yes
```

See Also

`ntrans-j2ee`, `service-j2ee`, `error-j2ee`

init-passthrough

Applicable in `Init`-class directives.

The `init-passthrough` function initializes the passthrough plugin. This function must be called before the passthrough plugin can be used.

Parameters

none

Example

```
Init fn="load-modules" shlib="c:/install_dir/lib/passthrough.dll"
funcs="init-passthrough,auth-passthrough,check-passthrough,
service-passthrough" NativeThread="no"

Init fn="init-passthrough"
```

See Also

`auth-passthrough`, `check-passthrough`, `service-passthrough`

init-uhome

Applicable in `Init`-class directives.

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

Parameters

The following table describes parameters for the `init-uhome` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-9 `init-uhome` parameters

Parameter	Description
`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 file to load.

Parameters

The following table describes parameters for the `load-modules` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-10 `load-modules` parameters

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

Table 3-10 `load-modules` parameters

Parameter	Description
`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, Windows only) 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
```

perf-init

Applicable in `Init`-class directives.

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

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

Parameters

The following table describes parameters for the `perf-init` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-11 perf-init parameters

Parameter	Description
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

The following table describes parameters for the `pool-init` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-12 `pool-init` parameters

Parameter	Description
`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

The following table describes parameters for the `register-http-method` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-13 `register-http-method` parameters

Parameter	Description
`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.

Parameters

The following table describes parameters for the `stats-init` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-14 `stats-init` parameters

Parameter	Description
`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 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

`service-dump`

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

In addition, native thread pool parameters can be added to the `init.conf` file for convenience. For more information, see the *Sun ONE Application Server Administrator's Configuration File Reference.*

Parameters

The following table describes parameters for the `thread-pool-init` function. The left column lists the parameter name, and the right column describes what the parameter does.

Table 3-15 `thread-pool-init` parameters

Parameter	Description
`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`

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 Sun ONE Application 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:

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

Before writing custom SAFs, you should familiarize yourself with the request handling process. Also, before writing a custom SAF, check if a built-in SAF already accomplishes the tasks you have in mind. After you write the SAF, you must add a directive to `obj.conf` so that your new function gets invoked by the server at the appropriate time.

See Chapter 1, "Syntax and Use of obj.conf", and Chapter 2, "Predefined SAFs and the Request Handling Process", for information about request handling, built-in SAFs, and the `obj.conf` file.

For a complete list of the NSAPI routines for implementing custom SAFs, see Chapter 6, "NSAPI Function Reference".

Future Compatibility Issues

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

- Instruct plugin users how to edit the configuration files (such as `init.conf` and `obj.conf`) manually. Do not have the plugin installation software edit these configuration files.
- Keep the source code so you can recompile the plugin.

The SAF Interface

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

Here is the C interface for a SAF:

```
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 153 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=/Sun/AppServer7/domains/domain1/server1/userdb/rs
```

In this case, the pb parameter passed to basic-ncsa contains name/value pairs that correspond to auth-type=basic and dbm=/Sun/AppServer7/domains/domain1/server1/userdb/rs.

NSAPI provides a set of functions for working with pblock data structures. For example, pblock_findval() returns the value for a given name in a pblock. See "Parameter Block Manipulation Routines" on page 162 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 6, "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 6, "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 162. Chapter 6, “NSAPI Function Reference”, provides information about all of the routines available.

For examples of custom SAFs, see *install_dir*/`samples/nsapi`, and also see Chapter 5, “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 150.

The Sun ONE Application 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 162.

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 166.

Compile and Link

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

Follow these guidelines for compiling and linking.

Include Directory and nsapi.h File

Add the *install_dir*`/include` directory to your makefile to include the `nsapi.h` file.

Libraries

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

The following table lists the library that you need to link to. The left column lists the platform, and the right column lists the library.

Table 4-1 Libraries

Platform	Library
Windows	`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

The following table lists the options for generating a shared library. The left column lists the platform, and the right column lists the options.

Table 4-2 Linker commands and options

Platform	Options
Solaris Sparc	`ld -G` or `cc -G`
Windows	`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 to specify which directories should be searched for shared objects during runtime to resolve symbols.

The following table lists the linker flags. The left column lists the platform, and the right column lists the flags.

Table 4-3 Linker flags

Platform	Flags
Solaris Sparc	`-R` *dir*`:`*dir*
Windows	(no flags, but the `appservd40.dll` file must be in the system PATH variable)
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

The following table lists the flags and defines that you need to use for compilation of your source code. The left column lists the platform, and the right column lists the flags and defines.

Table 4-4 Compiler flags and defines

Platform	Flags/Defines
Solaris Sparc	`-DXP_UNIX -D_REENTRANT -KPIC -DSOLARIS`
Windows	`-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`

Compiling iPlanet Web Server 6.x Plugins on Solaris

You must recompile and relink a plugin for use in Sun ONE Application Server 7 if it meets all of these conditions:

- The plugin was developed on the Solaris platform.
- The plugin was developed for use with iPlanet Web Server version 6.*x* or earlier.
- The plugin is written in C++.
- The plugin uses exceptions.

Once recompiled for Sun ONE Application Server 7, the plugin will no longer work in iPlanet Web Server 6.*x*. Therefore, you must maintain separate binary versions for iPlanet Web Server 6.*x* and Sun ONE Application Server 7.

To build a plugin for Sun ONE Application Server 7, you must use version 5.0 or higher of the Sun WorkShop C/C++ compiler (also called Forte C/C++). Do not specify the `-compat` flag (`-compat=4` is the same as `-compat`, but `-compat=5` is the same as not specifying the `-compat` flag).

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 *install_dir*/`samples/nsapi` 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 `appservd` main executable, must be built with the `-G` option, which specifies that symbols must be resolved at runtime.

Previous versions of iPlanet Web Server, however, were built on AIX 4.1, which did not support native runtime-linking. Web Server had specific additional software (provided by IBM AIX development) 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 Sun ONE Application Server, add an `Init` directive that invokes the `load-modules` SAF to `init.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 `init.conf`:

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

Instruct the Server to Call the SAFs

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

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

- *Directive* is one of the server directives, such as `AuthTrans`, `Service`, and so on.
- *function-name* is the name of the SAF to execute.
- *nameN*`="`*valueN*`"` are the names and values of parameters which are passed to the SAF.

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

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

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

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

For example, suppose your plugin defines two new SAFs, `do_small_anim()` and `do_big_anim()` which both take `speed` parameters. These functions run animations. All files to be treated as small animations reside in the directory:

```
D:/Sun/AppServer7/domains/domain1/server1/docs/animations/small
```

while all files to be treated as full screen animations reside in the directory:

```
D:/Sun/AppServer7/domains/domain1/server1/docs/animations/fullscrn
```

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:/Sun/AppServer7/domains/domain1/server1/docs/animations/small"
name="small_anim"
NameTrans fn=pfx2dir from="/animations/fullscrn"
dir="D:/Sun/AppServer7/domains/domain1/server1/docs/animations/fullscrn"
name="fullscrn_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="fullscrn_anim">
Service fn=do_big_anim speed=20
</Object>
```

Reconfigure the Server

After modifying `obj.conf`, you need to reconfigure the server. See the *Sun ONE Application Server Administrator's Guide* 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://`*hostname*`/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 server log to help with debugging.

Overview of NSAPI C Functions

NSAPI provides a set of C functions that are used to implement SAFs. They serve several purposes. They provide platform-independence across Sun ONE Application 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 6, "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 3, "SAFs in the init.conf File".

- `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.sun.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 an SAF using NSAPI. Since the CGI environment variables are not available to NSAPI, you'll retrieve them from the NSAPI parameter blocks.

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.

The following table indicates how each CGI environment variable can be obtained in NSAPI. The left column lists the CGI variables, and the right column lists the NSAPI parameter blocks.

Table 4-5 Parameter blocks for CGI variables

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

Table 4-5 Parameter blocks for CGI variables

CGI getenv()	NSAPI
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
Sun ONE 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);
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

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, the role of the configuration file `obj.conf`, and the pre-defined SAFs that are available. For details on both these topics, see Chapter 1, “Syntax and Use of obj.conf”, and Chapter 2, “Predefined SAFs and the Request Handling Process”.

For a list of the NSAPI functions for creating new SAFs, see Chapter 6, “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 *install_dir*/samples/nsapi 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 Sun ONE Application Server by adding the following directive in the Init section of init.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 init.conf, you always need to restart the Sun ONE Application 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 user names and passwords and checks a given user's password against the one in the static data array. The *userdb* parameter is not used in this function.

AuthTrans directives work in conjunction with PathCheck directives. Generally, an AuthTrans function checks if the 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 Sun ONE Application Server, add the following Init directive to init.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 *install_dir*/samples/nsapi directory.

NameTrans Example

The `ntrans.c` file in the *install_dir*/`samples/nsapi` 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://`*host_name*`/cgi/marketing,/jan/releases/hardware`

In this case, the URL of the requested resource (which would be a CGI program) is `http://`*hostname*`/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 Sun ONE Application Server, add the following `Init` directive to `init.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 *install_dir*/`samples/nsapi` directory.

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 `init.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 `init.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 *install_dir*`/samples/nsapi` directory.

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 `init.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 *install_dir*`/samples/nsapi` directory.

Service Example

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

For a more complex example, see the file `service.c` in the `examples` directory, which is discussed in "More Complex Service Example" on page 182.

Installing the Example

To load the shared object containing your functions add the following line in the `Init` section of the `init.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 `init.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 init.conf
*/
NSAPI_PUBLIC int init-simple-service(pblock *pb, Session *sn,
Request *rq)
{
   /* Get the message from the parameter in the directive in
    * init.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. 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 `init.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
```

Source Code

The source code is in `service.c` in the *install_dir/*`samples/nsapi` 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 `init.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 `init.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 *install_dir*/`samples/nsapi` directory.

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 *Sun ONE Application Server Performance Tuning, Sizing, and Scaling Guide*.

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 *install_dir*/`samples/nsapi` directory.

Chapter 6

NSAPI Function Reference

This chapter lists all the public C functions and macros of NSAPI in alphabetic order. These are the functions you use when writing your own Server Application Functions (SAFs).

See Chapter 2, "Predefined SAFs and the Request Handling Process", for a list 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 Sun ONE Application Server 7.

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 `init.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 existing 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 a server 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 `init.conf` file. For more information, see the *Sun ONE Application Server Administrator's Configuration File Reference.*

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 `init.conf` file. For more information, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

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 `init.conf` file. For more information, see the *Sun ONE Application Server Administrator's Configuration File Reference*.

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 `*.sun.com` and you want to make sure that a string matches it, such as `foo.sun.com`.

For information about wildcard patterns you can use in this function, see Appendix B, "Wildcard Patterns".

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 `*.sun.com` and you want to make sure that a string matches it, such as `foo.sun.com`.

For information about wildcard patterns you can use in this function, see Appendix B, "Wildcard Patterns".

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/sun/*";
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 `*.sun.com` and you want to make sure that a string matches it, such as `foo.sun.com`.

For information about wildcard patterns you can use in this function, see Appendix B, "Wildcard Patterns".

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.

For information about wildcard patterns you can use in this function, see Appendix B, "Wildcard Patterns".

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. You can use `system_unix2local` to convert the UNIX file name into the format that Windows accepts. In the UNIX environment, this function does nothing, but may be called for portability.

Syntax

```
char *system_unix2local(char *path, char *lp);
```

Returns

A pointer to the local file system path string

Parameters

`char *path` is the UNIX-style 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, 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 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 C, "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 Windows./) 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 init.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 7

Creating Custom Server-Parsed HTML Tags

HTML files can contain tags that are executed on the server. For general information about server-parsed HTML tags, see the *Sun ONE Application Server Developer's Guide to Web Applications*.

In Sun ONE Application Server 7, you can define your own server-side tags. For example, you could define the tag `HELLO` to invoke a function that prints "Hello World!" You could have the following code in your `hello.shtml` file:

```
<html>
   <head>
      <title>shtml custom tag example</title>
   </head>
   <body>
      <!--#HELLO-->
   </body>
</html>
```

When the browser displays this code, each occurrence of the `HELLO` tag calls the function.

The steps for defining a customized server-parsed tag are:

1. Define the Functions that Implement the Tag.

   You must define the tag execution function. You must also define other functions that are called on tag loading and unloading and on page loading and unloading.

2. Write an Initialization Function.

   Write an initialization function that registers the tag using the `shtml_add_tag` function.

3. Load the New Tag into the Server.

Define the Functions that Implement the Tag

Define the functions that implement the tags in C, using NSAPI.

- Include the header `shtml_public.h`, which is in the directory *install_dir*/`include/shtml`.
- Link against the `shtml` shared library. On Windows, `shtml.dll` is in *install_dir*/`bin`. On UNIX platforms, `libshtml.so` or `.sl` is in *install_dir*/`lib`.

`ShtmlTagExecuteFunc` is the actual tag handler. It gets called with the usual NSAPI *pblock*, *Session*, and *Request* variables. In addition, it also gets passed the `TagUserData` created from the result of executing the tag loading and page loading functions (if defined) for that tag.

The signature for the tag execution function is:

```
typedef int (*ShtmlTagExecuteFunc)(pblock*, Session*, Request*,
TagUserData, TagUserData);
```

Write the body of the tag execution function to generate the output to replace the tag in the `.shtml` page. Do this in the usual NSAPI way, using the `net_write` NSAPI function, which writes a specified number of bytes to a specified socket from a specified buffer.

For more information about writing NSAPI plugins, see Chapter 4, "Creating Custom SAFs".

For more information about `net_write` and other NSAPI functions, see Chapter 6, "NSAPI Function Reference".

The tag execution function must return an `int` that indicates whether the server should proceed to the next instruction in `obj.conf` or not, which is one of:

- `REQ_PROCEED` -- the execution was successful.
- `REQ_NOACTION` -- nothing happened.
- `REQ_ABORTED` -- an error occurred.
- `REQ_EXIT` -- the connection was lost.

The other functions you must define for your tag are:

- `ShtmlTagInstanceLoad`

  This is called when a page containing the tag is parsed. It is not called if the page is retrieved from the browser's cache. It basically serves as a constructor, the result of which is cached and is passed into `ShtmlTagExecuteFunc` whenever the execution function is called.

- `ShtmlTagInstanceUnload`

  This is basically a destructor for cleaning up whatever was created in the `ShtmlTagInstanceLoad` function. It gets passed the result that was originally returned from the `ShtmlTagInstanceLoad` function.

- `ShtmlTagPageLoadFunc`

  This is called when a page containing the tag is executed, regardless of whether the page is still in the browser's cache or not. This provides a way to make information persistent between occurrences of the same tag on the same page.

- `ShtmlTagPageUnLoadFn`

  This is called after a page containing the tag has executed. It provides a way to clean up any allocations done in a `ShtmlTagPageLoadFunc` and hence gets passed the result returned from the `ShtmlTagPageLoadFunc`.

The signatures for these functions are:

```
#define TagUserData void*
typedef TagUserData (*ShtmlTagInstanceLoad)(
   const char* tag, pblock*, const char*, size_t);
typedef void (*ShtmlTagInstanceUnload)(TagUserData);
typedef int (*ShtmlTagExecuteFunc)(
   pblock*, Session*, Request*, TagUserData, TagUserData);
typedef TagUserData (*ShtmlTagPageLoadFunc)(
   pblock* pb, Session*, Request*);
typedef void (*ShtmlTagPageUnLoadFunc)(TagUserData);
```

Here is the code that implements the `HELLO` tag:

```
/*
 * mytag.c: NSAPI functions to implement #HELLO SSI calls
 *
 *
 */

#include "nsapi.h"
#include "shtml/shtml_public.h"

/* FUNCTION : mytag_con
 *
 * DESCRIPTION: ShtmlTagInstanceLoad function
 */
#ifdef __cplusplus
extern "C"
#endif
TagUserData
```

```
mytag_con(const char* tag, pblock* pb, const char* c1, size_t t1)
{
    return NULL;
}

/* FUNCTION : mytag_des
 *
 * DESCRIPTION: ShtmlTagInstanceUnload
 */
#ifdef __cplusplus
extern "C"
#endif
void
mytag_des(TagUserData v1)
{

}

/* FUNCTION : mytag_load
 *
 * DESCRIPTION: ShtmlTagPageLoadFunc
 */
#ifdef __cplusplus
extern "C"
#endif
TagUserData
mytag_load(pblock *pb, Session *sn, Request *rq)
{
    return NULL;
}

/* FUNCTION : mytag_unload
 *
 * DESCRIPTION: ShtmlTagPageUnloadFunc
 */
#
#ifdef __cplusplus
extern "C"
#endif
void
mytag_unload(TagUserData v2)
{

}

/* FUNCTION : mytag
 *
```

```
 * DESCRIPTION: ShtmlTagExecuteFunc
 */
#ifdef __cplusplus
extern "C"
#endif
int
mytag(pblock* pb, Session* sn, Request* rq, TagUserData t1,
TagUserData t2)
{
   char* buf;
   int length;
   char* client;
   buf = (char *) MALLOC(100*sizeof(char));
   length = util_sprintf(buf, "<h1>Hello World! </h1>", client);
   if (net_write(sn->csd, buf, length) == IO_ERROR)
   {
      FREE(buf);
      return REQ_ABORTED;
   }
   FREE(buf);
   return REQ_PROCEED;
}

/* FUNCTION : mytag_init
*
* DESCRIPTION: initialization function, calls shtml_add_tag() to
* load new tag
*/
#
#ifdef __cplusplus
extern "C"
#endif
int
mytag_init(pblock* pb, Session* sn, Request* rq)
{
   int retVal = 0;
// NOTE: ALL arguments are required in the shtml_add_tag() function
   retVal = shtml_add_tag("HELLO", mytag_con, mytag_des, mytag,
mytag_load, mytag_unload);

   return retVal;
}
/* end mytag.c */
```

Write an Initialization Function

In the initialization function for the shared library that defines the new tag, register the tag using the function `shtml_add_tag`. The signature is:

```
NSAPI_PUBLIC int shtml_add_tag (
   const char* tag,
   ShtmlTagInstanceLoad ctor,
   ShtmlTagInstanceUnload dtor,
   ShtmlTagExecuteFunc execFn,
   ShtmlTagPageLoadFunc pageLoadFn,
   ShtmlTagPageUnLoadFunc pageUnLoadFn);
```

Any of these arguments can return `NULL` except for the `tag` and `execFn`.

Load the New Tag into the Server

After creating the shared library that defines the new tag, you load the library into the Sun ONE Application Server in the usual way for NSAPI plugins. That is, add the following directives to the configuration file `init.conf`:

1. Add an `Init` directive whose `fn` parameter is `load-modules` and whose `shlib` parameter is the shared library to load. For example, if you compiled your tag into the shared object *install_dir*/`hello.so`, it would be:

   ```
   Init funcs="mytag,mytag_init" shlib="install_dir/hello.so"
   fn="load-modules"
   ```

2. Add another `Init` directive whose `fn` parameter is the initialization function in the shared library that uses `shtml_add_tag` to register the tag. For example:

   ```
   Init fn="mytag_init"
   ```

Appendix A

Data Structure Reference

NSAPI uses many data structures which are defined in the `nsapi.h` header file, which is in the directory *install_dir*/`include`.

The NSAPI functions described in Chapter 6, "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 268.

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 internal data structures in nsapi_pvt.h, and therefore that most existing NSAPI plugins will be both binary and source compatible with Sun ONE Application Server 7.

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 6, "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

Wildcard Patterns

This appendix describes the format of wildcard patterns used by the Sun ONE Application Server.

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

The following table shows wildcard patterns. The left column lists patterns, and the right column lists uses of the patterns.

Table B-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.
`[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

The following table shows wildcard examples. The left column lists patterns, and the right column lists results of the patterns.

Table B-2 Wildcard examples

<table>
<tr><th>Pattern</th><th>Result</th></tr>
<tr><td>*.sun.com</td><td>Matches any string ending with the characters .sun.com.</td></tr>
<tr><td>(quark|energy).sun.com</td><td>Matches either quark.sun.com or energy.sun.com.</td></tr>
<tr><td>198.93.9[23].???</td><td>Matches a numeric string starting with either 198.93.92 or 198.93.93 and ending with any 3 characters.</td></tr>
<tr><td>*.*</td><td>Matches any string with a period in it.</td></tr>
<tr><td>*~sun-*</td><td>Matches any string except those starting with sun-.</td></tr>
<tr><td>*.sun.com~quark.sun.com</td><td>Matches any host from domain sun.com except for a single host quark.sun.com.</td></tr>
<tr><td>*.sun.com~(quark|energy|neutrino).sun.com</td><td>Matches any host from domain .sun.com except for hosts quark.sun.com, energy.sun.com, and neutrino.sun.com.</td></tr>
<tr><td>*.com~*.sun.com</td><td>Matches any host from domain .com except for hosts from subdomain sun.com.</td></tr>
<tr><td>type=*~magnus-internal/*</td><td>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.</td></tr>
</table>

Appendix C

Time Formats

This appendix describes the format strings used for dates and times. These formats are used by some built-in SAFs such as `append-trailer`, and by server-parsed HTML (`parse-html`).

The following table describes the format strings for dates and times. The left column lists time format symbols, and the right column explains the meanings of the symbols.

Table C-1 Time formats

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"`

Table C-1 Time formats

Symbol	Meaning
%D	Date `"%m/%d/%y"`
%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 D

Dynamic Results Caching Functions

The functions described in this appendix allow you to write a results caching plugin for Sun ONE Application Server. A results caching plugin, which is a `Service` SAF, caches data, a page, or part of a page in the application server address space, which the application 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 application 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 application 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:

install_dir`/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;
}
```

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 an application 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

Sun ONE Application Server 7 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.

The following table shows commonly used request headers. The left column lists request headers, and the right column lists descriptions of those headers.

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.
`User-agent`	The name and version of the browser software.

Table E-1 Common request headers

Request header	Description
`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.

The following table shows commonly used HTTP status codes. The left column lists status codes, and the right column lists descriptions of those codes.

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 relay to the application 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 response 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 cannot handle very large amounts of data, so they return this code.
414	The URI is longer than the maximum the application 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 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. For more information about quality of service, see the *Sun ONE Application Server Performance Tuning, Sizing, and Scaling Guide*.

Response Headers

The response headers contain information about the server and the response data.

The following table shows commonly used response headers. The left column lists response headers, and the right column lists descriptions of those headers.

.

Table E-3 Common response headers

Response header	Description
`Server`	The name and version of the application 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 plugins can use them without any change.

The buffered streams layer adds following features to the Sun ONE Application 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 application server is expected to handle such low-level protocol issues.

Output buffering has been built in to the NSAPI functions that transmit data, such as `net_write` (see Chapter 6, "NSAPI Function Reference"). You can specify the following Service SAF parameters that affect stream buffering, which are described in detail in Chapter 2, "Predefined SAFs and the Request Handling Process".

- `UseOutputStreamSize`

- `flushTimer`
- `ChunkedRequestBufferSize`
- `ChunkedRequestTimeout`

The `UseOutputStreamSize`, `ChunkedRequestBufferSize`, and `ChunkedRequestTimeout` parameters also have the equivalent `init.conf` directives; see the *Sun ONE Application Server Administrator's Configuration File Reference*. The `obj.conf` parameters override the `init.conf` directives.

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

To override the default behavior when invoking an SAF that uses one of the NSAPI functions `net_read` or `netbuf_grab` (see Chapter 6, "NSAPI Function Reference"), you can specify the value of the parameter in `obj.conf`, for example:

```
Service fn="my-service-saf" type=perf UseOutputStreamSize=8192
```

Alphabetical List of Pre-defined SAFs

A

B

C

D

E

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

Alphabetical List of NSAPI Functions and Macros

C

D

F

L

M

N

P

R

S

U

V

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W