HTTPD for MVS 3.8j
User's Guide

Document Number ML07-0001-1

October 2026

Mike Großmann

First Edition (October 2026)

This edition applies to Version 4 Release 5 of HTTPD, the HTTP server for MVS 3.8j (httpd 4.5.1), and to all subsequent releases and modifications until otherwise indicated in new editions.

The console sessions that show a start of the server or its version were captured on an MVS/CE system, a throwaway container, and on a TK5 system, from a build of the 4.5.1 source made before the release was published. The build stamp in them is shown as `nnnnnnn`. A start of the published package of 4.5.1 was not captured for this edition. Two sessions come from builds of the 4.5.0 and 4.5.1 source that show the version `4.5.1-DEV` or `4.5.0-DEV`: the start on a system without a user table in the chapter on security and the banner in the chapter on removing HTTPD. The other console and client sessions may come from earlier builds of 4.5.0 and 4.5.1. The sessions in the chapters on installing and upgrading HTTPD were captured on a TK5 system: the install on a system without HTTPD and the upgrade from 4.5.0 to 4.5.1. A first install on an MVS/CE system showed the same start. The sessions in the chapter on removing HTTPD were captured on release 4.4.0 on a TK5 system, except the section on a development build, which comes from builds of 4.5.0. The removal job for 4.5.0 was run on a TK5 system as well. The removal job for 4.5.1 was run on an MVS/CE system that held the released 4.5.0 and on a TK5 system that held 4.5.1. The chapters name what they measured.

Comments on this book may be addressed to the issue tracker of the mvslovers/httpd repository on GitHub.

© Copyright Mike Großmann 2026. All rights reserved.

Contents

Figures . . . . . **vii**

About This Book . . . . . **ix**
The Examples in This Book ix
How This Book Is Organized ix
Related Publications ix

Chapter 1. Introducing HTTPD **1**
What HTTPD Does 1
The Parts 1
How a Request Is Handled 2
What HTTPD Needs 2
Version and Lineage 3
Which Chapter for Which Task 3

Chapter 2. Installing HTTPD **5**
What Is in the Package 5
What You Need 7
UFSD Is Optional 7
mvsMF Is Optional 7
RAKF Is Required for Logins 8
Authority 8
Sending the Two XMIT Files 9
Stopping a Running HTTPD 10
Allocating the Product Data Sets 10
Editing and Submitting the Install Job 12
Reading the Result 13
Verifying the Install 17
The Procedure and the Parmlib Member 18
Defining the Users and Profiles in RAKF 20
The User of the Started Task 20
The Profiles 21
The Document Root Disk 23
Starting HTTPD 24
Restarting 25
What HTTPD Serves After the Install 25
Problems at Installation 26

Chapter 3. A First Request **27**
What the Console Shows 27
Choosing a Client 28
Getting the Welcome Page 28
Reading the Response 29
A File That Is Not There 30
Two Requests on One Connection 30
What Can Go Wrong 31

Chapter 4. Configuring HTTPD **33**
Which Member HTTPD Reads 33

The Format of the Member . . . 34
A Minimal Member . . . 34
A Production Member . . . 34
Reading Back What HTTPD Parsed . . . 35
Changing the Port . . . 35
Worker Threads . . . 36
Keep-Alive and Client Time Limits . . . 37
The Request Count of a Connection . . . 37
The Diagnostic of a Client Time-out . . . 38
Sessions and the Client Time Limit . . . 38
The Document Root, the Code Page, the Time Zone and the Trace . . . 39
The Document Root . . . 39
The Code Page . . . 40
The Time Zone . . . 40
The Trace . . . 40
The Request Trace . . . 40
Recording SMF Records . . . 42
What a Bad Line or a Bad Value Does . . . 43
A Line Without an Equals Sign . . . 43
An Unknown Keyword . . . 43
A Bad Value . . . 43
Refusals . . . 44
Retired Keywords . . . 44

Chapter 5. Routes and Server Modules . . . 47
The Two Kinds of Route . . . 47
MOD= Routes . . . 47
LOC= Routes . . . 48
The Order of the Routes and the First Match . . . 48
What a Pattern Matches . . . 48
The First Line Wins . . . 49
The Fixed Paths . . . 50
The Standard Module: mvsMF . . . 50
Where the Module Must Be . . . 51
Modules Held in Storage: RESIDENT . . . 51
Context Blocks for Modules . . . 52
A Route to a Missing Module . . . 52
Warnings and Refusals While HTTPD Reads the Routes . . . 53

Chapter 6. Authentication and Security . . . 55
How a Request Is Checked . . . 55
AUTH=: What a Route Asks For . . . 55
NONE . . . 56
BASIC . . . 56
FORM . . . 57
TOKEN . . . 58
When to Use TOKEN . . . 60
The Realm . . . 60
How Long a Login Lasts . . . 60
A Changed RACF Password and the Cached Login . . . 61
The Identity of the Server . . . 61
A Security System That Accepts Any User ID . . . 62
How HTTPD Reacts . . . 63
How to Recognise It . . . 64

What to Do . . . 64
Two More Rules . . . 64
RES=: Checking a Profile . . . 65
Refusing a Route Whose Profile Is Missing . . . 66
What Needs No Login . . . 67
Console Messages for Logins . . . 67
What HTTPD Does Not Do . . . 68

Chapter 7. Serving Files . . . 69
Where the Files Are . . . 69
Building a Web Root Image . . . 69
Mounting the Image . . . 70
Changing the Document Root . . . 71
Directory Requests . . . 71
Content Types . . . 72
Text, Binary and the Code Page . . . 73
Server Side Includes . . . 74
Paths . . . 74
What Is Not Supported . . . 74
Static Locations . . . 77
When UFSD Is Not There . . . 78
What Can Go Wrong . . . 78

Chapter 8. Operating HTTPD . . . 79
Starting . . . 79
A healthy start . . . 79
A start that is refused . . . 81
A missing Parmlib member is not a refusal . . . 82
Stopping . . . 82
Cancelling . . . 83
Restarting . . . 84
The Operator Commands . . . 84
Help . . . 85
The configuration, the port, the version and the time . . . 85
Who is logged in . . . 87
The threads . . . 87
Storage . . . 88
The statistics . . . 88
Changing the number of workers and the SMF level . . . 88
Commands that are not accepted . . . 89
The Display Modules over HTTP . . . 90
Setting up the routes . . . 90
What authentication they need . . . 90
HTTPDM, a dump of storage . . . 91
HTTPDSRV, the control blocks . . . 92
HTTPDMTT, the Master Trace Table . . . 93
Worker Abends . . . 94
Recording with SMF . . . 94
Things to Know . . . 95
Watching a Long-Running Server . . . 95

Chapter 9. Writing a Server Module . . . 97
What a Server Module Is . . . 97
The Smallest Complete Module . . . 97
The Entry Point . . . 99

Reading the Request 99
Sending the Response 100
The End of the Headers 101
Storage 103
Building 103
Deploying and Routing 105
The Library 105
The Route 105
The Restart 105
What the Procedure Must Have 106
Testing with curl 106
When a Module Abends 107
Long-Running Handlers 107
HTTPD Levels and the Function Vector 108
Things to Know 109

Chapter 10. Upgrading HTTPD 111
One Release at a Time 111
Before You Start 112
Upgrading by One Release 114
Stop the Server 114
Send the XMIT Files 114
Edit and Submit the Install Job 115
Reading the Result 115
Verifying the Upgrade 120
What the SMP Upgrade Deletes 122
Upgrading from HTTPD 4.0.x 122
Upgrading from HTTPD 3.x 123
After the Upgrade: Review Your Own Files 124
Problems at Upgrade 125

Chapter 11. Removing HTTPD 127
What HTTPD Put on the System 127
Step 1: Stop the Server 128
Step 2: Cut the FMIDs out of the SMP Inventory 128
Reading the return codes 131
Step 3: Read the LIST 131
Step 4: Scratch the Libraries 133
Step 5: The Parts that SMP Never Owned 134
Step 6: Check That It Is Gone 135
Running the Job Twice 136
What Changes for HTTPD 4.5.0 136
What Changes for HTTPD 4.5.1 136
A Development Build with a Later FMID 137

Chapter 12. Troubleshooting 141
Where to Look First 141
A Start That Ends 142
The Port Is Busy 142
A Route Without a Valid Policy 143
The Procedure Has the Wrong DD Names 144
Authorization Fails 145
The Start Goes On, but Not as Intended 145
The Member Is Not There 146
Lines the Server Skips 147

Sessions That Are Too Short . . . 148
The Client Cannot Connect . . . 149
The Status Is Not 200 . . . 149
401, the Route Needs a Login . . . 150
302, the Login Form . . . 151
403, the Resource Is Denied . . . 152
404, There Is No File . . . 153
405 and 501, the Method . . . 153
Logins . . . 154
The Server Has Not Its Own Identity . . . 154
Every Login Is Refused, HTTPD042W . . . 154
A Changed Password Does Not Take Effect . . . 154
A Module Does Not Run . . . 155
A Module Abends with S0C4 . . . 156
A Response Takes Five Seconds . . . 156
A Request Is Slow . . . 157
Static Files Are Not Served . . . 158
Requests Wait or Workers Abend . . . 159
Reading the Master Trace Table . . . 159
What to Collect . . . 160

Appendix A. Glossary . . . 161

Index . . . 165

Figures

Tables

1. Chapters and tasks . . . 3
2. The files of the installation package . . . 6
3. The load modules . . . 6
4. The members of the sample library . . . 6
5. The data sets of the installation . . . 6
6. The steps of the install job . . . 12
7. The DD statements of the procedure . . . 19
8. RAKF 1.2.x and 2.0.0 . . . 20
9. Problems of the installation . . . 26
10. The lines of D C . . . 35
11. Keywords for time and for connections . . . 37
12. The SMF levels . . . 42
13. The reactions to a bad value . . . 43
14. The retired keywords . . . 44
15. The route lines . . . 47
16. The two kinds of pattern . . . 48
17. The messages for resident modules . . . 51
18. The values of AUTH= . . . 55
19. The limits of a login . . . 60
20. Signs of a RAKF without tables . . . 64
21. The answers of a route whose profile is missing . . . 66
22. The messages for logins . . . 67
23. Some content types . . . 72
24. The messages of a start, in order . . . 80
25. The commands for the operator . . . 84
26. The lines of D C . . . 86
27. HELLO, line by line . . . 98
28. The request variables of most use . . . 99
29. The storage rules of a module . . . 103
30. The keys of the project file . . . 104
31. The tests of HELLO . . . 106
32. The texts of a failed module . . . 107
33. The results of http_new_session . . . 108
34. Releases, FMIDs and what each one deletes . . . 111
35. Features of HTTPD 3.x that are not in 4.x . . . 124
36. Problems at upgrade . . . 125
37. What release 4.4.0 installed . . . 127
38. From the symptom to the section . . . 141
39. The messages that end the start . . . 142
40. The two forms of HTTPD002I . . . 145
41. Messages that let the start go on . . . 145
42. The status codes that point to a problem . . . 149
43. The reasons for a failed module . . . 155
44. Where the time of a request goes . . . 158

About This Book

This book shows how to install, configure and run HTTPD (httpd), the HTTP server for MVS 3.8j: how to put it on a system, how to describe routes and server modules in the configuration member, how HTTPD authenticates a client with the security system, how it serves files, how an operator starts, queries and stops it, how to write a server module of one's own, and how to upgrade and remove it. Every configuration keyword, console command, interface function, record layout and message in full is the subject of the companion volume, the *HTTPD Reference*.

The book describes HTTPD 4.5, as it comes with the release. HTTPREXX and HTTPLUA, the server modules for REXX Server Pages and for Lua, have their own books.

The Examples in This Book

The console screens and the client sessions in this book are output that was recorded from a running HTTPD. None of them was typed by hand. Before they were printed, some values were replaced: passwords by asterisks, session tokens by **`<token>`**, the digits of job and task numbers by **`n`**, the build stamp after the version by **`nnnnnnn`**, the names and addresses of the test systems by **`localhost`**, **`mvs.example.org`** and the address **`192.0.2.17`** of the documentation range, and the home directory of the user by **`$HOME`**. The addresses in the examples are therefore not real addresses.

How This Book Is Organized

Chapter 1	"Introducing HTTPD".
Chapter 2	"Installing HTTPD".
Chapter 3	"A First Request".
Chapter 4	"Configuring HTTPD".
Chapter 5	"Routes and Server Modules".
Chapter 6	"Authentication and Security".
Chapter 7	"Serving Files".
Chapter 8	"Operating HTTPD".
Chapter 9	"Writing a Server Module".
Chapter 10	"Upgrading HTTPD".
Chapter 11	"Removing HTTPD".
Chapter 12	"Troubleshooting".
Appendix A	"Glossary".

Related Publications

ML07-0002	*HTTPD Reference*: the Parmlib keywords, the route syntax, the console commands, the started task, the HTTP behaviour and limits, the server module interface, the display modules, the SMF records and the messages of HTTPD.

ML05-0001	*UFSD User's Guide*: the file system server that holds the document tree HTTPD serves.
ML02-0001	*MBT User's Guide*: the build tool that builds HTTPD and packages its releases.

Chapter 1. Introducing HTTPD

HTTPD is an HTTP/1.1 server for MVS 3.8j. It runs as a started task and listens on a TCP port. This chapter says what HTTPD does, what it is made of, how it handles a request, what it needs on the system, and which chapter of this book to read for which task.

What HTTPD Does

HTTPD has two jobs.

- **It hosts server modules.** A server module is a load module that HTTPD calls for the requests to a path. The main module is mvsMF, which gives MVS a REST interface in the style of z/OSMF. Clients such as scripts, command line tools and web front ends use it to work with data sets, jobs and the console. Chapter 5, “Routes and Server Modules” on page 47 explains how a path is tied to a module, and Chapter 9, “Writing a Server Module” on page 97 explains how to write one.
- **It serves static files.** HTML pages, style sheets, images and other files come from a file system that UFSD keeps (Chapter 7, “Serving Files” on page 69). A request that no module owns is answered from there.

HTTPD answers a request in the version of HTTP that the client used, and it keeps a connection open for more requests when the client allows it. The check of a user against the security product and the SMF record of a request are part of the server.

HTTPD is not a general purpose web server. Some things it does not do:

- It speaks HTTP over plain TCP. It has no TLS support. Chapter 6, “Authentication and Security” on page 55 says what that means for a login.
- It does not start a program for each request. A server module is a load module that HTTPD calls in its own task.
- It does not take files. A POST, PUT or DELETE request for a static file is refused (“What Is Not Supported” on page 74).

The Parts

A running HTTPD is made of these parts:

Started task
: The program HTTPD runs as a started task from a procedure. The procedure names the load library, the Parmlib member and the identity under which HTTPD runs. You start it with the operator command **`S HTTPD`** and stop it with **`P HTTPD`** (Chapter 8, “Operating HTTPD” on page 79). The procedure is described under “The Started Task and the Package” in the *HTTPD Reference*.

Parmlib member
: The configuration. The **`HTTPPRM`** statement of the procedure points to a member of a Parmlib data set. The member holds lines of the form **`KEYWORD=VALUE`**: the port, the number of worker threads, the document root and the routes. Without a member HTTPD starts with its defaults and no routes (Chapter 4, “Configuring HTTPD” on page 33).

Routes
: A route ties a path to something that answers it. A **`MOD=`** line names a server module, and a **`LOC=`** line names a path that is served from the files. Each route

	says whether a client has to log in. The first route that matches the path is the one that is used, and a path that no route matches is public and is served from the files (Chapter 5, “Routes and Server Modules” on page 47).
Server modules	Programs that HTTPD loads when a request for their route arrives. A module is only active when a route names it. The sample Parmlib member names mvsMF.
Web root	The files that HTTPD serves. They live in a UFS file system that UFSD keeps in a data set, below the path of the **DOCROOT** keyword. HTTPD reaches them through the UFSD started task (Chapter 7, “Serving Files” on page 69).
RAKF	The security product of the system. HTTPD asks it to check a user ID and a password, and to check whether a user may use a resource (Chapter 6, “Authentication and Security” on page 55). RAKF also provides the SVC through which HTTPD can obtain its authorization (see “What HTTPD Needs”).

How a Request Is Handled

A request passes the same steps, whatever it asks for:

1. The socket thread of HTTPD waits for connections. When a client connects, the connection goes to a worker thread. The number of workers lies between **MINTASK** (3 by default) and **MAXTASK** (9 by default).
2. The worker reads the request line and the header lines from the client. A request that is too long or malformed is refused here.
3. The worker parses the request: the method, the path, the query string and, for a POST, the body.
4. The worker works out who the client is from the credential that came with the request (a session cookie, a bearer token or a user ID and a password), and looks for the route that matches the path.
5. The route decides whether the request may go on. A route that needs a login answers a client that has not logged in with a **401** response. A client that has logged in but may not use the route gets a **403**. The paths **/login** and **/logout** belong to HTTPD itself and are answered before this check.
6. If the route names a server module, HTTPD calls the module, and the module writes the response. Otherwise HTTPD looks for a file in the web root and sends it.
7. HTTPD writes an SMF record for the request if the **SMF** keyword asks for one. The default is no record.
8. If the connection can carry another request, the worker resets itself and waits for the next request on the same connection. Otherwise it closes the connection.

The steps and their limits are listed under “HTTP Behaviour and Limits” in the *HTTPD Reference*. Chapter 3, “A First Request” on page 27 shows the steps at work on a request for a file.

What HTTPD Needs

The installation steps are in Chapter 2, “Installing HTTPD” on page 5. In short, a system needs this:

MVS 3.8j	The operating system that HTTPD is written for.
APF authorization	HTTPD has to be authorized when it runs. There are two ways. One is the SVC that RAKF provides (SVC 244), through which the program obtains the authorization itself. The other is an entry for the load library in the APF list. If HTTPD cannot obtain the authorization, it writes message **HTTPD012E**. Chapter 12, “Troubleshooting” on page 141 covers that case.
RAKF	Needed for logins. HTTPD checks a user ID and a password with RAKF.

UFSD	Needed for static files. HTTPD links the UFS library into its load module, so there is nothing more to install, but the UFSD started task has to run and hold a web root image. A site that uses HTTPD only for server modules can set **UFS=0** (“When UFSD Is Not There” on page 78). The build of HTTPD requires UFSD 1.5.2 or later.
mvsMF	Needed only for the routes of the sample Parmlib member that name it. Without such a route HTTPD serves files only.

HTTPD runs under a user ID of its own. The procedure sets it with the **STCUSER** and **STCGRP** values, and Chapter 2, “Installing HTTPD” on page 5 tells how to define that user to RAKF.

Version and Lineage

HTTPD began as a web server by Mike Rayborn for MVS 3.8j, and the last version of that line is 3.3. HTTPD 4.0 is a rewrite by Mike Großmann with HTTP/1.1, persistent connections and a configuration in Parmlib, and this book describes the 4.5 release of that line.

The version of a running server is in message **HTTPD000I** on the console (“What the Console Shows” on page 27).

Which Chapter for Which Task

Read the chapters in order the first time. After that, use the table to find the one you need.

Table 1. Chapters and tasks

I want to ...	Chapter
Install HTTPD on a system	Chapter 2, “Installing HTTPD” on page 5
Send the first request and read the answer	Chapter 3, “A First Request” on page 27
Set the port, the worker threads, the time limits and the other keywords	Chapter 4, “Configuring HTTPD” on page 33
Tie a path to mvsMF or another module, or to a static area	Chapter 5, “Routes and Server Modules” on page 47
Make a route need a login, or limit it to some users	Chapter 6, “Authentication and Security” on page 55
Serve HTML pages and other files	Chapter 7, “Serving Files” on page 69
Start and stop HTTPD, read its messages and look inside a running server	Chapter 8, “Operating HTTPD” on page 79
Write a server module of my own	Chapter 9, “Writing a Server Module” on page 97
Move from one release to the next	Chapter 10, “Upgrading HTTPD” on page 111
Remove HTTPD from a system	Chapter 11, “Removing HTTPD” on page 127
Find out why something does not work	Chapter 12, “Troubleshooting” on page 141
Look up a word	Appendix A, “Glossary” on page 161

Every keyword, command, message and limit is listed in full in the *HTTPD Reference* (ML07-0002). This book explains how to do things and points to the Reference where a list is long.

Chapter 2. Installing HTTPD

This chapter installs HTTPD on an MVS 3.8j system (TK4-, TK5, MVS/CE or a Hercules build of your own) from the release package. The installation is done by SMP Release 4, the SMP that comes with MVS 3.8j, not SMP/E. SMP keeps a record of what was installed. So there is a defined way to replace it with the next release (Chapter 10, "Upgrading HTTPD" on page 111) and a defined way to take it out (Chapter 11, "Removing HTTPD" on page 127).

The captures in this chapter come from a first install of release 4.5.1 on a TK5 system, with one exception that the text names: the RAKF captures come from MVS/CE. The package was built from the source of 4.5.1 before the release was published. The build stamp in the start-up banner is therefore not the one of the published package ("Starting HTTPD" on page 24). Every step of the install job ended with return code 0000, **`HTTPD.LINKLIB`** and **`HTTPD.AHTTPLOD`** each held the five load modules, and the SMP record showed the FMID **`THTP451`**. The file names in your package carry the number of your release. The steps do not change from one release to the next.

Report a problem as an issue at https://github.com/mvslovers/httpd/issues. Add the two build stamps that HTTPD writes at the start (**`HTTPD000I`** and **`HTTPD005I`**, "Starting HTTPD" on page 24), the console messages and, for an installation problem, the job output. Questions go to the Discord server of the project.

The names of the data sets carry no version. **`HTTPD.LINKLIB`** is the same name in every release. The **`STEPLIB`** of the procedure and an APF entry are written once. The price is that two releases cannot be installed side by side. An upgrade replaces what is there, and there is no way back to the previous release: the install job accepts the new level, and SMP refuses a **`RESTORE`** of an accepted function.

The installation is a sequence of tasks. Each one has a goal, what you type, what comes back and what can go wrong:

1. Read what is in the package ("What Is in the Package") and what the system needs ("What You Need" on page 7).
2. Send the two XMIT files to MVS ("Sending the Two XMIT Files" on page 9).
3. Stop a running HTTPD, if there is one ("Stopping a Running HTTPD" on page 10).
4. Allocate the product data sets ("Allocating the Product Data Sets" on page 10).
5. Edit and submit the install job ("Editing and Submitting the Install Job" on page 12), read the result ("Reading the Result" on page 13) and verify the work ("Verifying the Install" on page 17).
6. Put the procedure and the Parmlib member in place ("The Procedure and the Parmlib Member" on page 18).
7. Define the user and the profiles in RAKF ("Defining the Users and Profiles in RAKF" on page 20).
8. Put the webroot disk on the system and mount it ("The Document Root Disk" on page 23).
9. Start HTTPD ("Starting HTTPD" on page 24).

If this system already runs an older HTTPD, do not use this chapter. Use Chapter 10, "Upgrading HTTPD" on page 111. The FMID of this release, **`THTP451`**, deletes **`THTP450`** and nothing else ("One Release at a Time" on page 111).

What Is in the Package

The package is the archive **`httpd-4.5.1-dist.zip`** (or **`httpd-4.5.1-dist.tar.gz`**). It has six files. In their names, 4.5.1 is the release. The webroot image has no version in its name.

Table 2. The files of the installation package

File	What it is
README.md	The installation guide as a text file.
httpd-4.5.1-load.xmit	The five load modules, in a form that TSO RECEIVE reads. A little over 800,000 bytes.
httpd-4.5.1-samplib.xmit	The sample library: the procedure of the started task, a Parmlib member and the job that allocates the webroot disk. About 22,000 bytes.
httpd-4.5.1-alloc.jcl	Allocates the two libraries that SMP installs into. Run it once.
httpd-4.5.1-inst.jcl	The install job: TSO RECEIVE of both XMIT files, then SMP RECEIVE, APPLY and ACCEPT.
httpd-webroot.img	The webroot disk: a 1 MB file system image with the welcome page. It is not part of the SMP installation ("The Document Root Disk" on page 23).

The load XMIT holds five load modules. Only the first is the server:

Table 3. The load modules

Module	What it is
HTTPD	The server. It is the only module that the started task names.
HTTPDSRV	A display module for a browser. Route **/.dsrv**.
HTTPDM	A display module for a terminal client. Route **/.dm**.
HTTPDMTT	A display module for the master trace table. Route **/.dmtt**.
ABEND0C1	A module that abends on purpose, for testing. Route **/.abend**.

Nothing uses the last four until a **MOD=** line in the Parmlib member names them, and the shipped member has those lines as comments. They are diagnostic tools and need an **AUTH=** on the route. "The Display Modules" in the *HTTPD Reference* describes them.

The sample library holds three members:

Table 4. The members of the sample library

Member	What it is
HTTPD	The procedure of the started task ("The Procedure and the Parmlib Member" on page 18).
HTTPPRM0	A Parmlib member to copy and edit.
HTTPWEBR	A job that allocates the data set for the webroot disk ("The Document Root Disk" on page 23).

The installation creates these data sets. Their names are the same in every release:

Table 5. The data sets of the installation

Data set	Role

HTTPD.LINKLIB	The load library. SMP owns it. The **STEPLIB** of the procedure names it, and it can be on the APF list. Allocated by the allocation job.
HTTPD.AHTTPLOD	The distribution library: the copy that SMP keeps for the accepted level. Allocated by the allocation job.
HTTPD.HTTPLOAD	The staging library. TSO RECEIVE creates it, SMP reads the modules from it, and the last step of the install job scratches it.
HTTPD.SAMPLIB	The sample library. TSO RECEIVE creates it. It is a plain library that SMP does not own.
HTTPD.WEBROOT.UFS	The webroot disk. The job **HTTPWEBR** allocates it. SMP does not own it, and no upgrade touches it.

What You Need

- MVS 3.8j on Hercules, with SMP 4 in working order: the procedures **SMPREC** and **SMPAPP** and the **SYS1.SMP*** data sets. They are there on TK4-, TK5 and MVS/CE as delivered.
- RAKF, for the logins ("RAKF Is Required for Logins" on page 8). This one is not optional.
- UFSD, only if you want to serve static files ("UFSD Is Optional").
- mvsMF, only if you want the REST API that the shipped Parmlib member routes to ("mvsMF Is Optional").
- A user ID that may submit jobs, change the procedure library of the started tasks and change a PARMLIB.
- A way to send a file to the host in binary ("Sending the Two XMIT Files" on page 9).
- Disk space for the four data sets of SMP and TSO RECEIVE. The allocation job asks for 5 cylinders for each of the two libraries. The staging library is about the size of the load XMIT and goes away again at the end of the install job.

UFSD Is Optional

HTTPD serves static files from the file system of UFSD, and from nowhere else. The setting **DOCROOT** names a path in that file system. Without UFSD there is nothing to serve statically.

Nothing about the installation needs UFSD. The SYSMOD declares no prerequisite. A UFSD that cannot be reached at the start is a warning and not a failure:

```
HTTPD044W UNABLE TO INITIALIZE FILE SYSTEM
```

The server starts and listens, and every **MOD=** route works. Only the requests for static files fail. Beware of one thing: **HTTPD001I** still says **READY - SERVING /www** in this case. The message names the document root and does not say that the file system is there. If you want a server without static files, put **UFS=0** in the Parmlib member. That skips the initialisation and the warning ("The HTTPPRM Keywords" in the *HTTPD Reference*).

If you want static files, UFSD must run before HTTPD, as a started task of its own. HTTPD 4.5.1 is built against UFSD 1.5.2, as 4.5.0 was. A UFSD older than 1.5.0 is untested. The *UFSD User's Guide* installs UFSD.

mvsMF Is Optional

The shipped Parmlib member has four active routes, and all of them go to the server module **MVSMF**: **/zosmf/info**, **/zosmf/services/authenticate**, **/zosmf/restconsoles/*** and **/zosmf/***.

HTTPD loads a server module by name with the LINK SVC. The module must be in the **STEPLIB** of the started task or in the link list. Install mvsMF from its own package, which has its own guide, and put

its library there. If HTTPD does not find the module, a request for one of the four paths gets an error response, and the console shows

```
HTTPD908E EXTERNAL PROGRAM MVSMF ...
```

(the rest of the line says why). If you do not use mvsMF, comment out or delete the four lines. Then no request reaches a module.

RAKF Is Required for Logins

HTTPD has no user database. A login with a user ID and a password is verified by RAKF, and the access to a protected resource is checked against RAKF profiles under the identity of the user who is logged in (Chapter 6, "Authentication and Security" on page 55). The started task itself also needs a user ID in RAKF ("Defining the Users and Profiles in RAKF" on page 20). On a system without RAKF nobody can log in, and no setting turns that off.

RAKF 1.2.x needs valid tables, and loads them at the IPL. Check the tables before the IPL ("Defining the Users and Profiles in RAKF" on page 20). If RAKF 1.2.x has no valid user table, it accepts every user ID. HTTPD 4.5.1 and later notice this at the start, write **HTTPD042W** and refuse all logins ("A Security System That Accepts Any User ID" on page 62). Install HTTPD 4.5.1 or later where RAKF 1.2.x runs.

Authority

The load module **HTTPD** is linked with AC(1), and the server needs APF authorization, because without it the LINK SVC and the security calls are not available. SMP copies the module and does not bind it again, so the attribute stays. There are two ways to get the authorization.

Through the APF list. Add **HTTPD.LINKLIB** to **SYS1.PARMLIB(IEAAPF00)**. Each line has the data set name and the volume serial, separated by a blank. The previous last line gets a trailing comma. MVS 3.8j reads the APF list at the IPL only, and there is no dynamic APF list on TK5, so the entry costs one IPL. The name of the library does not change in later releases, so it costs only that one. The start shows

```
HTTPD002I AUTHORIZED BY LIBRARY (MODULE KEY 0)
```

Through RAKF. RAKF supplies the SVC 244, and HTTPD calls it itself at the start to get authorized. You need RAKF anyway, so this needs no other setup. The start shows

```
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
```

The capture of "Starting HTTPD" on page 24 is this second case.

An APF entry on MVS 3.8j names the data set and the volume. The entry does not only permit a volume, it dictates one. The allocation job says **UNIT=SYSDA** and no volume, so it puts the library wherever **SYSDA** picks, and the next section shows a second run that put both libraries on other volumes than the first. If the volume is not the one in the entry, you get a library that exists, holds the right module and is not authorized. Pin the volume on the **LINKLIB** statement of the allocation job:

```
//LINKLIB  DD  DSN=HTTPD.LINKLIB,DISP=(NEW,CATLG,DELETE),
//             UNIT=SYSDA,VOL=SER=your-apf-volume,
```

Nothing reports the mismatch, because HTTPD recovers from it: it falls back to the SVC and comes up authorized. The only place where the two ways can be told apart is **HTTPD002I**. So **AUTHORIZED BY SVC** on a system where you added an APF entry is not a decision of the site but a symptom.

If neither way gives the authorization, the start writes

```
HTTPD012E HTTPD UNABLE TO DYNAMICALLY OBTAIN APF AUTHORIZATION
```

The first word after the message number is the name of the program. There is no **HTTPD002I**, and the step ends with a return code that is not zero. “The Started Task and the Package” in the *HTTPD Reference* has the whole story of the authorization, also for the modules that run in the task.

Sending the Two XMIT Files

Goal: the two XMIT files exist as data sets on the MVS system. Both are sequential data sets with fixed blocks of 80 bytes (**RECFM=FB**, **LRECL=80**), and both must arrive in binary. A text transfer changes the bytes, and then TSO RECEIVE cannot read the file. The data set names are your choice. The captures use **HERC01.HTTPD.LOAD.XMIT** and **HERC01.HTTPD.SAMP.XMIT**.

By FTP. If FTPD or another FTP server runs on the system, switch to binary and send both files:

```
$ ftp -v -n mvs.example.org 2121
Connected to mvs.example.org.
220 MVS 3.8j FTPD Server
ftp> user HERC01
331 Send password please.
Password:
230 HERC01 is logged on.  Working directory is "HERC01.".
Remote system type is MVS.
ftp> progress off
Progress bar off.
ftp> binary
200 Representation type is Image
ftp> put httpd-4.5.1-load.xmit 'HERC01.HTTPD.LOAD.XMIT'
local: httpd-4.5.1-load.xmit remote: 'HERC01.HTTPD.LOAD.XMIT'
229 Entering Extended Passive Mode (|||2122|)
125 Storing data set HERC01.HTTPD.LOAD.XMIT
250 Transfer completed successfully.
843520 bytes sent in 00:03 (230.03 KiB/s)
ftp> put httpd-4.5.1-samplib.xmit 'HERC01.HTTPD.SAMP.XMIT'
local: httpd-4.5.1-samplib.xmit remote: 'HERC01.HTTPD.SAMP.XMIT'
229 Entering Extended Passive Mode (|||2122|)
125 Storing data set HERC01.HTTPD.SAMP.XMIT
250 Transfer completed successfully.
22240 bytes sent in 00:00 (196.11 KiB/s)
ftp> ls HTTPD.LOAD.XMIT
229 Entering Extended Passive Mode (|||2122|)
150 Opening data connection for file list
Volume Unit    Referred Ext Used Recfm Lrecl BlkSz Dsorg Dsname
TSO001 3390 2026/10/30  1   19  FB       80  3120  PS   HTTPD.LOAD.XMIT
226 List completed successfully.
ftp> ls HTTPD.SAMP.XMIT
229 Entering Extended Passive Mode (|||2122|)
150 Opening data connection for file list
Volume Unit    Referred Ext Used Recfm Lrecl BlkSz Dsorg Dsname
TSO001 3390 2026/10/30  1    1  FB       80  3120  PS   HTTPD.SAMP.XMIT
226 List completed successfully.
ftp> quit
221 Quit command received. Goodbye.
```

Read the capture from the top. After **binary** the server answers **Representation type is Image**, which is the binary mode. Each **put** ends with **250 Transfer completed successfully**. The byte count must equal the size of the file on your disk. The listing at the end shows the result on the host: **FB**, 80, a block size of 3120, and 19 tracks for the load XMIT. If the record format or the length is something else, the transfer was not binary.

By mvsMF and zowe. If mvsMF runs, the zowe CLI can upload a file to a data set in binary mode. If you upgrade, use FTP instead: mvsMF is the server that you have to stop.

By IND$FILE. A 3270 emulator with the IND$FILE transfer can send the file in binary, without line-end conversion. Allocate the data set first as **FB** with a record length of 80, so that the label has the right record format.

Note: Only the FTP route is captured in this book. The zowe and IND$FILE routes are described here without a session.

What can go wrong. A text transfer, or a data set with another record format. The RECEIVE step of the install job then fails. Send the file again in binary, into a data set that does not yet exist.

Stopping a Running HTTPD

Goal: nothing has HTTPD's libraries open. Skip this task on a first installation.

The allocation job creates **HTTPD.LINKLIB**, which a running HTTPD has open through its **STEPLIB**. The install job writes into it. A job that cannot get the library waits. The console shows the wait, the stop and the end of the wait:

```
$HASP100 HTTPDALC ON INTRDR      HTTPD ALLOC
$HASP373 HTTPDALC STARTED - INIT  1 - CLASS A - SYS TK5R
IEF861I FOLLOWING RESERVED DATA SET NAMES UNAVAILABLE TO HTTPDALC
IEF863I DSN=HTTPD.LINKLIB
*IEF099I JOB HTTPDALC WAITING FOR DATA SETS
P HTTPD
HTTPD098I HTTPD SHUTTING DOWN
HTTPD060I SHUTDOWN SOCKET THREAD    TCB(9AD080) TASK(162FC8) STACKSIZE(32768)
HTTPD060I SHUTDOWN WORKER(174C48)   TCB(9A09C8) TASK(198FC8) STACKSIZE(65536)
HTTPD060I SHUTDOWN WORKER(174CC8)   TCB(9A0CF8) TASK(187FC8) STACKSIZE(65536)
HTTPD060I SHUTDOWN WORKER(0ED7B0)   TCB(9AD828) TASK(176FC8) STACKSIZE(65536)
HTTPD416I STATS: 0 REQUESTS, 0 ERRORS, 0 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
IEF404I HTTPD - ENDED - TIME=23.41.09
$HASP395 HTTPD    ENDED
IEF403I HTTPDALC - STARTED - TIME=23.41.09
$HASP150 HTTPD    ON PRINTER2       155 LINES
$HASP160 PRINTER2 INACTIVE - CLASS=Z
$HASP250 HTTPD    IS PURGED
IEF404I HTTPDALC - ENDED - TIME=23.41.09
$HASP395 HTTPDALC ENDED
$HASP309    INIT  1 INACTIVE ******** C=A
IKJ144I   UNDEFINED USERID(S)DUMMY
```

The first lines are the wait: **IEF861I** and **IEF863I** name the data set that the job cannot get, and **IEF099I** says the job waits. Stop HTTPD with **P HTTPD**. The job goes on as soon as the task has ended.

The stop is a normal MVS stop. The message **HTTPD098I** starts it, the threads end one by one (**HTTPD060I**), and **HTTPD416I** prints the counters of the run. **HTTPD099I** says that the shutdown is complete, and **$HASP395 HTTPD ENDED** is the line that tells you the address space is gone. Wait for it before you go on. There is no **F HTTPD,SHUTDOWN**, although the comments in the procedure still list it ("Console Commands" in the *HTTPD Reference*).

The capture is of a second run of the allocation job on a system that already had the libraries. It ends with a good-looking return code, and the next section explains why that is not good.

Allocating the Product Data Sets

Goal: the two libraries that SMP installs into exist and are cataloged. Run the allocation job **httpd-4.5.1-alloc.jcl** once, on a system that has no HTTPD yet. It allocates a program library with undefined records (**RECFM=U**, **BLKSIZE=15040**) for each of **HTTPD.LINKLIB** and **HTTPD.AHTTPLOD**. Each has 5 cylinders as primary space, 2 as secondary and 20 directory blocks. These two libraries are the data sets of this chapter that SMP owns.

The job has no **DELETE** step on purpose. After the SYSMOD is accepted, the distribution library holds the accepted copy of every module. A rerun that scratched it would leave the SMP records saying "accepted" with an empty library behind them, and nothing would tell you. The comment in the job gives the rule: if you must start over, cut the FMID out with UCLIN (Chapter 11, "Removing HTTPD" on page 127), and then delete the data sets by hand. An accepted function cannot be rejected or restored.

Edit **UNIT** and **VOL=SER** if **SYSDA** is not what you want. Pin the volume if you use the APF list ("Authority" on page 8). Submit the job. The console shows that the job started and that it ended:

```
$HASP100 HTTPDALC ON INTRDR      HTTPD ALLOC
$HASP373 HTTPDALC STARTED - INIT  1 - CLASS A - SYS TK5R
IEF403I HTTPDALC - STARTED - TIME=23.35.45
IEF404I HTTPDALC - ENDED - TIME=23.35.45
$HASP395 HTTPDALC ENDED
$HASP309    INIT  1 INACTIVE ******** C=A
IKJ144I   UNDEFINED USERID(S)DUMMY
```

The last line, **IKJ144I**, comes from the job card, which has **NOTIFY=DUMMY**. The user DUMMY does not exist. The message does not change the result of the job. The step and its return code are in the job log:

```
                                          J E S 2   J O B   L O G
23.35.45 JOB   nn  $HASP373 HTTPDALC STARTED - INIT  1 - CLASS A - SYS TK5R
23.35.45 JOB   nn  IEF403I HTTPDALC - STARTED - TIME=23.35.45
23.35.45 JOB   nn  IEFACTRT - Stepname  Procstep  Program   Retcode
23.35.45 JOB   nn  HTTPDALC   ALLOC               IEFBR14   RC= 0000
23.35.45 JOB   nn  IEF404I HTTPDALC - ENDED - TIME=23.35.45
23.35.45 JOB   nn  $HASP395 HTTPDALC ENDED
```

The step is **IEFBR14** and the return code is **0000**. The system messages of the job say where the data sets went:

```
IEF236I ALLOC. FOR HTTPDALC ALLOC
IEF237I 192  ALLOCATED TO LINKLIB
IEF237I 298  ALLOCATED TO SYS00001
IEF237I 191  ALLOCATED TO AHTTPLOD
IEF142I HTTPDALC ALLOC - STEP WAS EXECUTED - COND CODE 0000
IEF285I   HTTPD.LINKLIB                              CATALOGED     *--------0
IEF285I   VOL SER NOS= TSO003.
IEF285I   SYS1.UCAT.TK5                              KEPT          *--------0
IEF285I   VOL SER NOS= TK5001.
IEF285I   HTTPD.AHTTPLOD                             CATALOGED     *--------0
IEF285I   VOL SER NOS= TSO002.
IEF373I STEP /ALLOC   / START 26303.2335
IEF374I STEP /ALLOC   / STOP  26303.2335 CPU    0MIN 00.00SEC SRB    0MIN 00.00SEC VIRT     4K SYS   172K
IEF375I  JOB /HTTPDALC/ START 26303.2335
IEF376I  JOB /HTTPDALC/ STOP  26303.2335 CPU    0MIN 00.00SEC SRB    0MIN 00.00SEC
```

Read **IEF285I**. Both data sets are **CATALOGED**, and **VOL SER NOS=** names the volumes. Write down the volume of **HTTPD.LINKLIB** if you use the APF list.

What can go wrong: the data set exists. Run the job on a system that has the names already, and it also ends with condition code 0000. The system messages show the difference:

```
IEF236I ALLOC. FOR HTTPDALC ALLOC
IEF237I 191  ALLOCATED TO LINKLIB
IEF237I 298  ALLOCATED TO SYS00008
IEF237I 192  ALLOCATED TO AHTTPLOD
IEF142I HTTPDALC ALLOC - STEP WAS EXECUTED - COND CODE 0000
IEF287I   HTTPD.LINKLIB                              NOT CATLGD  2 *--------0
IEF287I   VOL SER NOS= TSO002.
IEF285I   SYS1.UCAT.TK5                              KEPT          *--------0
IEF285I   VOL SER NOS= TK5001.
IEF287I   HTTPD.AHTTPLOD                             NOT CATLGD  2 *--------0
IEF287I   VOL SER NOS= TSO003.
IEF373I STEP /ALLOC   / START 26303.2340
IEF374I STEP /ALLOC   / STOP  26303.2341 CPU    0MIN 00.00SEC SRB    0MIN 00.00SEC VIRT     4K SYS   172K
IEF375I  JOB /HTTPDALC/ START 26303.2340
IEF376I  JOB /HTTPDALC/ STOP  26303.2341 CPU    0MIN 00.00SEC SRB    0MIN 00.00SEC
```

IEF287I with **NOT CATLGD 2** says that the new data set was created and could not be cataloged, because the name is taken. Note the volumes: the new copies are on the other volumes than in the first run. They are not the libraries that the system uses. A return code of 0000 says that the job ran. It does not say that the work happened. Look at **IEF285I** and **IEF287I** every time, and delete the uncataloged copies by hand.

What can go wrong: a running HTTPD. The job waits (“Stopping a Running HTTPD” on page 10).

What can go wrong: SB37. The space is in cylinders on purpose. A primary in tracks reaches the 16-extent limit long before the volume is full, and the abend that follows reads like a full volume.

Editing and Submitting the Install Job

Goal: the install job runs with the names of your two XMIT data sets. Open **`httpd-4.5.1-inst.jcl`**. The only lines that you have to change are the two **`RECEIVE`** statements, in the steps **`RECV1`** and **`RECV2`**. They name the data sets that you sent. The capture is the difference between the shipped job and the edited one:

```
$ diff httpd-4.5.1-inst.jcl.orig httpd-4.5.1-inst.jcl
57c57
<   RECEIVE INDSN('CHANGE.ME.HTTPLOAD') -
---
>   RECEIVE INDSN('HERC01.HTTPD.LOAD.XMIT') -
66c66
<   RECEIVE INDSN('CHANGE.ME.SAMPLIB') -
---
>   RECEIVE INDSN('HERC01.HTTPD.SAMP.XMIT') -
```

Keep the rest. The job has eight steps:

Table 6. The steps of the install job

Step	What it does
`DELOLD`	Deletes **`HTTPD.HTTPLOAD`** and **`HTTPD.SAMPLIB`**, the two targets of the next steps. TSO RECEIVE refuses to merge into an existing data set, and without this step the job would run only once.
`RECV1`	TSO RECEIVE of the load XMIT into **`HTTPD.HTTPLOAD`**.
`RECV2`	TSO RECEIVE of the sample XMIT into **`HTTPD.SAMPLIB`**.
`RECV`	SMP RECEIVE. The SYSMOD travels inside the job and goes into the SMP inventory.
`APPLYCHK`	SMP APPLY with CHECK: a dry run that changes nothing.
`APPLY`	SMP APPLY: copies the five modules into **`HTTPD.LINKLIB`**.
`ACCEPT`	SMP ACCEPT: copies them into **`HTTPD.AHTTPLOD`** and makes this level the one a **`RESTORE`** would return to.
`CLEANUP`	Scratches the staging library **`HTTPD.HTTPLOAD`**.

The steps depend on each other through condition codes. The step **`RECV`** runs if **`RECV2`** ended with 0. **`APPLYCHK`** runs if **`RECV`** ended with 0, and **`APPLY`** runs if **`APPLYCHK`** ended with 0. So the apply does not start when the dry run found a problem. **`ACCEPT`** runs if **`APPLY`** ended with 4 or lower, and **`CLEANUP`** only if **`ACCEPT`** ended with 0.

The SYSMOD is part of the job as an inline data set, in a **`DD DATA,DLM=@@`** statement. It is plain text, so there is no third file to send and no third data set to allocate. The terminator is **`@@`** and not the usual **`/*`**, because the SYSMOD contains statements that start with **`//`** and a copy statement that ends with **`/*`**.

The steps **`APPLYCHK`**, **`APPLY`** and **`ACCEPT`** carry a note in the job: the DDs that override one of the procedure **`SMPAPP`** must come before the DDs that the step adds. In the wrong order, SMP quietly uses the data set of the procedure, and the messages still name the one you wanted. Do not reorder those statements.

Submit the job. From a workstation, FTP can do it with the file type JES:

```
$ ftp -v -n mvs.example.org 2121
Connected to mvs.example.org.
220 MVS 3.8j FTPD Server
ftp> user HERC01
331 Send password please.
Password:
```

```
230 HERC01 is logged on.  Working directory is "HERC01.".
Remote system type is MVS.
ftp> progress off
Progress bar off.
ftp> quote site filetype=jes
200 SITE command was accepted
ftp> put httpd-4.5.1-inst.jcl httpd-4.5.1-inst.jcl
local: httpd-4.5.1-inst.jcl remote: httpd-4.5.1-inst.jcl
229 Entering Extended Passive Mode (|||2122|)
125 Sending Job to JES internal reader FIXrecfm 80
250-It is known to JES as JOBnnnnn
250 Transfer completed successfully.
5914 bytes sent in 00:00 (80.83 KiB/s)
ftp> quit
221 Quit command received. Goodbye.
```

The command **`quote site filetype=jes`** switches the FTP server to job submission. The **`put`** then sends the file to the internal reader of JES, and the line **`250-It is known to JES as JOBnnnnn`** names the job. You can also submit the job from TSO, from the mvsMF job API or from a card reader. Under a minute later the job is done.

Reading the Result

Goal: know whether the install job did its work. The console shows that the job started and that it ended. It does not show the steps:

```
$HASP100 HTTPDINS ON INTRDR      HTTPD INSTALL
$HASP373 HTTPDINS STARTED - INIT  1 - CLASS A - SYS TK5R
IEF403I HTTPDINS - STARTED - TIME=23.36.30
IEF404I HTTPDINS - ENDED - TIME=23.37.19
$HASP395 HTTPDINS ENDED
$HASP309    INIT  1 INACTIVE ******** C=A
IKJ144I   UNDEFINED USERID(S)DUMMY
```

The step table is in the job log, with the time of each step. Every step ends with **`RC= 0000`**:

```
                                          J E S 2   J O B   L O G
23.36.30 JOB   nn  $HASP373 HTTPDINS STARTED - INIT  1 - CLASS A - SYS TK5R
23.36.30 JOB   nn  IEF403I HTTPDINS - STARTED - TIME=23.36.30
23.36.30 JOB   nn  IEFACTRT - Stepname  Procstep  Program   Retcode
23.36.30 JOB   nn  HTTPDINS   DELOLD              IDCAMS    RC= 0000
23.36.31 JOB   nn  HTTPDINS   RECV1               IKJEFT01  RC= 0000
23.36.32 JOB   nn  HTTPDINS   RECV2               IKJEFT01  RC= 0000
23.36.32 JOB   nn  HTTPDINS   RECV      HMASMP    HMASMP    RC= 0000
23.36.42 JOB   nn  HTTPDINS   APPLYCHK  HMASMP    HMASMP    RC= 0000
23.37.03 JOB   nn  HTTPDINS   APPLY     HMASMP    HMASMP    RC= 0000
23.37.19 JOB   nn  HTTPDINS   ACCEPT    HMASMP    HMASMP    RC= 0000
23.37.19 JOB   nn  HTTPDINS   CLEANUP             IDCAMS    RC= 0000
23.37.19 JOB   nn  IEF404I HTTPDINS - ENDED - TIME=23.37.19
23.37.19 JOB   nn  $HASP395 HTTPDINS ENDED
```

The return codes mean this:

- **0000.** The step did what it says.
- **0004.** A warning. It is normal in one place: the **`APPLY`** step of an upgrade ends with 0004 (Chapter 10, “Upgrading HTTPD” on page 111). A first installation ends with 0000 in every step.
- **0008 or higher.** The step failed. The conditions stop the later steps, so the first step with a high code is the one to read.

The messages of every step are in the job output. The steps in order:

DELOLD. The step ends with 0000, although the capture looks like a failure:

```
IDCAMS  SYSTEM SERVICES                                    TIME: 23:36:30       10/30/26     PAGE   1

  DELETE HTTPD.HTTPLOAD NONVSAM SCRATCH PURGE
IKJ56228I DATA SET HTTPD.HTTPLOAD NOT IN CATALOG OR CATALOG CAN NOT BE
IKJ56228I ACCESSED
IDC0551I **ENTRY HTTPD.HTTPLOAD NOT DELETED
IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 8
```

```
  DELETE HTTPD.SAMPLIB NONVSAM SCRATCH PURGE
IDC0550I ENTRY (A) HTTPD.SAMPLIB DELETED
IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 0

  SET MAXCC=0

IDC0002I IDCAMS PROCESSING COMPLETE. MAXIMUM CONDITION CODE WAS 0
```

The first delete says **NOT DELETED** and ends with condition code 8, because **HTTPD.HTTPLOAD** does not exist yet. The statement **SET MAXCC=0** then makes the step end with 0. This is normal and not an error. The second delete found an old **HTTPD.SAMPLIB** on the system of the capture and deleted it (**IDC0550I**). Members that you edited in an old sample library are gone with it.

RECV1 and RECV2. TSO RECEIVE unpacks an XMIT file into a data set:

```
  RECEIVE INDSN('HERC01.HTTPD.LOAD.XMIT')     DATASET('HTTPD.HTTPLOAD')
NJE38 RECEIVE v2.5.0

Receiving 'HTTPD.V4R5M1.LINKLIB' from ORIGNODE(IBMUSER)

Receive into 'HTTPD.HTTPLOAD' successful
END
```

The line **Receive into 'HTTPD.HTTPLOAD' successful** is the one to find. The name after **Receiving** is the name the XMIT had on the machine where it was made, and it is not the name of your data set. The sample library has the same shape:

```
  RECEIVE INDSN('HERC01.HTTPD.SAMP.XMIT')     DATASET('HTTPD.SAMPLIB')
NJE38 RECEIVE v2.5.0

Receiving 'HTTPD.SAMPLIB' from ORIGNODE(HTTPD)

Receive into 'HTTPD.SAMPLIB' successful
END
```

RECV. SMP reads the SYSMOD from the inline data set and stores it. The output repeats the statements and ends with a return code. In it you see the function **THTP451**, the statement **++VER(Z038) DELETE(THTP450)**, and one **++MOD** statement for each of the five modules.

```
     DATE 26.303  TIME 23:36:32      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0001

HMA4240    HMASMP EXEC PARM = 'DATE=U'
 RESETRC .
HMA2050    RESETRC PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


 RECEIVE SELECT(THTP451) .




----------++FUNCTION(THTP451) .

----------++VER(Z038) DELETE(THTP450)
----------   /* httpd 4.5.1                                          */
----------   /* Load modules are copied from the LKLIB, not re-bound.      */ .

----------++JCLIN .

----------++MOD(ABEND0C1) LKLIB(HTTPLOAD) DISTLIB(AHTTPLOD) .

----------++MOD(HTTPD) LKLIB(HTTPLOAD) DISTLIB(AHTTPLOD) .

----------++MOD(HTTPDM) LKLIB(HTTPLOAD) DISTLIB(AHTTPLOD) .

----------++MOD(HTTPDMTT) LKLIB(HTTPLOAD) DISTLIB(AHTTPLOD) .

----------++MOD(HTTPDSRV) LKLIB(HTTPLOAD) DISTLIB(AHTTPLOD) .

HMA3930    SYSMOD THTP451 SUCCESSFULLY RECEIVED


     DATE 26.303  TIME 23:36:32      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0002

                       RECEIVE  SUMMARY  REPORT
```

```
     SYSMOD   STATUS     TYPE      ------------------------

     THTP451  RECEIVED   FUNCTION
     DATE 26.303  TIME 23:36:32      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0003

HMA2050    RECEIVE PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


HMA2050    HMASMP PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
```

APPLYCHK. The dry run. Its output has the same reports as the real run and changes nothing.

APPLY. The real work. Look for the five **HMA2380** lines:

```
     DATE 26.303  TIME 23:36:42      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0001

HMA4240    HMASMP EXEC PARM = 'DATE=U'
 APPLY S(THTP451) DIS(WRITE) .
HMA4140    SMPCDS DIRECTORY SUCCESSFULLY LOADED FOR IN-STORAGE UPDATE OPERATIONS


HMA4180    INLINE JCLIN PROCESSING SUCCESSFUL FOR SYSMOD=THTP451

HMA2380    COPY SUCCESSFUL - MOD=ABEND0C1 - LMOD=ABEND0C1 - LIBRARY=LINKLIB - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPD - LMOD=HTTPD - LIBRARY=LINKLIB - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDM - LMOD=HTTPDM - LIBRARY=LINKLIB - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDMTT - LMOD=HTTPDMTT - LIBRARY=LINKLIB - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDSRV - LMOD=HTTPDSRV - LIBRARY=LINKLIB - SYSMOD=THTP451 - RETURN CODE=00
HMA2270    APPLY PROCESSING SUCCESSFULLY COMPLETED FOR SYSMOD THTP451
HMA3680    SMPCDS IN STORAGE DIRECTORY SUCCESSFULLY REWRITTEN
HMA2050    APPLY PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
     DATE 26.303  TIME 23:37:03      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0002

SYSMOD STATUS REPORT FOR APPLY PROCESSING

  NOTE:  '-' INDICATES THE REQUISITE SYSMOD CONDITION IS NOT SATISFIED
         '*' INDICATES THE NON SATISFIED REQUISITE SYSMOD CONDITION IS BYPASSED


SYSMOD   STATUS    TYPE      FMID     REQUISITE AND SUPEDBY SYSMODS

THTP450  DELETED   FUNCTION

THTP451  APPLIED   FUNCTION  THTP451
     DATE 26.303  TIME 23:37:03      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0003

ELEMENT SUMMARY REPORT FOR APPLY PROCESSING


ELEM   ELEMENT   ELEM      CURRENT  CURRENT  MAC/SRC   DISTSRC   ASSEM    LOAD MOD  ---LMOD SYSLIB---  SYSMOD   SYSMOD
TYPE   NAME      STATUS    FMID     RMID     SYSLIB    LIBRARY   NAMES                                 NAME     STATUS

MOD    ABEND0C1  APPLIED   THTP451  THTP451                                ABEND0C1  LINKLIB            THTP451  APPLIED

MOD    HTTPD     APPLIED   THTP451  THTP451                                HTTPD     LINKLIB            THTP451  APPLIED

MOD    HTTPDM    APPLIED   THTP451  THTP451                                HTTPDM    LINKLIB            THTP451  APPLIED

MOD    HTTPDMTT  APPLIED   THTP451  THTP451                                HTTPDMTT  LINKLIB            THTP451  APPLIED

MOD    HTTPDSRV  APPLIED   THTP451  THTP451                                HTTPDSRV  LINKLIB            THTP451  APPLIED
     DATE 26.303  TIME 23:37:03      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0004

DELETED FUNCTION REPORT FOR  APPLY PROCESSING

SYSMOD          DELETED THE
CAUSING THE     FOLLOWING SYSMODS
DELETION        TYPE       SYSMOD

NO SYSMODS DELETED
     DATE 26.303  TIME 23:37:03      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0005

HMA2050    HMASMP PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
```

Each **HMA2380 COPY SUCCESSFUL** line names a module, the library **LINKLIB** and the SYSMOD, and ends with **RETURN CODE=00**. The five lines are the proof that five modules were copied. The final line **HMA2050 ... HIGHEST RETURN CODE IS 00** is only a summary. The copy is done by IEBCOPY, which SMP starts. Its output lists the same members and says how many tracks and directory blocks of the library are still unused:

```
                          IEBCOPY MESSAGES AND CONTROL STATEMENTS                          PAGE 0001
                    COPY OUTDD=LINKLIB,INDD=HTTPLOAD
                    SELECT MEMBER=((ABEND0C1,,R))         THTP451
                    SELECT MEMBER=((HTTPD,,R))            THTP451
                    SELECT MEMBER=((HTTPDM,,R))           THTP451
```

```
                    SELECT MEMBER=((HTTPDMTT,,R))          THTP451
                    SELECT MEMBER=((HTTPDSRV,,R))          THTP451
IEB167I  FOLLOWING MEMBER(S)  COPIED  FROM INPUT DATA SET REFERENCED BY HTTPLOAD -
IEB154I  ABEND0C1 HAS BEEN SUCCESSFULLY  COPIED
IEB154I  HTTPD    HAS BEEN SUCCESSFULLY  COPIED
IEB154I  HTTPDM   HAS BEEN SUCCESSFULLY  COPIED
IEB154I  HTTPDMTT HAS BEEN SUCCESSFULLY  COPIED
IEB154I  HTTPDSRV HAS BEEN SUCCESSFULLY  COPIED
IEB144I  THERE ARE 0000052 UNUSED TRACKS IN OUTPUT DATA SET REFERENCED BY LINKLIB
IEB149I  THERE ARE 0000019  UNUSED DIRECTORY BLOCKS IN OUTPUT DIRECTORY
IEB147I  END OF JOB -00 WAS HIGHEST SEVERITY CODE
```

ACCEPT. The same copy, into the distribution library **HTTPD.AHTTPLOD**. The output has five **HMA2380** lines with **LIBRARY=AHTTPLOD**, and it ends with **HMA2270 ACCEPT PROCESSING SUCCESSFULLY COMPLETED**:

```
     DATE 26.303  TIME 23:37:04      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0001

HMA4240    HMASMP EXEC PARM = 'DATE=U'
 ACCEPT S(THTP451) DIS(WRITE) .
HMA4140    SMPACDS DIRECTORY SUCCESSFULLY LOADED FOR IN-STORAGE UPDATE OPERATIONS



HMA2380    COPY SUCCESSFUL - MOD=ABEND0C1 - LMOD=ABEND0C1 - LIBRARY=AHTTPLOD - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPD - LMOD=HTTPD - LIBRARY=AHTTPLOD - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDM - LMOD=HTTPDM - LIBRARY=AHTTPLOD - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDMTT - LMOD=HTTPDMTT - LIBRARY=AHTTPLOD - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDSRV - LMOD=HTTPDSRV - LIBRARY=AHTTPLOD - SYSMOD=THTP451 - RETURN CODE=00
HMA2270    ACCEPT PROCESSING SUCCESSFULLY COMPLETED FOR SYSMOD THTP451
HMA3680    SMPACDS IN STORAGE DIRECTORY SUCCESSFULLY REWRITTEN
HMA2050    ACCEPT PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
     DATE 26.303  TIME 23:37:19      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0002

SYSMOD STATUS REPORT FOR ACCEPT PROCESSING

  NOTE:  '-' INDICATES THE REQUISITE SYSMOD CONDITION IS NOT SATISFIED
         '*' INDICATES THE NON SATISFIED REQUISITE SYSMOD CONDITION IS BYPASSED


SYSMOD   STATUS    TYPE      FMID     REQUISITE AND SUPEDBY SYSMODS

THTP450  DELETED   FUNCTION

THTP451  ACCEPTED  FUNCTION  THTP451
     DATE 26.303  TIME 23:37:19      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0003

ELEMENT SUMMARY REPORT FOR ACCEPT PROCESSING


ELEM   ELEMENT   ELEM      CURRENT  CURRENT  MAC/SRC   DISTSRC   ASSEM    LOAD MOD  ---MOD DISTLIB---  SYSMOD   SYSMOD
TYPE   NAME      STATUS    FMID     RMID     DISTLIB   LIBRARY   NAMES                                 NAME     STATUS

MOD    ABEND0C1  ACCEPTED  THTP451  THTP451                                          AHTTPLOD          THTP451  ACCEPTED

MOD    HTTPD     ACCEPTED  THTP451  THTP451                                          AHTTPLOD          THTP451  ACCEPTED

MOD    HTTPDM    ACCEPTED  THTP451  THTP451                                          AHTTPLOD          THTP451  ACCEPTED

MOD    HTTPDMTT  ACCEPTED  THTP451  THTP451                                          AHTTPLOD          THTP451  ACCEPTED

MOD    HTTPDSRV  ACCEPTED  THTP451  THTP451                                          AHTTPLOD          THTP451  ACCEPTED
     DATE 26.303  TIME 23:37:19      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0004

DELETED FUNCTION REPORT FOR  ACCEPT PROCESSING

SYSMOD          DELETED THE
CAUSING THE     FOLLOWING SYSMODS
DELETION        TYPE       SYSMOD

NO SYSMODS DELETED
     DATE 26.303  TIME 23:37:19      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0005

HMA2050    HMASMP PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
```

CLEANUP. The staging library goes away:

```
IDCAMS  SYSTEM SERVICES                                        TIME: 23:37:19       10/30/26     PAGE   1

  DELETE HTTPD.HTTPLOAD NONVSAM SCRATCH PURGE
IDC0550I ENTRY (A) HTTPD.HTTPLOAD DELETED
IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 0

IDC0002I IDCAMS PROCESSING COMPLETE. MAXIMUM CONDITION CODE WAS 0
```

The system of these captures had never had **THTP450**. The reports say **THTP450 DELETED** all the same: SMP lists the FMID that the statement **++VER DELETE** names, whether or not the system holds it. When the install is done, both zones of SMP hold **THTP450** as a mark with the line **DELBY = THTP451** ("Verifying the Install"). The return codes are the same as on a system that had it.

Verifying the Install

Goal: prove that the modules are in the library and that SMP knows them. A return code says that the job ran. It does not say that the work happened.

The members of the library. List the directory of **HTTPD.LINKLIB**. In TSO, **LISTDS 'HTTPD.LINKLIB' MEMBERS** does it. The utility IEHLIST prints more. Its output for the library of the capture:

```
                            SYSTEMS SUPPORT UTILITIES---IEHLIST                          PAGE    1
 DIRECTORY INFO FOR SPECIFIED PDS ON VOL TSO003
HTTPD.LINKLIB
          MEMBERS     TTRC     VARIABLE USER DATA ---(USER DATA AND TTRC ARE IN HEX)
          ABEND0C1  0000162C  0001010000 000000C2C2 0152583060 0027B08800 00010000
          HTTPD     00030F2C  0004110000 000000C2C2 047E480558 012FD88800 00010100
          HTTPDM    000C0B2C  000C250000 000000C2C2 016DD830C8 003BF88800 00010000
          HTTPDMTT  000F0A2C  000F250000 000000C2C2 01669833A0 002D888800 00010000
          HTTPDSRV  0012072C  0012290000 000000C2C2 02442009F8 00BA888800 00010000

OF THE 00020 DIRECTORY BLOCKS ALLOCATED TO THIS PDS, 00019 ARE(IS) COMPLETELY UNUSED
```

There must be five members: **ABEND0C1**, **HTTPD**, **HTTPDM**, **HTTPDMTT** and **HTTPDSRV**. The last line says that 19 of the 20 directory blocks are completely unused. The first line names the volume where the library is.

The SMP record. Ask SMP for the SYSMOD and for one load module. The job is the procedure **SMPAPP** with these control statements:

```
//LISTCDS  EXEC SMPAPP
//SMPCNTL  DD  *
 RESETRC .
 LIST CDS SYSMOD(THTP451) .
 LIST CDS LMOD(HTTPD) .
/*
```

The output:

```
     DATE 26.303  TIME 23:37:53      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0001

HMA4240    HMASMP EXEC PARM = 'DATE=U'
 RESETRC .
HMA2050    RESETRC PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


 LIST CDS SYSMOD(THTP451) .
     DATE 26.303  TIME 23:37:53      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0002

SMPCDS   SYSMOD ENTRIES


  NAME

THTP451   TYPE            = FUNCTION
          STATUS          = REC  APP  ACC
          FMID            = THTP451
          JCLIN           = YES
          DATE/TIME REC   = 26.303  23:36:32
                    APP   = 26.303  23:36:58
                    ACC   = 26.303  23:37:14
          DELETE VER(001) = THTP450
          MOD             = ABEND0C1  HTTPD     HTTPDM    HTTPDMTT  HTTPDSRV

     DATE 26.303  TIME 23:37:53      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0003

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


 LIST CDS LMOD(HTTPD) .
     DATE 26.303  TIME 23:37:53      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0004
```

```
SMPCDS  LOAD MODULE ENTRIES


  NAME

HTTPD     LASTUPD         = THTP451  TYPE=ADD
          SYSTEM LIBRARY  = LINKLIB
          LKED ATTRIBUTES = COPY

     DATE 26.303  TIME 23:37:53      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0005

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


HMA2050    HMASMP PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
```

Read the capture for four facts. The status is **REC APP ACC**: the SYSMOD is received, applied and accepted. The line **DELETE VER(001)** names the predecessor that the SYSMOD replaces. The line **MOD =** lists the five modules. And **LASTUPD = THTP451** for the load module **HTTPD**, with **SYSTEM LIBRARY = LINKLIB**, says that the module in the load library comes from this level.

If the member list is short, or the status lacks **ACC**, the install did not do its work, whatever the return codes said.

The predecessor. The SYSMOD names **THTP450** as the FMID it deletes, so SMP keeps a mark for it, also on a system that never had it. A list of **SYSMOD(THTP450)** shows the type **FUNCTION** and the line **DELBY = THTP451**, with no status and no modules. Ask for it in the CDS with **LIST CDS SYSMOD(THTP450)** and in the ACDS with **LIST ACDS SYSMOD(THTP450)**: both zones have the mark. The mark is not a sign of an old release on the system.

The Procedure and the Parmlib Member

Goal: the procedure **HTTPD** is in a procedure library of the started tasks, and a Parmlib member exists for it. The sample library has both. Copy the procedure with IEBGENER:

```
//COPY     EXEC PGM=IEBGENER
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  DISP=SHR,DSN=HTTPD.SAMPLIB(HTTPD)
//SYSUT2   DD  DISP=SHR,DSN=SYS2.PROCLIB(HTTPD)
//SYSIN    DD  DUMMY
```

SYS2.PROCLIB is an example. Use the procedure library of the started tasks on your system. Copy **HTTPPRM0** the same way into the Parmlib. In the shipped member all keywords are comments, except the four mvsMF routes of “mvsMF Is Optional” on page 7. Edit it for your system (“The HTTPPRM Keywords” in the *HTTPD Reference* describes every keyword, and Chapter 4, “Configuring HTTPD” on page 33 the way to choose them).

The procedure starts with four symbolics:

```
//HTTPD    PROC M=HTTPPRM0,
//            D='SYS2.PARMLIB',
//            STCUSER=HTTPD,
//            STCGRP=USER
```

M The member of the Parmlib. The default is **HTTPPRM0**. Start with another member by **S HTTPD,M=HTTPPRM1**.

D The data set of the Parmlib. The default is **SYS2.PARMLIB**. Check that the data set named by D exists on your system. If you use another one, change the default in the procedure, or name the data set on every start with **S HTTPD,D='...'**.

STCUSER The user ID that the server logs on as at the start (“Defining the Users and Profiles in RAKF” on page 20).

STCGRP The group of that user. The name of the symbolic is **STCGRP**, but the keyword in the PARM is **STCGROUP**: a symbolic parameter name is at most 7 characters long.

The program statement is

```
//HTTPD    EXEC PGM=HTTPD,REGION=8M,TIME=1440,
//            PARM='STCUSER=&STCUSER STCGROUP=&STCGRP'
```

The **REGION=8M** is the stock value. A server that is to run many REXX or Lua requests at the same time needs more, and **MAXTASK** must not be raised without it: "The Started Task and the Package" in the *HTTPD Reference* gives the sizes, and "Troubleshooting" the symptom.

The PARM is separated by blanks, not by commas. A **CONFIG=** in the PARM is a leftover of an old release. It gives **HTTPD024W** and **HTTPD024I**, and the server takes its configuration from the DD **HTTPPRM** as usual.

The data definitions. The procedure ships with these DD statements:

Table 7. The DD statements of the procedure

DD	Purpose
STEPLIB	**HTTPD.LINKLIB**. The load library of the server and of its modules. The package build writes the name into the procedure.
SYSUDUMP, **SNAP**	Dump output.
HTTPDERR, **HTTPDOUT**, **HTTPDBG**, **HTTPSTAT**	Output of the server, to the spool.
HTTPDIN	Dummy.
HTTPPRM	The Parmlib member: **DSN=&D(&M)**, **DISP=SHR**, **FREE=CLOSE**.
HASPCKPT, **HASPACE1**	Comment lines. The JES2 checkpoint and the spool. Optional: see below.

Do not add the DDs **SYSPRINT**, **SYSTERM** and **SYSIN**. The server refuses to start with them: **HTTPD014E**, **HTTPD015E** and **HTTPD016E** name them. The messages **HTTPD017W**, **HTTPD018W** and **HTTPD019W** report that **HTTPDOUT**, **HTTPDERR** or **HTTPDIN** is missing, and they also stop the start. "Messages" in the *HTTPD Reference* lists them.

HASPCKPT and HASPACE1. HTTPD itself never opens them. They are for the server modules. A module runs in the task of HTTPD and has no allocations of its own, and the jobs API of mvsMF reads the JES2 checkpoint and spool in that task. The shipped procedure has both statements as comment lines, and you need them in two cases only:

- You run mvsMF 1.3.0 or earlier. It opens the two data sets by their DD names. Without the statements, a job list answers 500, and a request for one job by name and id answers 404 "job not found", a spool read included, although the job exists. The console shows **MVSMF201E UNABLE TO OPEN THE JES2 CHECKPOINT AND SPOOL DATA SETS**.
- The JES2 of your system builds its data set names with a prefix other than **SYS1** (its **$DSNPRFX** value). From mvsMF 1.4.0 on, the jobs API allocates **SYS1.HASPCKPT** and **SYS1.HASPACE** itself when the procedure has no DD, and it looks only for these names.

In both cases, remove the two comment marks. The values in the shipped lines (**UNIT=3350** with **VOL=SER=MVS000** and **VOL=SER=SPOOL1**) are site values. Compare them with the JES2 procedure of your system before the first start, and change them to what JES2 uses. With mvsMF 1.4.0 or later on a system with the prefix **SYS1**, leave the lines as they are. A DD in the procedure wins over the allocation, so active lines do no harm either. "The JES2 data sets" in the *HTTPD Reference* has the details.

Missing HTTPPRM. If the DD **HTTPPRM** does not exist, the server writes

```
HTTPD020W CANNOT OPEN DD:HTTPPRM -- USING DEFAULTS
```

and starts with built-in defaults: port 8080 and no routes. A member that opens and cannot be read to the end gives **HTTPD038E**, and the server does not start.

Defining the Users and Profiles in RAKF

Goal: RAKF knows the identity of the started task and the profiles that the shipped routes name. This task has two parts, and the first matters more.

Which RAKF do you have? TK4- and TK5 have RAKF 1.2.x. MVS/CE has RAKF 2.0.0. The two differ in what you type:

Table 8. RAKF 1.2.x and 2.0.0

	RAKF 1.2.x (TK4-, TK5)	**RAKF 2.0.0 (MVS/CE)**
Password in the table USERS	Required	Blank
Where the password is kept	In clear text, in **SYS1.SECURE.CNTL(USERS)**	Not in the table
Profiles	Edit the member by hand	**RDEFINE** and **PERMIT**

The User of the Started Task

RAKF has no table of started tasks. Every started task runs under the same account, **STC** in the group **STCGROUP**, and on the stock profiles that group holds ALTER on every data set on the system. HTTPD replaces this identity at the start. It logs on the user **STCUSER** in the group **STCGRP** with RACINIT, without a password, and writes

```
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
```

The capture of “Starting HTTPD” on page 24 shows it. If the user is not defined, the logon fails. The server writes **HTTPD004W** with the user, the group and a return code, and goes on. It then runs as the account it inherited, with far more access than you meant. A warning is all you get, and that is the danger. So a start without **HTTPD004I** is not a good start.

Add a line to the table **SYS1.SECURE.CNTL(USERS)**. The columns are fixed: the user ID in 1 to 8, the group in 10 to 17, the password in 19 to 26, the Operations flag in 28 and the Special flag in 30. The first two lines in the next examples are a ruler and are not part of the member. On TK5, with a placeholder password that you replace:

```
        1         2         3
12345678901234567890123456789
HTTPD    USER     ppppppp N
```

On MVS/CE, where the password stays blank:

```
        1         2         3
12345678901234567890123456789
HTTPD    USER              N
```

Rules for the line:

- Choose the password as you would for any account of that group. HTTPD never uses it, because it logs on without one, but the account exists and can be tried. On RAKF 1.2.x the table keeps passwords in clear text, so give READ on **SYS1.SECURE.CNTL** only to the people who administer RAKF.
- On RAKF 1.2.x a blank password is refused when the table is loaded. **S RAKFUSER** ends with **RAKFUIDS2 INPUT DATA INVALID OR OUT OF SEQ.**, then **RAKFUIDSX ** PROGRAM TERMINATED **** and an abend U0200. The echo of the card always shows ********** in the password column, empty or not, so the echo does not help. Nothing in the table is updated. When this happens at the IPL, RAKF 1.2.x has no user table and accepts every user ID. HTTPD 4.5.1 and later write **HTTPD042W** then and refuse all logins (“A Security System That Accepts Any User ID” on page 62). Look for **RAKFUIDS4 USER TABLE UPDATED** in the log after the IPL.
- Keep Operations at **N**. With **Y** the account has the operations authority, which is access to data sets without a profile.
- Keep both tables in ascending order, as RAKF requires. A table that RAKF rejects is not loaded (see the blank password above).
- Do not use the user ID **HTTPPRB** or **STC** for a person. HTTPD uses **HTTPPRB** for its own test logon at the start, and refuses a login as **STC** (“A Security System That Accepts Any User ID” on page 62).
- Keep **STCGRP** equal to the group on the **USERS** line, so that the two do not drift apart. RAKF takes the group that you request at the logon and does not check that the user is connected to it.

The user needs READ on the Parmlib data set and on whatever HTTPD opens before a client identity exists. The stock rule for data sets gives that. If you narrowed it, give the group READ on the Parmlib data set, or the start fails. The user does not need a profile for SVC 244: HTTPD returns the authorization under the account it took it with, not under **STCUSER**.

Another name or group goes into the start command: **S HTTPD,STCUSER=name,STCGRP=group**, or into the defaults of the procedure.

The Profiles

Every route of the Parmlib member that has a **RES=class:resource** needs a profile for that resource. The shipped member has one, the mvsMF console service:

```
MOD=MVSMF /zosmf/restconsoles/*          AUTH=TOKEN RES=FACILITY:MVSMF.CONSOLE
```

The profile is checked against the identity of the client on every request, so the user of the started task needs none of them. A route whose resource has no profile is not denied. RAKF treats a resource without a profile as allowed, the check of the resource does nothing, and only the check of **AUTH=** is left. HTTPD says so at the start:

```
HTTPD425W NO PROFILE FOR FACILITY:MVSMF.CONSOLE -- /zosmf/restconsoles/* NOT GATED
```

With the shipped member this is what a system without the profile prints. Until the profile exists, every user who can log in may issue operator commands through mvsMF up to 1.3.0. From mvsMF 1.4.0 on, mvsMF checks the same profile itself and refuses every user until it exists, so the warning then concerns only this route. Define it. A start without the message does not prove that the profile is right. It proves only that the message is not there, so look at the table.

From HTTPD 4.5.0 the keyword **RES_NOPROFILE=DENY** turns the permissive reading into a refusal. Once the profiles are defined and the start prints no **HTTPD425W**, add the keyword to the member. A route whose profile is forgotten or not loaded yet then answers 403, and the start prints **HTTPD426W NO PROFILE FOR class:resource -- path REFUSES ALL REQUESTS**, instead of opening the route to everyone who can log in. Leave it off while a route still relies on the permissive reading. “Refusing a Route Whose Profile Is Missing” on page 66 has the details. The shipped member has the keyword as a comment line.

On MVS/CE, the commands **RDEFINE** and **PERMIT** define a profile. Run them in a TSO batch step. The capture defines the console profile, a second one, **HTTPD.ADMIN**, which a route such as **LOC=/admin/*** could name, and a third, **HTTPD.DENIED**, that is defined and permitted to nobody:

```
RDEFINE MVSMF.CONSOLE CLASS(FACILITY) UACC(NONE)
RDEFINE: FACILITY profile defined: MVSMF.CONSOLE
RDEFINE: Run RAKFPROF to activate the changes
PERMIT MVSMF.CONSOLE CLASS(FACILITY) ID(ADMIN) ACCESS(READ)
PERMIT:  access READ    granted to FACILITY/MVSMF.CONSOLE
PERMIT: ID = ADMIN
PERMIT: Run RAKFPROF to activate the changes
RDEFINE HTTPD.ADMIN CLASS(FACILITY) UACC(NONE)
RDEFINE: FACILITY profile defined: HTTPD.ADMIN
RDEFINE: Run RAKFPROF to activate the changes
PERMIT HTTPD.ADMIN CLASS(FACILITY) ID(ADMIN) ACCESS(READ)
PERMIT:  access READ    granted to FACILITY/HTTPD.ADMIN
PERMIT: ID = ADMIN
PERMIT: Run RAKFPROF to activate the changes
RDEFINE HTTPD.DENIED CLASS(FACILITY) UACC(NONE)
RDEFINE: FACILITY profile defined: HTTPD.DENIED
RDEFINE: Run RAKFPROF to activate the changes
END
```

Each **RDEFINE** writes the universal line of a profile with **UACC(NONE)**. Each **PERMIT** adds a line that gives READ to a group or a user, here **ADMIN**. Run **RDEFINE** first. The profile is not active yet: the reply says **Run RAKFPROF to activate the changes**. Choose the group of the people who may issue operator commands with care. READ on the console profile means every operator command, and nothing narrows it per command.

On TK5 these commands do not exist, and the member **SYS1.SECURE.CNTL(PROFILES)** is edited by hand. The columns are fixed: the class in 1 to 8, the resource in 9 to 52, the group in 53 to 60 and the permission in 61 to 66. The universal line of a profile has a blank group and comes first. The lines for the same profiles as above (the first two lines are a ruler):

```
         1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890123456
FACILITYHTTPD.ADMIN                                      NONE
FACILITYHTTPD.ADMIN                                ADMIN READ
FACILITYHTTPD.DENIED                                     NONE
FACILITYMVSMF.CONSOLE                                    NONE
FACILITYMVSMF.CONSOLE                              ADMIN READ
```

Merge them into the member in ascending order. A line that is out of order stops the whole table from loading.

Reloading. The tables are read when the RAKF tasks start, so a change needs a start. The task **RAKF** reads both tables. **RAKFPROF** and **RAKFUSER** read one each, the profiles and the users. The capture shows the profile task on MVS/CE:

```
S RAKFPROF
$HASP100 RAKFPROF ON STCINRDR
RAKF0010I STC RAKFPROF STARTED USING DEFAULT STC ACCOUNT
$HASP373 RAKFPROF STARTED
IEF403I RAKFPROF - STARTED - TIME=00.40.36
RAKFPROF7  RAKF PROFILES UPDATED
$HASP395 RAKFPROF ENDED
IEF404I RAKFPROF - ENDED - TIME=00.40.36
```

RAKFPROF7 RAKF PROFILES UPDATED is the line that says the table is loaded.

What can go wrong. A profile for a group the person is not in: the request is refused with 403. The *HTTPD Reference* chapter “Routes and Authentication Policy” has the rules for the route modes, and Chapter 6, “Authentication and Security” on page 55 shows how to use them.

The Document Root Disk

Goal: UFSD has a file system mounted at the path of **DOCROOT**, and the file system holds the welcome page. Skip the task if you do not use static files, and if you already have a webroot disk that you serve. The data set **HTTPD.WEBROOT.UFS** is very likely that disk, and no upgrade touches it.

The package carries the disk as a file, **httpd-webroot.img**. It is a 1 MB file system image with 256 blocks of 4096 bytes, and it holds one page, **index.html**: the welcome page. The image travels outside SMP, because SMP has no element type for a data set with undefined records, and because site content is the one thing an install must never replace.

Allocate the data set. Submit the job **HTTPWEBR** from the sample library: submit the member from **HTTPD.SAMPLIB** itself. Its important statement is

```
//WEBROOT  DD  DSN=HTTPD.WEBROOT.UFS,DISP=(NEW,CATLG),
//             UNIT=SYSDA,SPACE=(4096,256),
//             DCB=(DSORG=PS,RECFM=U,BLKSIZE=4096)
```

The block size 4096 is not a choice. The image is formatted with it, and UFSD compares the two when it mounts. A mismatch gives **UFSD062E SUPERBLOCK VALIDATION FAILED FOR HTTPD.WEBROOT.UFS**. The space is counted in blocks of 4096 and has a primary extent only: the file system is addressed by block number inside the primary extent, so a block in a secondary extent would never be reached. The job card has no **NOTIFY** operand on purpose. That operand is replaced only by TSO SUBMIT and by mvsMF, and any other way in gets a JCL error.

Upload the image. Use IND$FILE in binary mode, into the data set that the job allocated. IND$FILE takes the record format and the block size from the label of an existing data set, and it invents some only when the data set is not there. So allocating first is what pins the block size. FTP in binary mode and the zowe CLI in binary mode are other ways in. They are not measured for this data set, so the mount is the test. A text transfer destroys the image.

Mount it. The mount belongs to UFSD and not to HTTPD. Put the statement into the Parmlib member of UFSD (**UFSDPRM0**, unless you named another):

```
MOUNT    DSN(HTTPD.WEBROOT.UFS)  PATH(/www)  MODE(RO)
```

and restart UFSD. The statement is the right one for the default **DOCROOT=/www**: it keeps the case of the path and creates the mount point. The operator form of the command, **F UFSD,MOUNT DSN=...,PATH=...**, upper-cases the path (it mounts **/WWW**) and does not create the directory. Paths are case sensitive, so the default document root finds nothing under **/WWW**. The *UFSD User's Guide* describes both forms and the command that lists the mounts.

Verify. Start HTTPD ("Starting HTTPD" on page 24) and request the root path:

```
$ curl -s -D - -o /dev/null http://localhost:8080/
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:41 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/html
X-Content-Type-Options: nosniff
Transfer-Encoding: chunked
```

The status is **200 OK** and the type is **text/html**. The body is the welcome page, which says "It works. The mainframe is online." Chapter 3, "A First Request" on page 27 reads such an answer line by line.

Replacing the disk later. Stop HTTPD and wait for **$HASP395 HTTPD ENDED**. Stop UFSD. Upload the new image into the same data set. Start UFSD, then HTTPD.

What can go wrong. **UFSD062E** at the mount: the block size is not 4096, or the file was sent in text mode. A **404** on every request: the mount point is not the document root, for example **/WWW** against **/www**. A start with **HTTPD044W**: UFSD is not running (see "UFSD Is Optional" on page 7).

Starting HTTPD

Goal: the server is up and does what you expect. The checklist before the first start:

- the load library is on the APF list, or RAKF is running ("Authority" on page 8);
- the procedure is in the procedure library, and the HASP data sets are right ("The Procedure and the Parmlib Member" on page 18);
- the Parmlib member exists and the DD points at it;
- the user and the profiles are in RAKF and the tables are reloaded ("Defining the Users and Profiles in RAKF" on page 20);
- UFSD runs and has mounted the webroot, if you use static files.

Start the task from the console:

```
S HTTPD
```

A start with another member or another Parmlib is **S HTTPD,M=HTTPPRM1** or **S HTTPD,D='MY.PARMLIB'**. A healthy start looks like this:

```
S HTTPD
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=23.38.46
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/info
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/services/authenticate
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9AD080) TASK(162FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0ED7B0)   TCB(9AD828) TASK(176FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(174CC8)   TCB(9A0CF8) TASK(187FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(174C48)   TCB(9A09C8) TASK(198FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

Read the lines in order:

HTTPD000I The start. The version and the build stamp. The capture shows **nnnnnnn** in place of the stamp (see below).

HTTPD005I The C library that HTTPD is linked with. Send the two lines with a problem report.

HTTPD002I How the task got its authorization ("Authority" on page 8).

HTTPD004I The identity of the task now: **HTTPD/USER**. Without this line the task runs as the inherited account.

RAKF0004 On TK5 (RAKF 1.2.x) and MVS/CE (RAKF 2.0.0), and only in 4.5.1 and later: the server tests the security system once, with a logon that must fail (see below and "A Security System That Accepts Any User ID" on page 62).

HTTPD036I One line for each **MOD=** route.

HTTPD054I The server listens on the port.

HTTPD061I One line for the socket thread and each worker.

HTTPD001I HTTPD is ready. The text after **SERVING** is the document root.

The member used in the capture had three mvsMF routes. The shipped member has four active routes, so a start with it shows the lines for the routes that you have.

The build stamp. The capture comes from a package that was built from the source of 4.5.1 on a developer machine before the release was published. Its stamp is replaced by **nnnnnnn** in the capture, and the capture leaves out the line **HTTPD006W BUILT FROM A MODIFIED WORKING TREE**, which a build from a modified working tree writes after **HTTPD005I**. A start of the published package was not captured for this book. The version in your own **HTTPD000I** must be 4.5.1.

The line from RAKF. Since 4.5.1 the server tests the security system once at the start. It logs on the user **HTTPPRB** with a password that cannot be right. A system that checks refuses the logon. On TK5 (RAKF 1.2.x) and MVS/CE (RAKF 2.0.0) the refusal writes **RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM: USER:HTTPPRB** to the console, between **HTTPD004I** and the first **HTTPD036I**, as the capture shows. The line is expected, and no user **HTTPPRB** needs to exist. The capture comes from TK5. A first install of 4.5.1 on a throwaway MVS/CE system showed the same line in the same place. If the server writes **HTTPD042W** instead, the security software has accepted the test logon. On RAKF 1.2.x this happens when the user table is missing or was rejected at the IPL (“A Security System That Accepts Any User ID” on page 62). The server then refuses every login until the security software runs correctly and HTTPD has been restarted. Routes that need no login still answer.

A start that does not come up ends the task with a condition code other than 0, and 8 for a refused configuration. A clean stop ends with 0. The messages in front of a refused start say why. The most common ones:

- **HTTPD037E HTTPD IS ALREADY ACTIVE ON PORT n, THIS INSTANCE ENDS**: another HTTPD already serves the port.
- **HTTPD030I**: the port is in use. The server waits and tries again. After the last try, **HTTPD030E** ends the instance. The keywords **BIND_SLEEP** and **BIND_TRIES** set the wait and the number of tries. Both default to 10 and are limited to the range 1 to 100.
- **HTTPD012E**: no APF authorization (“Authority” on page 8).
- **HTTPD411E** followed by **HTTPD420E**: a route has a mode that does not exist.
- **HTTPD048E**: the retired keyword **LOGIN** with a value that used to require a login. A **LOGIN** with **NONE** or no value gives the warning **HTTPD048W** instead. Convert the routes to **AUTH=**.

Restarting

To pick up a change of the Parmlib member, stop and start:

```
P HTTPD
S HTTPD
```

Do not start before the stop has finished. Wait for **$HASP395 HTTPD ENDED**. A start that overlaps the end of the old task finds the port taken (**HTTPD030I**). Chapter 8, “Operating HTTPD” on page 79 has the operator side: how to watch a running server, how to change a setting without a restart, and what every command prints.

What HTTPD Serves After the Install

With the shipped member and nothing else installed, HTTPD answers on the port 8080 as follows:

- A request for a path under **/zosmf/** goes to mvsMF. Two of the four routes, **/zosmf/info** and **/zosmf/services/authenticate**, have **AUTH=NONE**. The other two need a login token (**AUTH=TOKEN**). Without mvsMF installed, each of these requests gives an error response and the message **HTTPD908E**.
- Every other path is a request for a static file, from the document root. With the webroot disk mounted, a request for **/** returns the welcome page. Without UFSD, such a request fails.

- A path that no route claims is public. A route without **AUTH=** is public too. So do not add a route without a mode.

The display modules are not reachable. Nothing names them until you add a **MOD=** line. Chapter 3, "A First Request" on page 27 sends the first requests. Chapter 4, "Configuring HTTPD" on page 33 and Chapter 5, "Routes and Server Modules" on page 47 change what is served, and Chapter 6, "Authentication and Security" on page 55 says who may fetch it.

Problems at Installation

Chapter 12, "Troubleshooting" on page 141 has the general table. The ones that belong to the install:

Table 9. Problems of the installation

What you see	What it means and what to do
The allocation job waits with **IEF099I**	A running HTTPD has the library. Stop it with **P HTTPD** ("Stopping a Running HTTPD" on page 10).
IEF287I ... NOT CATLGD 2 in the allocation job	The names exist. The new copies are not the libraries in use. Delete them by hand ("Allocating the Product Data Sets" on page 10).
The RECEIVE step ends with a return code that is not zero	The XMIT file was sent in text mode, or its record format is not FB 80 ("Sending the Two XMIT Files" on page 9).
All steps end with 0 but **HTTPD.LINKLIB** has no members	A return code says that the job ran. List the members and the SMP record ("Verifying the Install" on page 17).
HTTPD012E at the start	No APF authorization and no SVC ("Authority" on page 8).
HTTPD002I AUTHORIZED BY SVC although the library is on the APF list	The library is on another volume than the APF entry ("Authority" on page 8).
No **HTTPD004I**, or **HTTPD004W** instead	The user of the task is not defined in RAKF, or the logon failed ("Defining the Users and Profiles in RAKF" on page 20).
HTTPD042W at the start, and every login gets 401	RAKF 1.2.x accepts any user ID, because its user table is missing or was rejected. Correct the tables, IPL, and restart HTTPD ("A Security System That Accepts Any User ID" on page 62).
HTTPD425W or **HTTPD426W** at the start	A route names a resource that has no profile ("Defining the Users and Profiles in RAKF" on page 20).
HTTPD044W at the start	UFSD is not running ("UFSD Is Optional" on page 7).
UFSD062E at the mount	The webroot data set has not the block size 4096, or the image was sent in text mode ("The Document Root Disk" on page 23).
HTTPD020W at the start	The DD **HTTPPRM** is missing ("The Procedure and the Parmlib Member" on page 18).
HTTPD014E, **HTTPD015E** or **HTTPD016E** at the start	The procedure has a DD that it must not have ("The Procedure and the Parmlib Member" on page 18).

To take HTTPD out of the system again, follow Chapter 11, "Removing HTTPD" on page 127.

Chapter 3. A First Request

This chapter sends a few requests to a running HTTPD with **curl** and reads what comes back: the welcome page, a plain text file, a file that is not there, and two requests on one connection. The requests ask for static files from the document root. Routes, server modules and logins come in later chapters.

You need a running HTTPD (Chapter 2, “Installing HTTPD” on page 5, Chapter 8, “Operating HTTPD” on page 79), the host name of the MVS system and the port of HTTPD. The examples run **curl** on a machine that reaches HTTPD as **localhost** on port 8080, which is the default of **PORT**. Replace **localhost:8080** with the host name and port of your system. The web root image that ships with HTTPD holds the welcome page and nothing else. The other files in the examples (**hello.txt**, **data.json** and a few more) are small test files. To repeat an example, put a file of that name into your web root (Chapter 7, “Serving Files” on page 69).

What the Console Shows

Before the first request, look at the console. A healthy start ends with message **HTTPD001I**. The lines before it show what HTTPD has set up:

```
S HTTPD
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.44.51
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/info
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/services/authenticate
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/restconsoles/*
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CD7C8) TASK(152FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(164D00)   TCB(9BBEB8) TASK(166FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(164C00)   TCB(9BBB88) TASK(177FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(164B80)   TCB(9BB858) TASK(188FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

The lines have this meaning:

HTTPD000I The start. It names the version and the build of the program.

HTTPD005I The C runtime library that HTTPD is linked with.

HTTPD002I How the started task got its authorization: by the library it was fetched from, or by an SVC (the case in the capture).

HTTPD004I The started task has given up the default identity that RAKF gives to every started task. It runs under the user ID and the group from the **STCUSER** and **STCGRP** values of the procedure (here **HTTPD** and **USER**).

RAKF0004 A line from RAKF, not from HTTPD. The server logs on once with a user ID that cannot exist, to test the security product, and RAKF refuses it. A line **HTTPD042W** in its place is not a good start (“A Security System That Accepts Any User ID” on page 62).

HTTPD036I One line for each **MOD=** route: the server module and the path it is registered for. A configuration without **MOD=** routes has none of these lines.

HTTPD054I HTTPD listens on the port. **ANY** means all interfaces.

HTTPD061I A thread has started: the socket thread first, then the workers. The default is three workers.

HTTPD001I HTTPD is ready. The text after **SERVING** is the document root.

Do not send requests before **HTTPD001I** has been written. How to start, stop and watch HTTPD is the subject of Chapter 8, “Operating HTTPD” on page 79. A message that is not in this list is described under “Messages” in the *HTTPD Reference*.

Choosing a Client

Any HTTP client can talk to HTTPD. A browser shows the page. **curl** shows more, and four forms of it are used in this book:

- **curl url** prints the body of the response and nothing else. Use it to see what a page contains.
- **curl -i url** prints the status line and the headers first, then the body. Use it for most of the sessions in this book.
- **curl -v url** prints the request that **curl** sends too, each line with the prefix **>**, and the response, each line with the prefix **<**, and some lines of its own that start with *****. Use it when it matters what the client sent, for example for the connection that is reused in “Two Requests on One Connection”.
- **curl -I url** sends a HEAD request and prints the status line and the headers, and nothing else. HTTPD answers a HEAD for a static file with the headers that a GET gets, and with no body:

```
$ curl -I http://localhost:8080/hello.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 17:11:59 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 61
```

The **Content-Length** is the size of the file. HTTPD 4.4.0 and earlier answer this request with **501 Not Implemented**. A HEAD request for a path that belongs to a server module goes to the module, and the module decides (Chapter 5, “Routes and Server Modules” on page 47).

Getting the Welcome Page

A request for **/** asks for a directory. HTTPD looks in that directory for **index.html** and then for **default.html** (see “Directory Requests” on page 71). The web root that ships with HTTPD has an **index.html**, the welcome page. Fetch it:

```
curl http://localhost:8080/
```

The command prints the HTML text of the page. The page has the title “Welcome to HTTPD”, the line “It works.” and the text “You are connected to HTTPD, an HTTP/1.1 server running on MVS 3.8j.” A browser at the same URL shows it as a page. The headers are of more interest. This command prints them and nothing else:

```
$ curl -s -D - -o /dev/null http://localhost:8080/
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:41 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/html
X-Content-Type-Options: nosniff
```

```
Transfer-Encoding: chunked
```

The option **-s** keeps **curl** quiet about its progress, **-D -** writes the response headers to the standard output, and **-o /dev/null** throws the body away.

Reading the Response

The first line is the status line: the HTTP version, the status code and a short text. HTTPD answers in the version of the request. A request in HTTP/1.1 gets **HTTP/1.1**, and a request in HTTP/1.0 gets **HTTP/1.0**. The headers follow, one on each line, up to an empty line. The body comes after it.

Every response that HTTPD writes starts with the same four headers:

Date The date and time at which HTTPD built the response, in the form of RFC 1123.

Server Always **HTTPD Server**. It does not name the version.

X-Powered-By Always **MVSLOVERS**.

Connection **keep-alive** if HTTPD takes another request on this connection, **close** if it closes the connection after this response.

A response for a static file adds more headers. The examples show them in the order in which HTTPD writes them:

Cache-Control **no-store**, sent for HTML documents. It keeps a browser from keeping a copy of the page. A text file has no such header.

Content-Type The type of the file, chosen from the end of its name (“Content Types” on page 72).

X-Content-Type-Options **nosniff**. It tells a browser to believe the **Content-Type**.

Content-Length The size of the file in bytes. A response without it uses **Transfer-Encoding: chunked** instead.

The welcome page above has **Transfer-Encoding: chunked** and no **Content-Length**. HTTPD scans an HTML document for server side includes while it sends it, so it does not know the length in advance (“Server Side Includes” on page 74). A text file has a **Content-Length**:

```
$ curl -i http://localhost:8080/hello.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 61

Hello from the HTTPD document root.
This file is plain text.
```

The file is 61 bytes long. The body follows the empty line. To see the whole conversation and not only the response, use **curl -v**:

```
$ curl -v http://localhost:8080/hello.txt
* Host localhost:8080 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:8080...
* Connected to localhost (::1) port 8080
* using HTTP/1.x
> GET /hello.txt HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.14.1
```

```
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< Date: Tue, 27 Oct 2026 05:41:42 GMT
< Server: HTTPD Server
< X-Powered-By: MVSLOVERS
< Connection: keep-alive
< Content-Type: text/plain
< X-Content-Type-Options: nosniff
< Content-Length: 61
<
{ [61 bytes data]
* Connection #0 to host localhost left intact
Hello from the HTTPD document root.
This file is plain text.
```

The lines with **>** are what **curl** sent. A request in HTTP/1.1 must have a **Host** header. HTTPD refuses one without it with a **400** response.

A File That Is Not There

A request for a path that is not in the document root is answered with **404 Not Found**:

```
$ curl -i http://localhost:8080/nothere.html
HTTP/1.1 404 Not Found
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

404 document "/nothere.html" not found
```

The body is one line of plain text. It repeats the path as HTTPD decoded it (“Paths” on page 74). The same answer comes for a directory without an index file and for a path with two dots in it. The answer does not say which of these it was. The 404 response has **Cache-Control: no-store** and a body of plain text. The status codes of HTTPD are in “HTTP Behaviour and Limits” in the *HTTPD Reference*.

Two Requests on One Connection

A client that wants two files can send both requests on one connection. Give **curl** two URLs:

```
$ curl -v http://localhost:8080/hello.txt http://localhost:8080/css/site.css
* Host localhost:8080 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:8080...
* Connected to localhost (::1) port 8080
* using HTTP/1.x
> GET /hello.txt HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.14.1
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< Date: Tue, 27 Oct 2026 05:41:53 GMT
< Server: HTTPD Server
< X-Powered-By: MVSLOVERS
< Connection: keep-alive
< Content-Type: text/plain
< X-Content-Type-Options: nosniff
< Content-Length: 61
<
{ [61 bytes data]
* Connection #0 to host localhost left intact
Hello from the HTTPD document root.
This file is plain text.
* Re-using existing http: connection with host localhost
> GET /css/site.css HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.14.1
> Accept: */*
```

```
>
* Request completely sent off
< HTTP/1.1 200 OK
< Date: Tue, 27 Oct 2026 05:41:53 GMT
< Server: HTTPD Server
< X-Powered-By: MVSLOVERS
< Connection: keep-alive
< Content-Type: text/css
< X-Content-Type-Options: nosniff
< Content-Length: 67
<
{ [67 bytes data]
* Connection #0 to host localhost left intact
body { font-family: sans-serif; margin: 2em; }
h1 { color: #036; }
```

The line "Re-using existing http: connection" shows it. The first response has **`Connection: keep-alive`**, so HTTPD waits for the next request. It waits for **`KEEPALIVE_TIMEOUT`** seconds (5 by default), and **`KEEPALIVE_MAX`** limits the number of requests on one connection (100 by default, plus the one that is answered with **`Connection: close`**). Both are described in "The HTTPPRM Keywords" in the *HTTPD Reference*. When the limit is reached, HTTPD answers with **`Connection: close`** and closes the connection after the response.

An HTTP/1.0 client gets a different answer. In HTTP/1.0 a connection ends after one response, unless the client asks for **`Connection: keep-alive`**:

```
$ curl -i --http1.0 http://localhost:8080/hello.txt
HTTP/1.0 200 OK
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 61

Hello from the HTTPD document root.
This file is plain text.
```

The status line says **`HTTP/1.0`** and the response has **`Connection: close`**. HTTPD closes the connection after the body.

A request in HTTP/1.1 can end the connection too. The client can send **`Connection: close`**, and a request that has a body (a **`Content-Length`** or a **`Transfer-Encoding`** header) is always the last one on its connection. "What Is Not Supported" on page 74 shows examples.

What Can Go Wrong

- **The connection is refused or times out.** HTTPD is not running, the port is not the value of **`PORT`**, or a firewall is in the way. Look for **`HTTPD001I`** and **`HTTPD054I`** on the console. Chapter 12, "Troubleshooting" on page 141 covers a start that fails.
- **Every request for a file gets a 404, also the one for `/`.** The document root has no files, or HTTPD cannot reach it. Chapter 7, "Serving Files" on page 69 describes the document root and the web root image.
- **A request for the path of a server module gets a 401.** The route needs a login (Chapter 6, "Authentication and Security" on page 55).
- **The page is old.** A browser shows what it kept in its cache. HTML documents carry **`Cache-Control: no-store`**, other files do not.

Chapter 4. Configuring HTTPD

HTTPD takes its whole configuration from one member of a Parmlib data set. There is no other file and no command that writes the configuration. This chapter is organised by what you want to do: select the member, write it, read back what HTTPD understood, and change the port, the worker tasks, the time limits, the document root and the recording of SMF records. It ends with what HTTPD does with a bad line. The routes (`MOD=` and `LOC=`) have their own chapter, Chapter 5, “Routes and Server Modules” on page 47, and the authentication has Chapter 6, “Authentication and Security” on page 55. Every keyword, with its default and its limits, is listed in ‘The HTTPPRM Keywords’ in the *HTTPD Reference*.

Which Member HTTPD Reads

HTTPD reads the DD name **`HTTPPRM`**. The started task procedure points the DD name at one member of a Parmlib data set:

```
//HTTPPRM  DD  DSN=&D(&M),DISP=SHR,FREE=CLOSE
```

The procedure has two symbolic parameters for this line. **`M`** is the member, with the default **`HTTPPRM0`**. **`D`** is the data set, with the default **`SYS2.PARMLIB`**. You choose another member or another data set when you start the task. The member is the only difference between two configurations.

```
S HTTPD
S HTTPD,M=HTTPPRM1
S HTTPD,D='MY.PARMLIB'
```

The procedure and its other symbols are described in ‘The Started Task and the Package’ in the *HTTPD Reference*. The installation is in Chapter 2, “Installing HTTPD” on page 5.

Note: Two servers. To run a second HTTPD on the same system, give it its own **`PORT`** and so its own member. A second server on another port starts. A second server on a port that the first one holds ends with the return code 8 (“Changing the Port” on page 35).

Three things need a word:

- **`STCUSER`** and **`STCGROUP`** come from the PARM string of the procedure, not from the member. A line with one of them in the member gives **`HTTPD029W`**. Chapter 6, “Authentication and Security” on page 55 describes the identity of the task.
- A PARM word **`CONFIG=name`** has no effect. HTTPD writes **`HTTPD024W`** and **`HTTPD024I`** and tells you to select the member with **`S HTTPD,M=`**.
- A change in the member takes effect when you stop and start HTTPD.

When the DD name is missing, HTTPD writes **`HTTPD020W`** and starts with its defaults: port 8080 and no routes. A read error gives **`HTTPD038E`** and the return code 8. The effect of the missing routes shows in this capture. The path of a module is now a request for a file, and there is no such file.

```
$ curl -i http://localhost:8080/zosmf/info
HTTP/1.1 404 Not Found
Date: Tue, 27 Oct 2026 05:47:03 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked
```

```
404 document "/zosmf/info" not found
```

The Format of the Member

The member has one setting on each line, in the form **KEYWORD=VALUE**.

- Blanks around the keyword and around the value are removed. HTTPD changes the keyword to upper case. The value keeps its case. That matters for **REALM**, for paths and for module names in routes.
- A line that starts with **#** or ***** is a comment. A comment is a whole line. HTTPD has no comment at the end of a line, so text after a value is part of the value.
- HTTPD reads at most 255 characters of a line at once. There is no continuation line.
- When a keyword stands twice, the last line wins. The exceptions are **MOD** and **LOC**, where each line adds a route.

A line without an equals sign gives **HTTPD026W**. A keyword that HTTPD does not know gives **HTTPD029W**. “What a Bad Line or a Bad Value Does” on page 43 shows both.

A Minimal Member

A server that serves files needs very little. Everything else has a default. This member names the port and the directory of the files:

```
PORT=8080
DOCROOT=/www/docs
```

With this member HTTPD listens on port 8080, serves the files below **/www/docs** and has no route. Every path is a request for a file. A route is needed for a module, or for a path that must ask for a password (Chapter 5, “Routes and Server Modules” on page 47, Chapter 6, “Authentication and Security” on page 55).

A Production Member

A member for a server that has users writes down the time limits, the realm and the recording explicitly, so that the values are in the member and not only in the defaults. Every keyword in this example is described in the sections that follow. Most values are the defaults. The routes are examples. They are explained in Chapter 5, “Routes and Server Modules” on page 47.

```
# HTTPD production configuration
PORT=8080
MINTASK=3
MAXTASK=9
CLIENT_TIMEOUT=10
KEEPALIVE_TIMEOUT=5
KEEPALIVE_MAX=100
DOCROOT=/www
CODEPAGE=CP037
REALM=MVS.EXAMPLE
SESSION_TIMEOUT=30
SESSION_MAXAGE=480
SMF=ERROR
# routes: the first match wins
MOD=MVSMF /zosmf/info                    AUTH=NONE
MOD=MVSMF /zosmf/services/authenticate   AUTH=NONE
MOD=MVSMF /zosmf/*                       AUTH=TOKEN
LOC=/pub/*                               AUTH=NONE
LOC=/private/*                           AUTH=BASIC
```

The member that the captures of this guide use has more routes. Chapter 5, “Routes and Server Modules” on page 47 shows it.

Reading Back What HTTPD Parsed

After the start, ask HTTPD what it holds. The command is **`F HTTPD,D C`**.

```
F HTTPD,D C
HTTPD100I CONS(0) "D C"
HTTPD128I PORT 8080  MINTASK 3  MAXTASK 9
HTTPD129I CLIENT_TIMEOUT 10  KEEPALIVE 5/100
HTTPD130I SESSION_TIMEOUT 30 MIN
HTTPD135I SESSION_MAXAGE 480 MIN
HTTPD131I DOCROOT /www
HTTPD134I CODEPAGE CP037
HTTPD136I REALM MVSC
HTTPD132I SMF TYPE 243 LEVEL NONE
HTTPD133I CONFIG FROM DD:HTTPPRM
HTTPD137I RESIDENT YES  MODULES HELD 0
HTTPD138I RES_NOPROFILE ALLOW
```

The lines show the values that are in effect, after all the limits and corrections. Each line has a message of its own:

Table 10. The lines of D C

Message	Shows
`HTTPD128I`	The port and the least and the greatest number of worker tasks.
`HTTPD129I`	The client time limit, then the keep-alive time and count.
`HTTPD130I`	**`SESSION_TIMEOUT`** in minutes.
`HTTPD135I`	**`SESSION_MAXAGE`** in minutes.
`HTTPD131I`	The document root.
`HTTPD134I`	The code page.
`HTTPD136I`	The realm.
`HTTPD132I`	The SMF type and level.
`HTTPD133I`	The source of the configuration.
`HTTPD137I`	Whether modules are resident, and how many are held.
`HTTPD138I`	The value of **`RES_NOPROFILE`** (“Refusing a Route Whose Profile Is Missing” on page 66).

If a line shows another value than the one you wrote, HTTPD did not read the member the way you thought. Look for a warning at the start (“What a Bad Line or a Bad Value Does” on page 43). The routes are not part of this display. The display module **`HTTPDSRV`** shows them. Chapter 8, “Operating HTTPD” on page 79 describes the console commands and the display modules.

The worker task counts that **`D C`** shows are the configured values. A change with **`SET MI`** or **`SET MA`** does not appear here (“Worker Threads” on page 36).

Changing the Port

The default port is 8080. To change it, edit the member, stop the server and start it again. There is no command that moves the listener.

```
PORT=9090
DOCROOT=/www
```

After the start, ask for the port:

```
F HTTPD,D PORTS
HTTPD100I CONS(0) "D PORTS"
HTTPD102I HTTPD IS LISTENING ON PORT 9090
```

HTTPD also writes the port at start, in the message **HTTPD054I**. Before it listens, HTTPD takes an ENQ on the port. A second HTTPD that asks for the same port ends with **HTTPD037E** and the return code 8. When the bind itself fails, HTTPD tries again: **BIND_TRIES** (default 10) times, with **BIND_SLEEP** (default 10) seconds between the tries. The messages are **HTTPD030E** and **HTTPD030I**. A failure of the socket call is **HTTPD028E**, a failure of the listen call is **HTTPD031E**. **LISTEN_QUEUE** (default 5) is the length of the queue of connections that wait for HTTPD to accept them.

Note: A wrong port does not stop the server. A port value that is not valid gives **HTTPD023E**, but HTTPD goes on and listens on 8080. The member below has **PORT=0**. The server starts, and the message **HTTPD054I** names the port that it really uses. Check **HTTPD054I** or **D PORTS** after every change of the port.

```
PORT=0
DOCROOT=/www
```

```
S HTTPD,M=PRMBADP
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.26.58
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD023E INVALID PORT VALUE (0)
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

Worker Threads

HTTPD serves each request on a worker task. The pool starts with **MINTASK** tasks (default 3) and can grow to **MAXTASK** (default 9). You see the workers at start as lines **HTTPD061I STARTING WORKER**. The captures in “What a Bad Line or a Bad Value Does” on page 43 show three.

```
MINTASK=3
MAXTASK=9
```

In the member, both keywords accept 1 to 255. A value of 0 or less, or a text that is not a number, is ignored and the earlier value stays. A value above 255 is limited to 255.

The operator can change both numbers while the server runs, without a stop:

```
F HTTPD,SET MA 12
F HTTPD,SET MI 4
```

```
F HTTPD,SET MA 12
HTTPD100I CONS(0) "SET MA 12"
HTTPD107I .....EYE CTHDMGR   ....TASK 001A9FC8  .WAITECB 40000000
HTTPD108I ....FUNC 00098F1C  ...UDATA 000DB390  STACKSZE 00010000
HTTPD109I ..WORKER 000FB6D4  ...QUEUE 000FB07C  ...STATE 00000004 WAITING
```

```
HTTPD110I .MINTASK        3  .MAXTASK      12
HTTPD111I .WORKERS        3  .DISPCNT 18
HTTPD199I ------------------------------------------------------
```

The command answers with the lines of the worker manager. The line **HTTPD110I** shows the new **MINTASK** and **MAXTASK**. A raised **MINTASK** starts the missing workers at once. After **SET MI 4**, a line **HTTPD061I STARTING WORKER** follows.

The rules of the command differ from the rules of the member:

- The console accepts 1 to 100, not 255. A larger value is set to 100. In the capture below, 99999 gives a **MAXTASK** of 100.
- The console needs a number. Without one, and with a negative number, HTTPD answers **HTTPD149I** and changes nothing.
- A **MINTASK** above the **MAXTASK** raises the **MAXTASK**. A **MAXTASK** below the **MINTASK** lowers the **MINTASK**.
- The change belongs to the running worker manager only. HTTPD does not write it to the member, and **D C** keeps showing the values of the member. At the next start the member counts again.

```
F HTTPD,SET MA 99999
HTTPD100I CONS(0) "SET MA 99999"
HTTPD107I .....EYE CTHDMGR   ....TASK 001A9FC8  .WAITECB 40000000
HTTPD108I ....FUNC 00098F1C  ...UDATA 000DB390  STACKSZE 00010000
HTTPD109I ..WORKER 000FB6D4  ...QUEUE 000FB07C  ...STATE 00000004 WAITING
HTTPD110I .MINTASK        3  .MAXTASK     100
HTTPD111I .WORKERS        3  .DISPCNT 18
HTTPD199I ------------------------------------------------------
```

The commands and their messages are described in 'Console Commands' in the *HTTPD Reference*, and their daily use in Chapter 8, "Operating HTTPD" on page 79.

Keep-Alive and Client Time Limits

Three keywords set how long HTTPD waits for a client and how often a client may reuse a connection.

Table 11. Keywords for time and for connections

Keyword	Default	Meaning
CLIENT_TIMEOUT	10	Seconds that HTTPD waits for the next line of a request. The largest value is 255.
KEEPALIVE_TIMEOUT	5	Seconds that HTTPD waits for the next request on a connection that has already carried a request.
KEEPALIVE_MAX	100	How many requests a connection carries before HTTPD closes it (one more is answered with **Connection: close**). The values are 1 to 255.

CLIENT_TIMEOUT counts for each line of the request, not for the whole request. A new connection waits for its first request line as long as **CLIENT_TIMEOUT**. **KEEPALIVE_TIMEOUT** applies from the second request on.

The Request Count of a Connection

A connection carries **KEEPALIVE_MAX** plus one request. HTTPD answers the last request of the connection with **Connection: close**. With the member

```
PORT=8080
DOCROOT=/www
```

```
KEEPALIVE_MAX=2
KEEPALIVE_TIMEOUT=5
```

the third response closes the connection:

```
$ curl -sv -o /dev/null -o /dev/null -o /dev/null http://localhost:8080/hello.txt http://localhost:8080/hello.txt http://localhost:8080/
hello.txt 2>&1 | grep -E "Connected|Re-using|Connection|^< HTTP|Closing|keep-alive|Keep-Alive|left intact"
* Connected to localhost (::1) port 8080
< HTTP/1.1 200 OK
< Connection: keep-alive
* Connection #0 to host localhost left intact
* Re-using existing http: connection with host localhost
< HTTP/1.1 200 OK
< Connection: keep-alive
* Connection #0 to host localhost left intact
* Re-using existing http: connection with host localhost
< HTTP/1.1 200 OK
< Connection: close
```

The curl command uses one connection three times. The first two responses say **`Connection: keep-alive`**, the third says **`Connection: close`**. Plan with one request more than the number that you write.

The Diagnostic of a Client Time-out

When a client is too slow, HTTPD writes a line to the console. Two keywords control it. Both default to 1 (on). **`CLIENT_TIMEOUT_MSG`** writes the line, and **`CLIENT_TIMEOUT_DUMP`** adds a dump of the characters that arrived. HTTPD writes them only when the client has sent at least one character of the line. A client that sends nothing leaves nothing on the console.

The test below sets a limit of 3 seconds, sends half of a request line and waits.

```
PORT=8080
DOCROOT=/www
CLIENT_TIMEOUT=3
CLIENT_TIMEOUT_MSG=1
CLIENT_TIMEOUT_DUMP=1
```

```
$ exec 3<>/dev/tcp/localhost/8080
$ printf 'GET /hello.tx' >&3
$ sleep 6
$ exec 3>&-
```

The console shows:

```
client 00144FC8 on socket 1 timed out after 3 seconds
Dump of 00145028 "Receive Buffer" (13 bytes)
+00000 C7C5E340 61888593 93964BA3 A7       :GET /hello.tx   :
```

The first line names the address of the client block, the socket and the seconds. It is in lower case and has no message ID. The dump shows what the server received: 13 bytes, **`GET /hello.tx`**. It tells you how far the request line came before the client stopped.

Raise **`CLIENT_TIMEOUT`** for clients on slow lines. Keep **`SESSION_TIMEOUT`** and **`SESSION_MAXAGE`** longer than it (“Sessions and the Client Time Limit”).

Note: **`CLIENT_STATS`** is a third keyword of this group. It defaults to 1. HTTPD stores the flag and the display module **`HTTPDSRV`** shows it. It changes nothing else.

Sessions and the Client Time Limit

The keywords **`SESSION_TIMEOUT`** (idle minutes, default 30) and **`SESSION_MAXAGE`** (total minutes, default 480) belong to the logins that Chapter 6, “Authentication and Security” on page 55 describes. This chapter shows only the check that HTTPD makes at the end of the start. HTTPD changes both values

to seconds and compares them with **CLIENT_TIMEOUT**. If one is not larger, HTTPD writes **HTTPD026E** for **SESSION_TIMEOUT** or **HTTPD032E** for **SESSION_MAXAGE** and goes on. A limit of 0 is off and is not compared.

```
PORT=8080
DOCROOT=/www
CLIENT_TIMEOUT=120
SESSION_TIMEOUT=1
SESSION_MAXAGE=2
```

```
S HTTPD,M=PRMSESS
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.30.24
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD026E SESSION_TIMEOUT (1 MIN) <= CLIENT_TIMEOUT (120 SEC), RAISE IT
HTTPD032E SESSION_MAXAGE (2 MIN) <= CLIENT_TIMEOUT (120 SEC), RAISE IT
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

The Document Root, the Code Page, the Time Zone and the Trace

The Document Root

DOCROOT is the UFS directory that HTTPD puts in front of the path of a request. The default is **/www**. The value must begin with a slash and HTTPD keeps at most 127 characters. HTTPD removes one slash at the end. A value without the first slash is ignored without a message, and the earlier value stays. Look at **D C** after a change.

With **DOCROOT=/www/docs** the request for **/manual.txt** reads **/www/docs/manual.txt**, and the request for the root gives the page **index.html** of that directory:

```
$ curl -i http://localhost:8080/manual.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:51:01 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 85

HTTPD manual draft
==================
The server serves this file from the UFS root.
```

```
$ curl -i http://localhost:8080/
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:51:01 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/html
X-Content-Type-Options: nosniff
Transfer-Encoding: chunked

<!DOCTYPE html>
<html><head><title>Documents</title></head>
<body><h1>Documents</h1>
<p>This is docs/index.html.</p></body></html>
```

A file in the old root is no longer reachable:

```
$ curl -i http://localhost:8080/hello.txt
HTTP/1.1 404 Not Found
```

```
Date: Tue, 27 Oct 2026 05:51:02 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

404 document "/hello.txt" not found
```

The identity of the task needs READ access to the document root (Chapter 6, "Authentication and Security" on page 55). The serving of files is described in Chapter 7, "Serving Files" on page 69. The keyword **UFS** (default 1) tells HTTPD to create its UFS system at start. Leave it at the default.

The Code Page

CODEPAGE selects the table that translates between the ASCII on the wire and the EBCDIC in HTTPD. The names are **CP037** (the default), **IBM1047** and **LEGACY**, in any case. **LEGACY** is the table of HTTPD 3.3.x. An empty value is **CP037**. Another name gives **HTTPD070E**, and HTTPD uses **CP037**. **D C** shows the name as the member has it (**HTTPD134I**), also when it is not one of the three.

The Time Zone

There is no Parmlib keyword for the time zone. HTTPD takes the offset from the system. The command **F HTTPD,D TI** shows the time in GMT and the local time with the offset in minutes:

```
F HTTPD,D TI
HTTPD100I CONS(0) "D TI"
HTTPD142I TIME 2026/10/27 05:44:39 GMT
HTTPD143I TIME 2026/10/27 00:44:39 LOCAL OFFSET=-300
```

The **Date** header of every response is in GMT. The old keyword **TZOFFSET** is retired ("Retired Keywords" on page 44). The message that it gives ends with the text **SET TZ IN THE SYSENV OR ENVIRON DD TO OVERRIDE IT FOR ALL TASKS**.

The Trace

DEBUG=1 switches the trace on. It has one level: a value above 0 means on and 5 means the same as 1. With the trace on, HTTPD writes its trace lines to the DD name **HTTPDBG** of the procedure. If HTTPD cannot open the DD name, it writes **HTTPD020E** and **HTTPD021I**, and the trace goes to **SYSTERM**. If a write to the trace fails, for example because the data set is full, HTTPD writes **HTTPD039W** once, switches the trace off and goes on. The trace stays off until the next start, and it ends at the point of the failure. Give the DD name more room and restart the server to trace again. Use the trace for a problem and switch it off after it. Chapter 12, "Troubleshooting" on page 141 helps with problems. The trace of **DEBUG** is not the request trace of the next section.

The Request Trace

When a request is slow, you need to know where its time went. The request trace is a table inside the server. For each request it records the time of each phase (the request line, the credential, the module, the send) and a few counters. You read it with **curl**. The trace is off by default, and it is not the trace of **DEBUG** ("The Document Root, the Code Page, the Time Zone and the Trace" on page 39). This section switches it on, checks it and switches it off. How to find a slow request with it is in "A Request Is Slow" on page 157.

Switch the trace on with the keyword **TRACE**. Its value is the number of requests that the table holds: 0, or a number from 16 to 1024. The value 0 is off. This member holds 64 requests. The routes are the ones of the lab server, and the route of HTTPDSRV is the one that shows the table:

```
PORT=8080
DOCROOT=/www
TRACE=64
MOD=MVSMF /zosmf/info                    AUTH=NONE
MOD=MVSMF /zosmf/services/authenticate   AUTH=NONE
MOD=MVSMF /zosmf/*                       AUTH=TOKEN
MOD=HTTPDSRV /.dsrv                      AUTH=FORM
LOC=/pub/*                               AUTH=NONE
```

The server reads the keyword at the start, and no command changes it. Stop the server and start it again. The message **HTTPD066I** says that the trace is on, how many requests the table holds and how much storage it uses:

```
S HTTPD,M=TRCON
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.28.35
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/info
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/services/authenticate
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD036I MODULE HTTPDSRV REGISTERED FOR /.dsrv
HTTPD417I LOCATION /pub/* REGISTERED
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD066I REQUEST TRACE ON, 64 RECORDS, 8736 BYTES
HTTPD061I STARTING SOCKET THREAD    TCB(9CD6E8) TASK(177FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(180B40)   TCB(9BBDD8) TASK(18EFC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(180A40)   TCB(9BBAA8) TASK(19FFC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(1809C0)   TCB(9BB298) TASK(1B0FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

Each record needs 136 bytes, and the block has a header of 32 bytes. 64 records are 8,736 bytes, 256 are 34,848 and 1024 are 139,296. The server gets the block from its region at the start and frees it at the stop. If the region has no room for it, the server writes **HTTPD067W** and runs without the trace. Give the task a larger region, or a smaller value.

The table holds the most recent requests, and a new request replaces an old one. Choose a number that is larger than the number of requests that the server serves in the time you want to look back. 64 is enough to start with.

The display module HTTPDSRV shows the table, with the target **TRACE**. Without a route for HTTPDSRV there is no way to read the trace (“The Display Modules over HTTP” on page 90). On a server where the trace is off, the answer is one line:

```
$ curl -s -u IBMUSER:****** "http://localhost:8080/.dsrv?target=TRACE"
#HTTPTRC off
```

A value that is not valid does not stop the start. The server writes **HTTPD444W** and runs with the trace off. A value of 8 is too small:

```
PORT=8080
DOCROOT=/www
TRACE=8
MOD=MVSMF /zosmf/info                    AUTH=NONE
MOD=MVSMF /zosmf/services/authenticate   AUTH=NONE
MOD=MVSMF /zosmf/*                       AUTH=TOKEN
MOD=HTTPDSRV /.dsrv                      AUTH=FORM
LOC=/pub/*                               AUTH=NONE
```

```
S HTTPD,M=TRCBAD
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.27.14
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD444W INVALID TRACE VALUE "8" (0 OR 16-1024), TRACE OFF
```

```
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/info
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/services/authenticate
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD036I MODULE HTTPDSRV REGISTERED FOR /.dsrv
HTTPD417I LOCATION /pub/* REGISTERED
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CD6E8) TASK(177FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(189D00)   TCB(9BBDD8) TASK(18BFC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(189C00)   TCB(9BBAA8) TASK(19CFC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(189B80)   TCB(9BB778) TASK(1ADFC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

To switch the trace off, remove the line or write **TRACE=0**, and restart the server.

Note: The table holds the address of the client and the first 23 characters of the path of every request, and anyone who may call the route of HTTPDSRV can read them. Switch the trace on to look for a problem, and off when you have found it. Give the route of HTTPDSRV an **AUTH=** ("What authentication they need" on page 90).

Recording SMF Records

HTTPD can write one SMF record for a request, and one for a connection. The default is that it writes none. To switch the recording on, choose a level.

Table 12. The SMF levels

Level	HTTPD writes
NONE	No record. This is the default.
ERROR	A request record for each request with the status 400 or more.
AUTH	The same requests as **ERROR**. The level is named for the statuses 401 and 403.
ALL	A request record for every request, and a session record for every connection.

There are two ways to set the level.

1. Write **SMF=ALL** in the member. The level is in effect from the start. With **SMF=ALL TYPE=244** you also choose the record type. The default type is 243, and the type must be in the range 128 to 255.
2. Use the command **F HTTPD,SET STATS ALL** while the server runs. The change lasts until the server stops. It does not change the member.

```
F HTTPD,SET STATS ALL
HTTPD100I CONS(0) "SET STATS ALL"
HTTPD126I SMF LEVEL SET TO ALL
```

Check the level with **D C**. The line **HTTPD132I** shows the type and the level (the capture in "Reading Back What HTTPD Parsed" on page 35). In the member, HTTPD compares the level with the four names for as many characters as you write. **ERR** is **ERROR**. **A** is **AUTH**, because **AUTH** comes before **ALL**. Write the whole name. The console command **SET STATS** accepts only the whole name. A word that is no level gives **HTTPD431W** in the member and **HTTPD126E** at the console. A type outside the range gives **HTTPD432W**.

```
F HTTPD,SET STATS BOGUS
HTTPD100I CONS(0) "SET STATS BOGUS"
HTTPD126E INVALID SET STATS VALUE "BOGUS"
```

If the level is above **NONE** at the start and SMF is not active in the system, HTTPD writes **HTTPD430W**. The server runs and writes no records. It tests for each record whether SMF is active, so the records start when SMF becomes active.

To find the records, select the record type (243 unless you chose another) with the SMF dump program of your site. The SMF parameters of the installation decide whether the system keeps that type. The layout of both records, field by field, is in 'SMF Records' in the *HTTPD Reference*. The counters that the display commands show (requests, errors, bytes) count at every level, also at **NONE**.

What a Bad Line or a Bad Value Does

HTTPD does not stop for most typing errors in the member. It writes a message, skips the line, and starts. After each change of the member, read the console lines between **HTTPD000I** and **HTTPD001I**.

A Line Without an Equals Sign

A line that is not a comment and has no equals sign gives **HTTPD026W** with the text of the line.

```
PORT=8080
DOCROOT=/www
this line has no equals sign
```

```
S HTTPD,M=PRMLINE
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.26.43
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD026W UNRECOGNIZED CONFIGURATION LINE: this line has no equals sign
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

An Unknown Keyword

A keyword that HTTPD does not know gives **HTTPD029W** with the keyword. A misspelling of a real keyword looks the same. The setting is not applied, so the default stays.

```
PORT=8080
DOCROOT=/www
COLOUR=BLUE
```

```
S HTTPD,M=PRMKEY
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.26.28
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD029W UNKNOWN CONFIGURATION KEYWORD: COLOUR
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

A Bad Value

What happens to a value that is not valid depends on the keyword. There are four cases:

Table 13. The reactions to a bad value

Reaction	Keywords and example

Kept	The earlier value stays and HTTPD writes nothing. A **MINTASK** of 0 or a text, and a **DOCROOT** that does not start with a slash, are of this kind.
Limited	HTTPD changes the value to the nearest allowed one. A **MAXTASK** of 300 becomes 255. **KEEPALIVE_MAX** is limited to 1 to 255.
Read as 0	A text that is not a number counts as 0. For the flags such as **DEBUG** and **UFS** this means off.
Message	A named value has its own warning: **HTTPD070E** for the code page, **HTTPD424W** for the realm, **HTTPD431W** and **HTTPD432W** for SMF, **HTTPD443W** for **RESIDENT**, **HTTPD023E** for the port (the server goes on with 8080), **HTTPD444W** for **TRACE** (the trace stays off).

Because the first case is silent, check the result with **D C** (“Reading Back What HTTPD Parsed” on page 35).

Refusals

In some cases HTTPD does not start. The message **HTTPD420E** says that the authorization policy of the routes is incomplete. The server ends with the return code 8. Two causes are the common ones.

- A route has an **AUTH=** value that does not exist. The message **HTTPD419E** says that the policy of that route is lost.
- The keyword **LOGIN** has a value other than **NONE**.

```
PORT=8080
DOCROOT=/www
LOC=/pub/*  AUTH=BASC
```

```
S HTTPD,M=PRMAUTH
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.28.50
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD411E UNKNOWN AUTH MODE 'BASC' -- USE NONE, FORM, BASIC OR TOKEN
HTTPD419E LOC=/pub/* COULD NOT BE REGISTERED -- ITS AUTH POLICY IS LOST
HTTPD420E ROUTE AUTHORIZATION POLICY INCOMPLETE -- HTTPD WILL NOT START
HTTPD098I HTTPD SHUTTING DOWN
HTTPD416I STATS: 0 REQUESTS, 0 ERRORS, 0 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
IEF404I HTTPD - ENDED - TIME=05.28.50
$HASP395 HTTPD    ENDED
```

Chapter 6, “Authentication and Security” on page 55 explains the values of **AUTH=**. The messages are in ‘Messages’ in the *HTTPD Reference*.

Retired Keywords

Older members may contain keywords that HTTPD no longer uses. Each one gets a message so that nobody believes it still works.

Table 14. The retired keywords

Keyword	Message	What to do
LOGIN=	**HTTPD048W** for an empty value or **NONE**, otherwise	Delete it and write **AUTH=** on each route. Any value other than **NONE** refuses the start, because the routes without **AUTH=** would turn public.

	HTTPD048E and **HTTPD420E**	
TZOFFSET=	**HTTPD025W**, then **HTTPD025I**	Delete it. The offset of the system applies (“The Document Root, the Code Page, the Time Zone and the Trace” on page 39).
RECLAIM=	**HTTPD422W**	An option of a **MOD** route, not a keyword. Delete it. The message says that the reclaim of the storage of modules is always on.
CGI=	**HTTPD410W**	Write **MOD=** instead. HTTPD reads the line as a **MOD=** route.

With this member

```
PORT=8080
DOCROOT=/www
LOGIN=NONE
```

HTTPD starts, and the console shows:

```
S HTTPD,M=PRMLOGN
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.27.29
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD048W LOGIN IS RETIRED -- USE AUTH= ON EACH MOD=/LOC= ROUTE
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

With **LOGIN=BASIC** the start is refused:

```
PORT=8080
DOCROOT=/www
LOGIN=BASIC
```

```
S HTTPD,M=PRMLOGB
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.27.44
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD048E LOGIN BASIC IS RETIRED -- ROUTES WITHOUT AUTH= WOULD BECOME PUBLIC
HTTPD420E ROUTE AUTHORIZATION POLICY INCOMPLETE -- HTTPD WILL NOT START
HTTPD098I HTTPD SHUTTING DOWN
HTTPD416I STATS: 0 REQUESTS, 0 ERRORS, 0 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
IEF404I HTTPD - ENDED - TIME=05.27.44
$HASP395 HTTPD    ENDED
```

Chapter 5. Routes and Server Modules

A route ties a path of a request to what HTTPD does with it. Without a route, HTTPD serves every path as a file. With a route, a path can go to a program (a server module), or it can stay a file and ask for a login first. This chapter shows how to write the routes in the Parmlib member, in which order, and what happens when a module is not there. The login that a route asks for is the subject of Chapter 6, “Authentication and Security” on page 55. The full syntax and the full list of messages are in ‘Routes and Authentication Policy’ in the *HTTPD Reference*.

The Two Kinds of Route

A route is one line in the member. There are two forms.

Table 15. The route lines

Line	Form
MOD	**MOD=program [pattern] [AUTH=mode] [RES=class:resource]**
LOC	**LOC=pattern [AUTH=mode] [RES=class:resource]**

A **MOD** route names a program. HTTPD links to the program, and the program answers the request. A **LOC** route has no program. HTTPD serves the path as a file from the document root, as it does without a route. The only difference is the policy: a **LOC** route can ask for a login or a RACF permission, and a path without a route cannot.

Both forms have the same options. **AUTH=** says how HTTPD asks a client for a login. **RES=** names a RACF resource that the user must be allowed to read. Both are explained in Chapter 6, “Authentication and Security” on page 55. A route without any option is public.

MOD= Routes

The first word after **MOD=** is the name of the load module. HTTPD changes it to upper case and keeps at most 8 characters. A longer name is cut without a message. The second word is the pattern: the path that the route claims.

```
MOD=MVSMF /zosmf/* AUTH=TOKEN
```

This line sends every path that starts with **/zosmf/** to the module **MVSMF**. The patterns are described in “The Order of the Routes and the First Match” on page 48.

If you leave out the pattern, HTTPD builds one from the program name: ***.** and the name in lower case. The line **MOD=REXX** becomes the pattern ***.rexx**. A pattern that starts with ***.** makes an extension route. For a request on an extension route HTTPD sets the variable **SCRIPT_FILENAME** to the document root followed by the path of the request, so that the module knows which file the client asked for. No other route gets this variable.

Each **MOD** line adds one route. Many routes can use the same module, as the examples in this chapter show. HTTPD reads at most 8 words of a line. Words after the eighth are ignored without a message.

LOC= Routes

A **LOC** route is the way to protect part of the document tree. The route does not change what HTTPD serves. It decides who may get it.

```
LOC=/private/*   AUTH=BASIC
LOC=/pub/*       AUTH=NONE
```

The first line asks for a user and a password for every path below **/private/**. The second line makes every path below **/pub/** public explicitly.

Two details prevent surprises:

- A pattern that ends in **/*** does not claim the directory itself. A route for **/private/*** does not cover the path **/private**. Add a plain-path route when you need that.
- A line that is only options, such as **LOC=AUTH=BASIC RES=FACILITY:HTTPD.ADMIN**, has no path. HTTPD does not start with such a line, because the path would be public ("Warnings and Refusals While HTTPD Reads the Routes" on page 53).

The serving of the files is described in Chapter 7, "Serving Files" on page 69.

The Order of the Routes and the First Match

HTTPD keeps all routes, **MOD** and **LOC** together, in one table in the order of the lines. For a request it walks the table from the top and uses the first route that matches. It never looks for the best match. The order of the lines is the only thing that sets the priority. The policy of the first route is the whole policy of the request. A later route that also matches is not asked.

Write the specific routes first and the general ones last.

What a Pattern Matches

HTTPD matches the path of the request, without the query string. A pattern is of one of two kinds.

Table 16. The two kinds of pattern

Kind	Rules
Plain path	The path and the pattern must be equal. The case of the letters does not count. **/status** matches **/STATUS** and does not match **/status/**.
With * or ?	The whole path must match. The case of the letters counts. * stands for any characters, also a slash. At the end of the pattern it also stands for no character, so **/pub/*** matches **/pub/**. Anywhere else it stands for at least one character. ? stands for one character.

The captures below use this member. It has two plain-path routes, both protected by Basic authentication:

```
PORT=8080
DOCROOT=/www
REALM=MVS.EXAMPLE
LOC=/hello.txt                AUTH=BASIC
LOC=/pub                      AUTH=BASIC
```

A plain path claims exactly that path and ignores the query string. The request for **/hello.txt** asks for a login, and so does the request with a query:

```
$ curl -i http://localhost:8080/hello.txt
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:53:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

```
$ curl -i "http://localhost:8080/hello.txt?x=1"
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:53:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

The route **`LOC=/pub`** does not claim a path below it. The file **`/pub/notice.txt`** is public:

```
$ curl -i http://localhost:8080/pub/notice.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:53:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 23

This notice is public.
```

A path that no route claims is public and is served as a file:

```
$ curl -i http://localhost:8080/README.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:53:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 36

notes without a file name extension
```

If you want to protect everything below a directory, write **`LOC=/pub/*`**. A route for a directory and for everything below it needs two lines, one for the directory and one with the trailing **`/*`**.

The First Line Wins

Two routes for the same path show the rule. In the first member the Basic route stands first:

```
PORT=8080
DOCROOT=/www
REALM=MVS.EXAMPLE
LOC=/private/*                  AUTH=BASIC
LOC=/private/*                  AUTH=NONE
```

The first route claims the path. The second route is never used, and the request needs a login:

```
$ curl -i http://localhost:8080/private/secret.txt
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:50:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
```

```
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

In the second member the public route stands first:

```
PORT=8080
DOCROOT=/www
REALM=MVS.EXAMPLE
LOC=/private/*                  AUTH=NONE
LOC=/private/*                  AUTH=BASIC
```

Now the same request is public. The route that asks for the login is dead, and HTTPD gives no message:

```
$ curl -i http://localhost:8080/private/secret.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:50:40 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 43

private: any logged-in user may read this.
```

Note: A general route above a specific one hides it. A line such as **`LOC=/* AUTH=NONE`** makes all routes below it unreachable. If it stands above a protected route, that route protects nothing. HTTPD does not warn about this. After a change of the routes, request the protected path without a login and check that you get the 401.

The Fixed Paths

The paths **`/login`** and **`/logout`** belong to HTTPD. A request for one of them goes to the login code of HTTPD whatever the routes say. Do not write a route for them. The paths that match **`/login.*`** and **`/favicon.*`** are exempt from the login for the methods GET and HEAD, so that the login page can load its parts (Chapter 6, “Authentication and Security” on page 55).

The Standard Module: mvsMF

The module that HTTPD is mostly used for is mvsMF. It answers the REST API of z/OSMF below the path **`/zosmf/`**. mvsMF is a separate product. Its load module is called **`MVSMF`**. This is the member that the captures of this guide use. It has the routes for mvsMF, and it routes the display modules (Chapter 8, “Operating HTTPD” on page 79).

```
# HTTPD lab configuration (ug-ops, ug-security, ug-routes)
PORT=8080
DOCROOT=/www
REALM=MVS.EXAMPLE
MOD=MVSMF /zosmf/info                    AUTH=NONE
MOD=MVSMF /zosmf/services/authenticate   AUTH=NONE
MOD=MVSMF /zosmf/restconsoles/*          AUTH=TOKEN RES=FACILITY:MVSMF.CONSOLE
MOD=MVSMF /zosmf/*                       AUTH=TOKEN
MOD=HTTPDSRV /.dsrv                      AUTH=FORM
MOD=HTTPDM /.dm                          AUTH=BASIC
MOD=HTTPDMTT /.dmtt                      AUTH=BASIC
LOC=/pub/*                               AUTH=NONE
LOC=/private/*                           AUTH=BASIC
LOC=/form/*                              AUTH=FORM
LOC=/token/*                             AUTH=TOKEN
LOC=/admin/*                             AUTH=BASIC RES=FACILITY:HTTPD.ADMIN
LOC=/denied/*                            AUTH=BASIC RES=FACILITY:HTTPD.DENIED
```

Read the four **`MVSMF`** routes from top to bottom.

- **/zosmf/info** and **/zosmf/services/authenticate** are **AUTH=NONE**. HTTPD lets the request through, and mvsMF asks for the password itself. A request to **/zosmf/info** without a password gets its 401 from mvsMF:

```
$ curl -i http://localhost:8080/zosmf/info
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:53:01 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Length: 0
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Pragma: no-cache
X-Content-Type-Options: nosniff
Content-Language: en
```

- **/zosmf/restconsoles/*** is **AUTH=TOKEN** and has a **RES=**. The user needs READ on the profile **MVSMF.CONSOLE** in the class **FACILITY**.
- **/zosmf/*** is **AUTH=TOKEN**. It takes everything else.

The order matters. The route for the console stands before **/zosmf/***, because the general route also matches the console path. If you swap the two lines, the console path falls to the general route, and the check of the resource is lost.

The meaning of **TOKEN** and of **RES=** is in Chapter 6, “Authentication and Security” on page 55.

Where the Module Must Be

HTTPD links to a module by name, in the task of the server. The system finds the module in the **STEPLIB** of the procedure or in the link list. A module in another library is not found. Put the module into the library that the **STEPLIB** names, or add the library to the **STEPLIB**.

Whether the library is APF-authorized decides the message **HTTPD002I** and the protection key in which the system fetches the module. This is described in ‘The Started Task and the Package’ in the *HTTPD Reference*. Because of that key, a module must not change its own static storage (Chapter 9, “Writing a Server Module” on page 97).

Modules Held in Storage: RESIDENT

By default HTTPD loads each module of a **MOD** route once, when it has read the member and before it serves a request. It holds the copy until it stops. A request then does not fetch the module from the library. HTTPD does this once for each program name, however many routes use the name. The member keyword is **RESIDENT**, with the values **YES** (the default) and **NO**.

D C shows the number of modules held in the line **HTTPD137I**. The member above uses four programs, and the line shows **MODULES HELD 4**.

A module that HTTPD cannot hold still works. Each request fetches it, as with **RESIDENT=NO**. The start tells you why:

Table 17. The messages for resident modules

Message	Meaning
HTTPD440W	HTTPD did not find the module at the start. It does not hold it. If the module is in the library later, the route works.
HTTPD441W	The module is not reentrant and not serially reusable. HTTPD does not hold it.

HTTPD442W	The load failed. The message gives the abend code and the reason code.

With **RESIDENT=NO** HTTPD does not look at the modules at the start. A missing module is then silent until the first request (see “A Route to a Missing Module”).

Context Blocks for Modules

A server module must not keep data in writable static storage. When a module needs data that lives from one request to the next, it asks HTTPD for a context block. The module calls the function **http_cgictx_get** with an 8-byte name (an eyecatcher) and a size. The first call for a name creates the block. Every later call with the same name, from any worker, returns the same block. The block stays until the server stops. HTTPD never frees it.

The keyword **CGI_CONTEXT_POINTERS** is the size of the table in which HTTPD records these blocks. Each name takes one entry. The table has one entry more than the number. The default is 255, which is also the largest value. If the table is full, the call returns no block (NULL) and the module has to deal with that. A module does not need the keyword to run. Leave it at the default unless the documentation of a module tells you to change it. How a module uses the block is described in Chapter 9, “Writing a Server Module” on page 97 and in ‘The Server Module Interface’ in the *HTTPD Reference*.

A Route to a Missing Module

A route to a module that is not in the library is not an error of the member. HTTPD registers the route and tells you at the start:

```
PORT=8080
DOCROOT=/www
MOD=NOSUCH /nosuch/*            AUTH=NONE
```

```
S HTTPD,M=PRMNOSU
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.34.49
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE NOSUCH REGISTERED FOR /nosuch/*
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD440W MODULE NOSUCH NOT FOUND AT STARTUP, NOT HELD
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD7F0)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD100)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD080)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
IEA703I 806- 4 HTTPD    HTTPD    MODULE ACCESSED NOSUCH
HTTPD908E EXTERNAL PROGRAM NOSUCH was not found
```

The message **HTTPD036I** says that the route exists. The message **HTTPD440W** says that the module is not found and not held. The server starts. The failure comes with the first request:

```
$ curl -i http://localhost:8080/nosuch/x
HTTP/1.1 503 Service Unavailable
Date: Tue, 27 Oct 2026 05:50:25 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
Transfer-Encoding: chunked

External program NOSUCH was not found
```

The client gets the status 503 and the text **`External program NOSUCH was not found`**. The console shows the system message **`IEA703I 806- 4`** and, after it, **`HTTPD908E`** (the last two lines of the start capture above). The completion code S806 means that the system found the module neither in the **`STEPLIB`** nor in the link list. If you see this, check the name in the route, then the library ("Where the Module Must Be" on page 51). The other reasons for a failure are listed in 'Routes and Authentication Policy' in the *HTTPD Reference*.

With **`RESIDENT=NO`** the start is silent about the module:

```
PORT=8080
DOCROOT=/www
RESIDENT=NO
MOD=NOSUCH /nosuch/*            AUTH=NONE
```

```
S HTTPD,M=PRMRESN
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.34.34
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE NOSUCH REGISTERED FOR /nosuch/*
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD7F0)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD100)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD080)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

Warnings and Refusals While HTTPD Reads the Routes

HTTPD reads the whole member before it decides whether to start. Most mistakes in a route give a warning, and the route is registered. One example shows the common ones. The member has a bad **`RESIDENT`** value and four routes with options that HTTPD does not accept:

```
PORT=8080
DOCROOT=/www
RESIDENT=MAYBE
MOD=NOSUCH /nosuch/*            AUTH=NONE
LOC=/a/*                        AUTH=NONE RES=FACILITY:HTTPD.ADMIN
LOC=/b/*                        AUTH=BASIC RES=NOCOLON
LOC=/c/*                        AUTH=NONE BOGUS=1
LOC=/d/*                        RES=FACILITY:HTTPD.NOPROFILE
```

```
S HTTPD,M=PRMRTE
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.35.14
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD443W INVALID RESIDENT VALUE "MAYBE" (YES|NO)
HTTPD036I MODULE NOSUCH REGISTERED FOR /nosuch/*
HTTPD414W AUTH=NONE IGNORES RES=FACILITY:HTTPD.ADMIN (PUBLIC ROUTE)
HTTPD417I LOCATION /a/* REGISTERED
HTTPD412W IGNORING MALFORMED RES= 'NOCOLON' (NEED CLASS:RESOURCE)
HTTPD417I LOCATION /b/* REGISTERED
HTTPD413W IGNORING UNKNOWN ROUTE OPTION 'BOGUS=1'
HTTPD417I LOCATION /c/* REGISTERED
HTTPD417I LOCATION /d/* REGISTERED
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD425W NO PROFILE FOR FACILITY:HTTPD.NOPROFILE -- /d/* NOT GATED
HTTPD440W MODULE NOSUCH NOT FOUND AT STARTUP, NOT HELD
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD188)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD088)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD008)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

Read the console lines one after the other.

- **HTTPD443W** is the wrong **RESIDENT** value. The earlier value stays.
- **HTTPD414W** is for **/a/*** . The route is public (**AUTH=NONE**), so HTTPD ignores the **RES=**.
- **HTTPD412W** is for **/b/*** . The text **NOCOLON** has no colon. HTTPD ignores the **RES=**, and the route has **AUTH=BASIC**.
- **HTTPD413W** is for **/c/*** . HTTPD does not know the word **BOGUS=1** and ignores it.
- **HTTPD425W** is for **/d/*** . The route has a **RES=** and no **AUTH=**, so it gets Basic authentication. But the security product has no profile for **FACILITY:HTTPD.NOPROFILE**, so any user who is logged in passes. The text says the route is **NOT GATED**. A mistake in the resource name gives the same message (Chapter 6, “Authentication and Security” on page 55). A class that RAKF does not define gives the same message with RAKF 1.2.x (measured on TK5). With RAKF 2.0.0 (MVS/CE) it gives no message and a 403 for every request. With **RES_NOPROFILE=DENY** in the member the message is **HTTPD426W** and the route refuses every request (“Refusing a Route Whose Profile Is Missing” on page 66).
- **HTTPD440W** is the missing module of the first route.

HTTPD checks only the words it knows. A line such as **MOD=ADMIN /admin/* AUTH BASIC** has the words **AUTH** and **BASIC** separately. HTTPD gives **HTTPD413W** twice and registers a public route. Look for **HTTPD413W** in the log after every change.

In some cases HTTPD refuses to start, because a route that asked for a login would otherwise turn public. An **AUTH=** value that does not exist gives **HTTPD411E**, **HTTPD419E** and **HTTPD420E**, and the return code is 8. The capture for that is in Chapter 4, “Configuring HTTPD” on page 33. A line in which an option stands where the program name or the path belongs also refuses the start, if it asks for a login or a resource. A line with only **AUTH=NONE**, or with no option, gives **HTTPD421W** for **MOD** or **HTTPD415W** for **LOC**, and the start goes on.

A change in a route takes effect when you stop and start HTTPD. To look at the routes that run now, use the display module **HTTPDSRV** (Chapter 8, “Operating HTTPD” on page 79).

Chapter 6. Authentication and Security

HTTPD checks every request against the routes of the Parmlib member (Chapter 5, “Routes and Server Modules” on page 47). A route can ask the client for a user ID and a password, and it can ask the security product whether that user may use a resource. This chapter shows the four ways a route can ask, what the client sees, how long a login lasts, which identity the server itself uses, and what HTTPD does not do. The syntax of the route lines is in Chapter 5, “Routes and Server Modules” on page 47. The display modules that show the logins are in Chapter 8, “Operating HTTPD” on page 79. The rules are described in full in ‘Routes and Authentication Policy’ in the *HTTPD Reference*.

The examples use the member from “The Standard Module: mvsMF” on page 50, the user ID **`IBMUSER`** and the host name **`localhost`**. The passwords in the captures are masked.

How a Request Is Checked

HTTPD checks a request in two stages, before it serves a file or links to a module. This holds for the methods GET, HEAD, PUT, POST and DELETE, for **`MOD`** routes and for **`LOC`** routes alike.

1. **Who are you?** If the route needs a login and the client has none, HTTPD answers with a request to log in. The answer depends on the **`AUTH=`** mode of the route.
2. **May you?** This stage applies only to a route with **`RES=`**. HTTPD asks the security product whether the user may read the resource. If not, the answer is 403.

HTTPD finds the user in the request itself. It tries these sources in this order and takes the first one that gives a user:

1. the cookie **`Sec-Token`**,
2. the cookie **`LtpaToken2`**,
3. the header **`Authorization: Bearer`** with a session token,
4. the header **`Authorization: Basic`** with user ID and password.

The login is bound to the address of the client. A token that was issued to one address does not work from another. Every route accepts every source. The **`AUTH=`** mode of a route does not choose the source. It decides only what a client without a login gets.

HTTPD changes the user ID and the password to upper case and uses the first 8 characters of each.

AUTH=: What a Route Asks For

Each route has one of four modes.

Table 18. The values of AUTH=

Mode	Client without a login	Use it for
`NONE`	Served.	Public content.
`BASIC`	401 with a Basic challenge.	Tools such as curl, and browsers that show a password dialog.

FORM	302 to the login page.	Pages for people at a browser.
TOKEN	401 without a challenge.	REST clients and scripts.

A route without **AUTH=** and without **RES=** is **NONE**. A route with **RES=** and without **AUTH=** is **BASIC**. Write the mode on every route, so that the member says what you mean. A mode that does not exist refuses the start (Chapter 4, “Configuring HTTPD” on page 33).

NONE

The route is public. HTTPD checks no login. The same holds for a path that no route claims.

```
$ curl -i http://localhost:8080/pub/notice.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:52:44 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 23

This notice is public.
```

BASIC

A client without a login gets the status 401 and a Basic challenge. The challenge names the realm (“The Realm” on page 60). A browser shows a password dialog. The body is the text **401 Unauthorized**, and the response has **Cache-Control: no-store**.

```
$ curl -i http://localhost:8080/private/secret.txt
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:52:44 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

A wrong password and an unknown user get the same answer, so a client cannot tell which of the two it was.

```
$ curl -i -u IBMUSER:wrongpw http://localhost:8080/private/secret.txt
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:52:44 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

```
$ curl -i -u NOSUCH:wrongpw http://localhost:8080/private/secret.txt
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:52:44 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

A correct password gets the file.

```
$ curl -i -u IBMUSER:****** http://localhost:8080/private/secret.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:52:44 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 43

private: any logged-in user may read this.
```

If the request has the header **X-CSRF-ZOSMF-HEADER**, whatever its value, HTTPD leaves out the **WWW-Authenticate** header. The answer is still 401. A browser then shows no password dialog.

FORM

A client without a login that sends a GET is sent to the login page. HTTPD answers 302 with **Location: /login**. It stores the address that the client asked for in the cookie **Sec-Uri**, encoded in base64, so that it can send the user back after the login.

```
$ curl -i -c cookies.txt http://localhost:8080/form/page.html
HTTP/1.1 302 Moved Temporarily
Date: Tue, 27 Oct 2026 05:52:56 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Location: /login
Set-Cookie: Sec-Uri=YYaWmZRhl4GHhUuIo5ST; path=/
Transfer-Encoding: chunked
```

The login page asks for a user ID and a password. The fields allow 8 characters.

```
$ curl http://localhost:8080/login | grep -E "<title>|<form|<input|<button"
  <title>Login Required</title>
   <form class="modal-content animate" action="/login" method="post">
      <input type="text" placeholder="Enter Userid" name="userid" maxlength="8" required>
      <input type="password" placeholder="Enter Password" name="password" maxlength="8" required>
    <input type="hidden" name="uri" value="/">
     <button type="submit">Login</button>
```

The form sends a POST to **/login** with the fields **userid** and **password**. A wrong password gets the page again, with the text **Invalid userid or password**. The status is 200. HTTPD closes the connection after every POST that has a body, so it does so after a failed login and after a successful one:

```
$ curl -i -b cookies.txt -d userid=IBMUSER -d password=wrongpw http://localhost:8080/login | grep -E "^HTTP|^Connection|Invalid"
HTTP/1.1 200 OK
Connection: close
<h2 class="message">Invalid userid or password</h2>
```

A correct password gets the status 303 back to the address in **Sec-Uri**. HTTPD deletes **Sec-Uri** and sets the cookie **Sec-Token**. The console shows **HTTPD007I** (“Console Messages for Logins” on page 67).

```
$ curl -i -b cookies.txt -c cookies.txt -d userid=IBMUSER -d password=****** http://localhost:8080/login
HTTP/1.1 303 See Other
Date: Tue, 27 Oct 2026 05:52:56 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Cache-Control: no-store
Location: /form/page.html
Set-Cookie: Sec-Uri=deleted; path=/; expires=Thu, 01 Jan 1970 00:00:00 GMT
Set-Cookie: Sec-Token=<token>
Transfer-Encoding: chunked
```

The client sends the cookie with every later request, and the page is served:

```
$ curl -i -b cookies.txt http://localhost:8080/form/page.html
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:52:57 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/html
X-Content-Type-Options: nosniff
Transfer-Encoding: chunked

<!DOCTYPE html>
<html><head><title>Form</title></head>
<body><h1>Page behind the login form</h1></body></html>
```

A GET of **`/logout`** ends the login. HTTPD clears both cookies:

```
$ curl -i -b cookies.txt -c cookies.txt http://localhost:8080/logout
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:52:57 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Set-Cookie: Sec-Uri=deleted; path=/; expires=Thu, 01 Jan 1970 00:00:00 GMT
Set-Cookie: Sec-Token=deleted; path=/; expires=Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: text/plain
Transfer-Encoding: chunked

You have been logged out.
```

After that the page asks for a login again.

```
$ curl -i -b cookies.txt http://localhost:8080/form/page.html
HTTP/1.1 302 Moved Temporarily
Date: Tue, 27 Oct 2026 05:52:57 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Location: /login
Set-Cookie: Sec-Uri=YYaWmZRhl4GHhUuIo5ST; path=/
Transfer-Encoding: chunked
```

A request with another method than GET or POST to a **`FORM`** route gets the status 200 and a text that says **`Invalid REQUEST_METHOD`**. The form is for browsers. Use **`BASIC`** or **`TOKEN`** for a program.

TOKEN

A client without a login gets the status 401, and HTTPD never sends a challenge, whatever the request holds.

```
$ curl -i http://localhost:8080/token/api.txt
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:52:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

The mode does not mean that Basic is refused. A request with a Basic header is checked and served, like any other route.

```
$ curl -i -u IBMUSER:****** http://localhost:8080/token/api.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:52:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 38
```

```
token area: bare 401 without a login.
```

A client that wants a token asks mvsMF for it. A POST to **/zosmf/services/authenticate** with a Basic header returns a session cookie **LtpaToken2**:

```
$ curl -i -c tok.txt -X POST -u IBMUSER:****** -H "X-CSRF-ZOSMF-HEADER: x" http://localhost:8080/zosmf/services/authenticate
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:53:06 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Set-Cookie: LtpaToken2=<token>; Path=/; HttpOnly; SameSite=Strict
Content-Type: application/json
Content-Length: 52
Cache-Control: no-store
Pragma: no-cache
X-Content-Type-Options: nosniff
Content-Language: en
Set-Cookie: LtpaToken2=<token>; Path=/; HttpOnly; SameSite=Strict

{"returnCode":0,"reasonCode":0,"message":"Success."}
```

HTTPD accepts that cookie on its own **TOKEN** routes, too:

```
$ curl -i -b tok.txt http://localhost:8080/token/api.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:53:06 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 38

token area: bare 401 without a login.
```

The same cookie opens **/zosmf/info**. The next capture sends the cookie without a password:

```
$ curl -i -b tok.txt -H "X-CSRF-ZOSMF-HEADER: x" http://localhost:8080/zosmf/info
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:53:06 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: application/json
Content-Length: 184
Cache-Control: no-store
Pragma: no-cache
X-Content-Type-Options: nosniff
Content-Language: en

{"zosmf_hostname":"localhost","zosmf_port":"8080","zosmf_version":"1","zosmf_full_version":"1.3.0","plugins":
[],"zosmf_saf_realm":"SAFRealm","api_version":"1","zos_version":"MVS 3.8j"}
```

The route **/zosmf/services/authenticate** is **AUTH=NONE** in the member. HTTPD does not ask for a login there, because mvsMF does it itself. A wrong password gets a 401 from mvsMF, with a JSON body:

```
$ curl -i -X POST -u IBMUSER:wrongpw -H "X-CSRF-ZOSMF-HEADER: x" http://localhost:8080/zosmf/services/authenticate
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:53:06 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: application/json
Content-Length: 188
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Pragma: no-cache
X-Content-Type-Options: nosniff
Content-Language: en

{"returnCode":8,"reasonCode":1,"message":"Login failed. Check whether the user ID and password you use for the Basic Auth is correct, and
if the user ID has the required SAF permissions."}
```

When a request that mvsMF does not own has no login, you see the bare 401:

```
$ curl -i http://localhost:8080/zosmf/restfiles/ds?dslevel=SYS1.PARMLIB
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:53:01 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

When to Use TOKEN

Use **TOKEN** for routes that programs call, such as the REST API of mvsMF. A program should get a plain 401 and handle it, not a login page or a Basic challenge that it does not understand. A user at a browser does better with **FORM**. A user of curl is fine with **BASIC**. The three modes accept the same credentials, so a mix on one server causes no conflict.

The Realm

The keyword **REALM** names the server in the Basic challenge. The captures above show **realm="MVS.EXAMPLE"**. The default is the SMF ID of the system, with at most four characters. If the system has none, the default is **MVS**. A value has 1 to 64 characters. It must not contain a control character or one of the characters double quote, backslash, **<**, **>** and **&**. Another value gives **HTTPD424W**, and the default stays. The line **HTTPD136I** of **D C** shows the realm in effect (Chapter 4, “Configuring HTTPD” on page 33).

How Long a Login Lasts

HTTPD keeps what a client proved. For a Basic request it keeps an entry that it finds by the address, the user ID and the password, with the RACF identity that RACF returned. For a form login and for a token it keeps a session of its own. Two keywords limit the time:

Table 19. The limits of a login

Keyword	Default	Meaning
SESSION_TIMEOUT	30	Minutes without a request. Every request starts the time again.
SESSION_MAXAGE	480	Minutes since the login, whatever the client does. Requests do not extend it.

Both values are in minutes. The value 0 switches a limit off. Both values go from 0 to 65535. HTTPD removes the expired logins about once a minute, so a login can last up to a minute longer than the limit says.

The limits must be longer than **CLIENT_TIMEOUT**, which counts in seconds. A login that expires while a request is still read would be dropped under the request. HTTPD compares the limits at the end of the start and warns:

```
S HTTPD,M=PRMSESS
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.30.24
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
```

```
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD026E SESSION_TIMEOUT (1 MIN) <= CLIENT_TIMEOUT (120 SEC), RAISE IT
HTTPD032E SESSION_MAXAGE (2 MIN) <= CLIENT_TIMEOUT (120 SEC), RAISE IT
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

HTTPD026E is for **SESSION_TIMEOUT** and **HTTPD032E** for **SESSION_MAXAGE**. The server starts. Raise the value. Chapter 4, “Configuring HTTPD” on page 33 shows the member.

HTTPD keeps at most 256 logins. When the store is full, a new login is rejected. A single client can hold at most 16 sessions. If it asks for a 17th, HTTPD ends the client’s session that has been idle the longest, but only if it has been idle for at least 60 seconds. Otherwise the new login is refused, and the form answers **Login failed, no session available**. Log out when a script is done, so that it does not use up its places.

A Changed RACF Password and the Cached Login

A Basic request does not call RACF every time. HTTPD looks up the address, the user ID and the password in its cache first. If it finds them, the request is answered from the cache. This is why a Basic route is cheap. It has a consequence when a password changes in RACF.

- **The new password works at once.** It is not in the cache, so HTTPD asks RACF.
- **The old password still works until the cached login ends.** A cache hit does not ask RACF. The login ends when it has been idle for **SESSION_TIMEOUT**, when it is older than **SESSION_MAXAGE**, or when a logout ends it. A client that uses the old password all the time keeps the login alive until **SESSION_MAXAGE**, counted from the first request. Requests do not extend that.

To make a changed password take effect for everyone at once, stop and start HTTPD. That drops all logins.

From HTTPD 4.5, a login that starts a session of its own checks the cached password with RACF again. This happens with a form login and with the authenticate request of mvsMF. If RACF refuses it, HTTPD marks the cached login as stale:

- RACF refuses with the return codes 4 (user not defined), 8 (wrong password), 12 (password expired) or 28 (access revoked). Any other failure says nothing about the password. HTTPD then leaves the cache alone and writes **HTTPD009W**.
- HTTPD never serves a stale login, not for the old password and not for a token.
- A stale login is removed after 10 minutes. The next login or the clean-up does this.
- If RACF accepts the password again within these 10 minutes, the mark is cleared.
- The form shows **Invalid userid or password** when RACF refuses the cached password.

A plain Basic request with the old password is not checked against RACF. It stays in the cache until it ends or until a login of the kinds above marks it.

Note: HTTPD 4.4 and earlier never mark a login as stale. The old password stays valid for Basic requests until the cache entry expires.

The Identity of the Server

RAKF has no table of started procedures. It gives every started task the same account, STC/STCGROUP. On a standard RAKF profile set this account has ALTER access to every data set. HTTPD does not want that. At the start it logs on as a user ID of its own, with RACINIT and no password.

The user ID and the group come from the PARM string of the procedure, not from the Parmlib member:

```
//HTTPD    EXEC PGM=HTTPD,REGION=8M,TIME=1440,
//            PARM='STCUSER=&STCUSER STCGROUP=&STCGRP'
```

The procedure has the symbols **STCUSER** (default **HTTPD**) and **STCGRP** (default **USER**). You can override them in the START command:

```
S HTTPD,STCUSER=WEBSRV,STCGRP=WEBGRP
```

The messages tell what happened. They appear in every start capture of this guide:

```
S HTTPD
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.44.51
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/info
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/services/authenticate
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/restconsoles/*
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CD7C8) TASK(152FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(164D00)   TCB(9BBEB8) TASK(166FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(164C00)   TCB(9BBB88) TASK(177FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(164B80)   TCB(9BB858) TASK(188FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

- **HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT** says that the server runs as the user ID and group shown.
- **HTTPD004W** says that the logon failed, for example because the user ID is not defined. The server keeps the identity that the task had, which is the more powerful one. Define the user ID and start again.
- **HTTPD003W** says that the server could not enter supervisor state for the logon. It also keeps the identity that the task had.
- **RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM: USER:HTTPPRB** belongs to a good start from HTTPD 4.5.1. It is the refusal of the test logon that the server makes after **HTTPD004I** ("A Security System That Accepts Any User ID"). A start without it, or with **HTTPD042W** instead, is not a good start.

The user ID of **STCUSER** needs READ access to the Parmlib member and to the document root. A server module that sets no identity of its own uses the identity of the task.

At the stop the server puts the old identity back before it releases its APF authorization. The authorization was taken under the old account and is given back under the same account. For this reason the user ID of **STCUSER** needs no access to the profile for SVC 244. 'The Started Task and the Package' in the *HTTPD Reference* explains the order.

A Security System That Accepts Any User ID

HTTPD has no user database. A login is a question to the security product: is this user ID valid with this password? The answer is only worth something if the product says no to a user it does not know.

RAKF 1.2.x can say yes to everybody. RAKF 1.2.x, the version on TK5, keeps its users in a table that it loads at the IPL from the member **SYS1.SECURE.CNTL(USERS)**. When the table is missing or invalid, RAKF 1.2.x has nobody to compare with. It answers every logon with return code 0 and hands back the account **STC** in the group **STCGROUP**, the account that it gives to every started task. RAKF 2.0.0 (MVS/CE) behaves differently: it refuses a user that it does not know.

An invalid table is easy to make. On RAKF 1.2.x a line with a blank password is enough: the load stops with **RAKFUIDS2** and **RAKFUIDSX** ("Defining the Users and Profiles in RAKF" on page 20), and nothing of the table is loaded.

Note: This was measured on TK5. TK4- was not measured. The statements in this section about RAKF 1.2.x are for TK5.

How HTTPD Reacts

HTTPD 4.5.1 and later look for this state themselves. At the start, after the logon of the started task, the server makes one logon with the user ID **HTTPPRB** and a password that cannot be right. A security product that checks refuses it. RAKF writes **RAKF0004** for it, and that is the line you see in every healthy start (“A healthy start” on page 79). A product that gives an ACEE back for it accepts any user ID. The server then writes

```
HTTPD042W SECURITY SYSTEM ACCEPTS ANY USERID, ALL LOGINS REFUSED
```

and goes on with the start. The server runs, and it refuses every login:

- A Basic request with a user ID and a password gets 401, as for a wrong password.
- A login with the form (“AUTH=: What a Route Asks For” on page 55) shows the login page again with the text **Invalid userid or password**.
- A request for a route with **AUTH=TOKEN** gets 401.
- A request that needs no login (“What Needs No Login” on page 67) is served as before.

The refusal is the same for every user ID, also for the ones that you have defined. HTTPD writes no message for a refused login, and RAKF writes none either, because the server does not ask it. Only **HTTPD042W** tells you, once, at the start. The next capture shows a start in this state. The system had neither the user table nor the profile table of RAKF:

```
S HTTPD
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=00.27.29
RAKF0003 RESOURCE TABLE INVALID/MISSING: ALLOWED:STC
RAKF0009  STC     ,HTTPD   ,DATASET ,MVSMF.LINKLIB
RAKF0003 RESOURCE TABLE INVALID/MISSING: ALLOWED:STC
RAKF0009  STC     ,HTTPD   ,DATASET ,HTTPD.LINKLIB
RAKF0003 RESOURCE TABLE INVALID/MISSING: ALLOWED:STC
RAKF0009  STC     ,HTTPD   ,FACILITY,SVC244
HTTPD000I HTTPD 4.5.1-DEV (FA5FD19) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
RAKF0001 USERID TABLE INVALID OR MISSING
RAKF0002 STC ACCESS ALLOWED: USER: HTTPD
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0001 USERID TABLE INVALID OR MISSING
RAKF0002 STC ACCESS ALLOWED: USER: HTTPPRB
HTTPD042W SECURITY SYSTEM ACCEPTS ANY USERID, ALL LOGINS REFUSED
RAKF0003 RESOURCE TABLE INVALID/MISSING: ALLOWED:STC
RAKF0009  STC     ,HTTPD   ,DATASET ,SYS1.PARMLIB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/info
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/services/authenticate
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/restconsoles/*
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9AD080) TASK(162FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(174D48)   TCB(9AD828) TASK(176FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(174C48)   TCB(9A0CF8) TASK(187FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(174BC8)   TCB(9A09C8) TASK(198FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1-DEV READY - SERVING /www
```

A client that tries to log in sees only the 401:

```
$ curl -i -u HERC01:****** -H "X-CSRF-ZOSMF-HEADER: true" http://localhost:8080/zosmf/restjobs/jobs
HTTP/1.1 401 Unauthorized
Date: Fri, 30 Oct 2026 21:58:30 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

How to Recognise It

Look at the console at the start. These lines say that RAKF has no valid table:

Table 20. Signs of a RAKF without tables

Message	Meaning
HTTPD042W	HTTPD found that the product accepts any user ID. This is the sign that counts. The server writes it once, at the start.
RAKF0001, **RAKF0002**	RAKF has no valid user table (**USERID TABLE INVALID OR MISSING**) and lets the task in (**STC ACCESS ALLOWED**). In the capture the pair appears twice: for the logon of the started task, and for **HTTPPRB**.
RAKF0003, **RAKF0009**	RAKF has no valid profile table (**RESOURCE TABLE INVALID/MISSING**). The second line names the data set or the resource that was checked. You see the pair for each check, also before **HTTPD000I**.

The two kinds of line do not always come together. A rejected user table gives **RAKF0001**, **RAKF0002** and **HTTPD042W** and no **RAKF0003**. In a healthy start there is exactly one RAKF line from the server, **RAKF0004** for **HTTPPRB**, and no **HTTPD042W**.

The display module HTTPDSRV shows the state in the row **httpd->sec_open** of the server block ("HTTPDSRV, the control blocks" on page 92). The row says **NO** in a healthy server and **YES, LOGIN REFUSED** in this state. You can read the row in a browser only if a route lets you reach HTTPDSRV without a login, because the display modules need a login like any other route. Otherwise read the start messages.

What to Do

1. Correct the tables before the IPL. Check **SYS1.SECURE.CNTL(USERS)** and **SYS1.SECURE.CNTL(PROFILES)** against "Defining the Users and Profiles in RAKF" on page 20. Check the passwords, the columns and the order of the lines.
2. After the IPL, read the log for the lines of the table load. The lines **RAKFPROF7 RAKF PROFILES UPDATED** and **RAKFUIDS4 USER TABLE UPDATED** say that the tables are loaded. The lines **RAKFUIDS2 INPUT DATA INVALID OR OUT OF SEQ.** and **RAKFUIDSX ** PROGRAM TERMINATED **** say that the user table was rejected.
3. **Restart HTTPD** ("Restarting" on page 84). The server makes its test once, at the start, and does not look again. A server that wrote **HTTPD042W** refuses logins until it is stopped and started, also after RAKF is repaired.
4. Read the start again. A good start has **RAKF0004** for **HTTPPRB** and no **HTTPD042W**.

RAKF has to be active before HTTPD asks it anything. On TK5 the member **SYS1.PARMLIB(RAKFINIT)** holds the word **YES**, and RAKF then becomes active at the IPL without a question at the console. This is an operating hint and not something that HTTPD checks. If RAKF was not active when HTTPD started, the logon of the started task fails with **HTTPD004W RACINIT ENVIR=CREATE FAILED ... RC=32** ("The Server Has Not Its Own Identity" on page 154).

Two More Rules

- **The user ID HTTPPRB is reserved.** Do not define a user with this name. The server makes one failed logon with it at every start. For a user of that name it would be one failed password attempt each time.
- **The user ID STC is no login.** HTTPD refuses a login with this user ID in every case, because it is the account of the started tasks on RAKF 1.2.x. The client sees the 401 or the login page, and no message is written.

The test needs the same supervisor state as the logon of the started task. If the server could not enter it (**HTTPD003W**), it makes no test and writes no **HTTPD042W**. Cure **HTTPD003W** first ("The Server Has Not Its Own Identity" on page 154).

HTTPD before 4.5.1 does not look for this state and has no **HTTPD042W**. Use HTTPD 4.5.1 or later where RAKF 1.2.x runs.

RES=: Checking a Profile

A route with **RES=class:resource** adds the second stage. After the user is known, HTTPD asks the security product whether the user may READ the resource. HTTPD asks under the identity of the user, not under that of the server. The profile belongs to you. HTTPD names none.

```
LOC=/admin/*    AUTH=BASIC RES=FACILITY:HTTPD.ADMIN
LOC=/denied/*   AUTH=BASIC RES=FACILITY:HTTPD.DENIED
```

On RAKF you define the profiles in the class **FACILITY** and permit the users. RAKF prints this when you do. It reminds you to activate the change:

```
RDEFINE MVSMF.CONSOLE CLASS(FACILITY) UACC(NONE)
RDEFINE: FACILITY profile defined: MVSMF.CONSOLE
RDEFINE: Run RAKFPROF to activate the changes
PERMIT MVSMF.CONSOLE CLASS(FACILITY) ID(ADMIN) ACCESS(READ)
PERMIT:  access READ    granted to FACILITY/MVSMF.CONSOLE
PERMIT: ID = ADMIN
PERMIT: Run RAKFPROF to activate the changes
RDEFINE HTTPD.ADMIN CLASS(FACILITY) UACC(NONE)
RDEFINE: FACILITY profile defined: HTTPD.ADMIN
RDEFINE: Run RAKFPROF to activate the changes
PERMIT HTTPD.ADMIN CLASS(FACILITY) ID(ADMIN) ACCESS(READ)
PERMIT:  access READ    granted to FACILITY/HTTPD.ADMIN
PERMIT: ID = ADMIN
PERMIT: Run RAKFPROF to activate the changes
RDEFINE HTTPD.DENIED CLASS(FACILITY) UACC(NONE)
RDEFINE: FACILITY profile defined: HTTPD.DENIED
RDEFINE: Run RAKFPROF to activate the changes
END
```

The lines say that the changes are not active until RAKFPROF has run.

A user who may read the resource gets the file:

```
$ curl -i -u IBMUSER:****** http://localhost:8080/admin/report.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:52:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 50

admin report: needs READ on FACILITY HTTPD.ADMIN.
```

A user who may not gets 403:

```
$ curl -i -u IBMUSER:****** http://localhost:8080/denied/data.txt
HTTP/1.1 403 Forbidden
Date: Tue, 27 Oct 2026 05:52:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

403 Forbidden
```

In this lab, **IBMUSER** may read **HTTPD.ADMIN**. The capture above shows no permit for **HTTPD.DENIED**, and the user gets the 403. The 403 has the body **403 Forbidden**. A user without a login never reaches this stage. The user gets the 401 of the first stage.

Note: A profile that does not exist can permit everybody. When the security product answers that no profile covers the resource (return code 4), HTTPD counts the request as permitted. Whether the product gives that answer for an unprotected resource depends on its settings. If it refuses instead (return code 8 or above), the user gets the 403. A typing error in the resource name looks the same as a missing profile. A class that RAKF does not define depends on the level of RAKF. RAKF 2.0.0 (MVS/CE) gives the return code 8, so every user gets the 403 and HTTPD writes no **HTTPD425W**. RAKF 1.2.x (measured on TK5) gives the return code 4, so the class looks like a missing profile. HTTPD therefore asks once for each route with **RES=** at the start. For a route without a profile it writes **HTTPD425W NO PROFILE FOR class:resource -- path NOT GATED**. The route still works, and every user who is logged in passes.
Look for **HTTPD425W** after each change. The probe is not made for **AUTH=NONE** routes. To make such a route refuse instead, set **RES_NOPROFILE=DENY** (“Refusing a Route Whose Profile Is Missing”).

HTTPD asks for the access READ for all methods. It has no separate permission for PUT, POST or DELETE.

Refusing a Route Whose Profile Is Missing

The default is the permissive reading of the note above. A route that was written to close a resource is open to every user who can log in as long as its profile is forgotten, misspelled or not loaded yet. The mvsMF console route (**FACILITY:MVSMF.CONSOLE**) is the case that matters: without its profile every user who can log in may issue operator commands through mvsMF up to 1.3.0.

The keyword **RES_NOPROFILE=DENY** turns the missing profile into a closed route. It is one setting for the whole server, not an option of a route. A switch on each route would have to be remembered on exactly the routes where a profile is forgotten. The setting applies to every route with **RES=** that is not **AUTH=NONE**. Put it in the member:

```
RES_NOPROFILE=DENY
LOC=/pub/*  AUTH=BASIC RES=FACILITY:HTTPD.TESTRES
```

The route is the same as before. At the start, **HTTPD426W** replaces **HTTPD425W** for each route without a profile. The next capture shows the lines that matter. The other lines of the start are left out.

```
HTTPD417I LOCATION /pub/* REGISTERED
HTTPD426W NO PROFILE FOR FACILITY:HTTPD.TESTRES -- /pub/* REFUSES ALL REQUESTS
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

The server starts. A route that refuses is the safe state, and you can define the profile while the server runs. Until the profile exists, a request is answered like this:

Table 21. The answers of a route whose profile is missing

Request	Answer
Without a login	Status 401. The first stage answers, as before.
A user who is logged in, whether or not the user would be permitted	Status 403 with the body **403 Forbidden: no RACF profile protects this resource**. The class and the resource name are not in the answer. They are in **HTTPD426W**.

Define the profile and permit the users, as in “Defining the Users and Profiles in RAKF” on page 20, and run **RAKFPROF**. Do not restart HTTPD. The check is made for each request against the profile as RAKF holds it then. After **RAKFPROF** has run, a user whom the profile permits gets the real answer. A user whom it does not permit gets the plain **403 Forbidden**, the same answer as for a profile that refuses. Before **RAKFPROF** has run, **RDEFINE** and **PERMIT** change nothing the server can see, and the

route still answers with the body about the missing profile. **HTTPD426W** belongs to the start and stays in the log. A start after the profile exists does not write it.

The two bodies tell you which case you have: *no RACF profile protects this resource* means that RAKF has no profile for the resource. Check the class and the name in the route against the profile, and check that **RAKFPROF** ran. The plain **403 Forbidden** means that a profile exists and does not permit the user.

When to turn it on. Turn it on when every route with **RES=** has its profile and the start writes no **HTTPD425W**. On a new system, set it before the first start if the profile definitions are part of the installation. Do not turn it on while a route still depends on the permissive reading, for example a **RES=** that is kept for documentation and has no profile behind it. Drop that **RES=** first. Nothing changes without the keyword. A member that does not have it behaves as before, and **D C** shows **HTTPD138I RES_NOPROFILE ALLOW**.

What the setting does not do. HTTPD sees only the answer of the security product that says "no profile" (return code 4). It cannot tell why the product had none to apply, and with **DENY** every such answer refuses. This guide does not state in which conditions, other than a missing profile, a product gives that answer. The setting does not touch a route with **AUTH=NONE**, and it does not change the answer of a profile that exists.

A wrong value stops the start. A value that is neither **ALLOW** nor **DENY** is not read as **ALLOW**. It gives **HTTPD427E** and **HTTPD420E**, and the server does not start ("A Route Without a Valid Policy" on page 143).

What Needs No Login

Some requests pass without a login, whatever the route says.

- A path that no route claims. HTTPD serves it as a file.
- A route with **AUTH=NONE**. That includes the two **MVSMF** routes for **/zosmf/info** and **/zosmf/services/authenticate** in the member of "The Standard Module: mvsMF" on page 50. mvsMF does its own authentication there.
- The paths that match **/login.*** and **/favicon.***, for the methods GET and HEAD. The case counts. This lets the login page load. A **MOD** route that claims such a path runs its module without a login.
- The paths **/login** and **/logout**. HTTPD handles them itself, whatever the routes or the method.

Everything else needs the login that its route asks for. The display modules are no exception. They have routes with **AUTH=** like any module.

A route that stands above another one decides for the paths of both ("The Order of the Routes and the First Match" on page 48). Check a protected path after each change of the routes without a login. The answer must be 401, 302 or 403.

Console Messages for Logins

HTTPD writes the logins that matter to the operator and nothing about a failed password. That is the client's concern, and it appears in the response. One message is about the security product and not about a login: **HTTPD042W** at the start ("A Security System That Accepts Any User ID" on page 62).

Table 22. The messages for logins

Message	Meaning
HTTPD007I	A form login worked. The message gives the user ID, the IP address of the client and the address of the ACEE.
HTTPD008I	A GET of **/logout** ended a login. The text is **LOGOUT SUCCESSFUL**.

HTTPD008W	The logout failed for a login that HTTPD knows. The browser sees the text **Logout failed.**.
HTTPD009W	From HTTPD 4.5. A login could not create a session because RACF failed for another reason than the password.
HTTPD042W	From HTTPD 4.5.1. At the start. The security product accepted a logon with a user ID and a password that cannot be right. HTTPD refuses all logins until it is started again (“A Security System That Accepts Any User ID” on page 62).

A logout without a login answers **You are not logged in. No logout performed.**, and the console stays quiet.

The next capture shows a session on the console. The line from RAKF in the middle is not an HTTPD message. RAKF reports a login of an unknown user ID, here from a request with the user **NOSUCH**, in its own message **RAKF0004**. HTTPD writes the other two:

```
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:NOSUCH
HTTPD007I USER IBMUSER  IP 192.0.2.17 LOGIN SUCCESSFUL ACEE(9BB190)
HTTPD008I USER IBMUSER  IP 192.0.2.17 LOGOUT SUCCESSFUL
```

The command **D L** lists the logins that exist (Chapter 8, “Operating HTTPD” on page 79).

What HTTPD Does Not Do

HTTPD is a small server. Know the limits before you open it to a network.

- **HTTPD has no TLS code.** It speaks plain HTTP. User IDs, passwords, Basic headers and session cookies cross the network in clear text. Treat the network between the client and the server as one that others can read, or keep the clients on a network that you control.
- **The check asks for READ only.** One route has one **RES=**, and the access is always READ. HTTPD has no separate permission for a method, and no check below the level of a route.
- **Routes match once.** The first route that matches decides. HTTPD does not warn when a general route hides a specific one.
- **A path without a route is public.** There is no default that denies.
- **A wrong resource name opens the route.** HTTPD warns once at the start (**HTTPD425W**) and not again. With **RES_NOPROFILE=DENY** the route is closed instead (“Refusing a Route Whose Profile Is Missing” on page 66). A class that RAKF does not define closes the route for every user with RAKF 2.0.0 (MVS/CE), with or without the keyword, and opens it with RAKF 1.2.x (measured on TK5), like a wrong resource name.
- **A password is at most 8 characters.** HTTPD changes the user ID and the password to upper case and cuts both at 8 characters.
- **The cache can outlive a password change.** See “A Changed RACF Password and the Cached Login” on page 61.
- **HTTPD does not write failed logins to the console.** The client sees the 401 or the form text. RAKF may write a message of its own, as the capture above shows.
- **HTTPD relies on the security product to refuse.** HTTPD 4.5.1 and later test this once at the start. If the product accepts any user ID, HTTPD refuses all logins (“A Security System That Accepts Any User ID” on page 62). The test does not run again while the server runs.

Chapter 7. Serving Files

HTTPD answers a request that no route claims by sending a file from its document root. This chapter shows where those files are, how they get there, and what HTTPD makes of a request for one: the content type, the directory rules, the path rules and the methods it accepts. The sessions use **curl** as in Chapter 3, "A First Request" on page 27. The Reference describes the same rules as a list. It is the chapter "HTTP Behaviour and Limits" in the *HTTPD Reference*, and the keywords are in "The HTTPPRM Keywords".

Where the Files Are

HTTPD does not read files from MVS data sets. It reads them from a UFS file system, which UFSD provides. The Parmlib keyword **DOCROOT** names the directory in that file system where the files are. The default is **/www**. HTTPD puts the value in front of the path of the request: a request for **/hello.txt** reads the file **/www/hello.txt**.

Message **HTTPD001I** names the document root when HTTPD is ready ("What the Console Shows" on page 27). A request for a file that HTTPD cannot open gets the same answer as a request for a file that is not there ("A File That Is Not There" on page 30).

Two parts make the document root:

- **A disk image.** The UFS file system lives in an MVS data set. The release of HTTPD includes an image with the welcome page. You build an image of your own with the host tool **ufsd-utils** ("Building a Web Root Image").
- **A mount.** UFSD attaches the image to the path **/www**. The mount is a statement in the Parmlib member of UFSD, not of HTTPD ("Mounting the Image" on page 70).

The keyword **DOCROOT** has three rules. The value must start with a slash, or HTTPD ignores it without a message and keeps the old value. HTTPD removes one slash at the end of a longer value. HTTPD keeps at most 127 characters. The Reference has the details.

Building a Web Root Image

To publish your own files, build an image on a workstation. The host tool **ufsd-utils** of UFSD does it in three commands. The first creates an empty image. The second copies a directory tree into it. The third lists what is in it:

```
$ ufsd-utils create httpd-webroot.img --size 1M --owner IBMUSER --group SYSPROG
Creating httpd-webroot.img (1M, blksize=4096, inodes=10.0%)
  Volume size:     256 blocks (1.00 MB)
  Block size:      4096 bytes
  Inode blocks:    2 (64 inodes)
  Data blocks:     252 (free: 251)
  Root owner:      IBMUSER/SYSPROG
  Format:          UFS370 v1 (time64 timestamps)

  Upload to MVS:  ufsd-utils upload httpd-webroot.img --dsn YOUR.DATASET.NAME
Done.
$ ufsd-utils cp -r static/ httpd-webroot.img:/
  README.txt -> /README.txt (36 bytes, txt)
  mkdir /admin
    report.txt -> /admin/report.txt (50 bytes, txt)
  mkdir /css
    site.css -> /css/site.css (67 bytes, txt)
  data.json -> /data.json (34 bytes, txt)
```

```
   mkdir /denied
     data.txt -> /denied/data.txt (50 bytes, txt)
   mkdir /docs
     index.html -> /docs/index.html (131 bytes, txt)
     manual.txt -> /docs/manual.txt (85 bytes, txt)
   mkdir /empty
   mkdir /form
     page.html -> /form/page.html (111 bytes, txt)
   hello.txt -> /hello.txt (61 bytes, txt)
   index.html -> /index.html (3573 bytes, txt)
   logo.png -> /logo.png (70 bytes, bin)
   mkdir /private
     secret.txt -> /private/secret.txt (43 bytes, txt)
   mkdir /pub
     index.html -> /pub/index.html (98 bytes, txt)
     notice.txt -> /pub/notice.txt (23 bytes, txt)
   mkdir /token
     api.txt -> /token/api.txt (38 bytes, txt)
$ ufsd-utils ls -l httpd-webroot.img:/
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 .
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 ..
-rwxr-xr-x   1 IBMUSER  SYSPROG          36 2026 Oct 27 05:42 README.txt
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 admin
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 css
-rwxr-xr-x   1 IBMUSER  SYSPROG          34 2026 Oct 27 05:42 data.json
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 denied
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 docs
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 empty
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 form
-rwxr-xr-x   1 IBMUSER  SYSPROG          61 2026 Oct 27 05:42 hello.txt
-rwxr-xr-x   1 IBMUSER  SYSPROG        3573 2026 Oct 27 05:42 index.html
-rwxr-xr-x   1 IBMUSER  SYSPROG          70 2026 Oct 27 05:42 logo.png
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 private
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 pub
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 token
$ ufsd-utils ls -l httpd-webroot.img:/docs
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 .
drwxr-xr-x   2 IBMUSER  SYSPROG         128 2026 Oct 27 05:42 ..
-rwxr-xr-x   1 IBMUSER  SYSPROG         131 2026 Oct 27 05:42 index.html
-rwxr-xr-x   1 IBMUSER  SYSPROG          85 2026 Oct 27 05:42 manual.txt
```

Read the output of the first command as follows. The image is 1 MB. It has 256 blocks of 4096 bytes, and 2 of them hold the inode table with room for 64 inodes. **`--owner`** and **`--group`** set the owner and the group of the root directory. The size and the number of inodes limit how many files and how much data the image holds. Use **`--size`** to make it larger, and see the *UFSD User's Guide* for the other options.

The output of **`cp -r`** has one line for each file and each directory. At the end of each file line the tool says **`txt`** or **`bin`**. It stores a text file in EBCDIC, in the code page IBM-1047, and a binary file as it is. HTTPD translates a UFS text file back with the same table ("Text, Binary and the Code Page" on page 73).

The create command suggests the next step, uploading the image with **`ufsd-utils upload`**. The data set that takes the image is a sequential data set with the record format U and a block size of 4096, the block size of the image. The job **`HTTPWEBR`** in the sample library allocates such a data set under the name **`HTTPD.WEBROOT.UFS`**. The upload must be binary. Chapter 2, "Installing HTTPD" on page 5 has the whole procedure, with the installation of the image that ships with HTTPD.

Mounting the Image

UFSD mounts the image at the start from its own Parmlib member. The statement for the data set above and the default document root is:

```
MOUNT    DSN(HTTPD.WEBROOT.UFS)  PATH(/www)  MODE(RO)
```

`MODE(RO)` mounts the image for reading. HTTPD only reads. The path in the statement must be the value of **`DOCROOT`**, letter for letter. Start UFSD before HTTPD, so that the first request finds the file system. Chapter 2, "Installing HTTPD" on page 5 shows the installation of the image, and the *UFSD User's Guide* describes the Parmlib member and the operator commands. UFSD changes an operator command to upper case before it reads it. A mount point such as **`/www`** therefore comes from the Parmlib statement (Chapter 2, "Installing HTTPD" on page 5).

To replace the content, stop HTTPD and UFSD, upload the new image to the same data set, and start UFSD, then HTTPD. Chapter 8, “Operating HTTPD” on page 79 describes stopping and starting.

Changing the Document Root

The document root need not be the root of the mounted file system. If the image has a directory **docs**, the member below serves that directory as the whole site:

```
PORT=8080
DOCROOT=/www/docs
```

After the restart, the path **/** reads **/www/docs/index.html**:

```
$ curl -i http://localhost:8080/
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:51:01 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/html
X-Content-Type-Options: nosniff
Transfer-Encoding: chunked

<!DOCTYPE html>
<html><head><title>Documents</title></head>
<body><h1>Documents</h1>
<p>This is docs/index.html.</p></body></html>
```

The file **/www/docs/manual.txt** is now the path **/manual.txt**:

```
$ curl -i http://localhost:8080/manual.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:51:01 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 85

HTTPD manual draft
==================
The server serves this file from the UFS root.
```

A file outside the new root is not found, although it is in the image:

```
$ curl -i http://localhost:8080/hello.txt
HTTP/1.1 404 Not Found
Date: Tue, 27 Oct 2026 05:51:02 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

404 document "/hello.txt" not found
```

The file **/www/hello.txt** exists, but the new document root is **/www/docs**. A change of **DOCROOT** takes effect at the next start of HTTPD (Chapter 4, “Configuring HTTPD” on page 33).

Directory Requests

A path that ends in a slash names a directory. HTTPD does not list a directory. It looks for the file **index.html** in it, and if that is not there, for **default.html**. A directory with an index file answers with the file:

```
$ curl -i http://localhost:8080/docs/
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/html
X-Content-Type-Options: nosniff
Transfer-Encoding: chunked

<!DOCTYPE html>
<html><head><title>Documents</title></head>
<body><h1>Documents</h1>
<p>This is docs/index.html.</p></body></html>
```

A directory without **`index.html`** and **`default.html`** gets a 404. The text repeats the path of the request:

```
$ curl -i http://localhost:8080/empty/
HTTP/1.1 404 Not Found
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

404 document "/empty/" not found
```

A directory in a path without the closing slash is not a file, and HTTPD does not add the slash for the client. It does not answer with a redirect:

```
$ curl -i http://localhost:8080/docs
HTTP/1.1 404 Not Found
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

404 document "/docs" not found
```

So a link to a directory has to end in a slash. The path **`/docs`** gets the answer above, and the path **`/docs/`** gets the index file.

Content Types

HTTPD chooses the **`Content-Type`** from the extension of the file name: the text after the last dot in the last part of the path. If that part has no dot, HTTPD looks at the whole path. The comparison ignores case, so **`LOGO.PNG`** and **`logo.png`** are the same.

Table 23. Some content types

Extension	Content-Type	Sent as
`html`, **`htm`**	**`text/html`**	text
`css`	**`text/css`**	text
`js`	**`application/x-javascript`**	text
`txt`	**`text/plain`**	text
`xml`	**`text/xml`**	text
`svg`	**`image/svg+xml`**	text
`png`	**`image/png`**	binary

jpg, **jpeg**	**image/jpeg**	binary
gif	**image/gif**	binary
ico	**image/x-icon**	binary
pdf	**application/pdf**	binary
zip	**application/zip**	binary
any other, or none	**text/plain**	text

The whole table is in “HTTP Behaviour and Limits” in the *HTTPD Reference*. Three captures show the effect. A style sheet is text and has its type:

```
$ curl -i http://localhost:8080/css/site.css
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/css
X-Content-Type-Options: nosniff
Content-Length: 67

body { font-family: sans-serif; margin: 2em; }
h1 { color: #036; }
```

A PNG image has its type, and the body is sent as it is stored. The capture shows only the headers:

```
$ curl -s -D - -o /dev/null http://localhost:8080/logo.png
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:43 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: image/png
X-Content-Type-Options: nosniff
Content-Length: 70
```

A file whose extension is not in the table is sent as **text/plain**, also when the file is JSON:

```
$ curl -i http://localhost:8080/data.json
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 34

{"name": "httpd", "static": true}
```

Note: Unknown extensions are text. HTTPD translates every file with an unknown extension as text, and a binary file of that kind arrives damaged. Keep to the extensions in the table for binary files, or rename the file.

Text, Binary and the Code Page

The last column of the table says how HTTPD sends the body.

- **Text.** HTTPD translates every byte from EBCDIC to ASCII. It uses the table for IBM-1047, whatever the Parmlib keyword **CODEPAGE** says.
- **Binary.** HTTPD sends the bytes as they are stored in the image.

The keyword **CODEPAGE** selects the table for the status line, the headers and the text of the responses that HTTPD writes itself, and for the decoding of the request (see “Paths” below). It has no effect on the content of a file. So a text file in the image has to be in IBM-1047. The tool **ufsd-utils** writes the

files of a text type in that code page (“Building a Web Root Image” on page 69). The chapter Chapter 4, “Configuring HTTPD” on page 33 tells how to set **CODEPAGE**.

Server Side Includes

HTTPD scans a file of the type **text/html** or **text/x-server-parsed-html** for server side includes while it sends it. These are the extensions **htm**, **html**, **shtm** and **shtml**. A directive such as **<!--#include virtual="/part.html" -->** is replaced by the text it names. Because the final size is not known, the response has no **Content-Length** and uses chunked transfer, as the response of the welcome page in “Getting the Welcome Page” on page 28 does. The directives and their limits are in “HTTP Behaviour and Limits” in the *HTTPD Reference*.

Any other file, also a file with the extension **txt**, is sent without a scan.

Paths

HTTPD does five things to the path of a request before it joins it to the document root:

- It cuts off the query. The text after the first question mark is not part of the file name.
- It decodes the path. Each **%xx** becomes the character with that hexadecimal value, translated through the table of **CODEPAGE**. A plus sign stays a plus sign in the path. To ask for a blank in a file name, write **%20**. A percent sign that is not followed by two hexadecimal digits, and the escape **%00**, stay as they are.
- It refuses a path with two dots in a row. The check is on the decoded path, so **%2e%2e** does not help either. The answer is a 404. A file name that has two dots in a row, such as **a..b.txt**, cannot be served.
- It cuts the path to 255 characters, and the joined name of the document root and the path to 255 characters. The client is not told.
- It looks at the extension for the content type (“Content Types” on page 72).

The answer for a path that does not name a file is the 404 that the first chapter shows. Its text contains the decoded path:

```
$ curl -i http://localhost:8080/nothere.html
HTTP/1.1 404 Not Found
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

404 document "/nothere.html" not found
```

What Is Not Supported

A static file is read-only for the client and always whole. HTTPD answers a GET and a HEAD. It does not answer anything else for a file.

HEAD. The curl option **-I** sends a HEAD request, as “Choosing a Client” on page 28 shows. HTTPD answers with the status and the headers that a GET gets, and sends no body. A file that is not there gets the 404 of a GET, with no body:

```
$ curl -I http://localhost:8080/nothere.html
HTTP/1.1 404 Not Found
Date: Tue, 27 Oct 2026 17:11:59 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
```

```
Cache-Control: no-store
Content-Type: text/plain
```

The connection stays open, as it does after a GET (“Two Requests on One Connection” on page 30). With **`curl -v -I`** and two URLs, the second HEAD uses the connection of the first:

```
$ curl -v -I http://localhost:8080/hello.txt http://localhost:8080/hello.txt
* Host localhost:8080 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:8080...
* Connected to localhost (::1) port 8080
* using HTTP/1.x
> HEAD /hello.txt HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.14.1
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< Date: Tue, 27 Oct 2026 17:11:59 GMT
< Server: HTTPD Server
< X-Powered-By: MVSLOVERS
< Connection: keep-alive
< Content-Type: text/plain
< X-Content-Type-Options: nosniff
< Content-Length: 61
<
* Connection #0 to host localhost left intact
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 17:11:59 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 61

* Re-using existing http: connection with host localhost
> HEAD /hello.txt HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.14.1
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< Date: Tue, 27 Oct 2026 17:11:59 GMT
< Server: HTTPD Server
< X-Powered-By: MVSLOVERS
< Connection: keep-alive
< Content-Type: text/plain
< X-Content-Type-Options: nosniff
< Content-Length: 61
<
* Connection #0 to host localhost left intact
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 17:11:59 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 61
```

A HEAD gets a **`Content-Length`** only where a GET gets one. An HTML file has none, because HTTPD looks for server-side includes in it (“Server Side Includes” on page 74). A HEAD for such a file has neither a length nor chunked transfer:

```
$ curl -I http://localhost:8080/docs/
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 17:11:59 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/html
X-Content-Type-Options: nosniff
```

The HEAD answer of HTTPD 4.4.0 and earlier is **`501 Not Implemented`**.

Other methods. POST, PUT and DELETE for a static file get **`501 Not Implemented`**. The three methods:

```
$ curl -i -X POST -d x=1 http://localhost:8080/hello.txt
HTTP/1.1 501 Not Implemented
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

501 Not Implemented
```

```
$ curl -i -X PUT -d x=1 http://localhost:8080/hello.txt
HTTP/1.1 501 Not Implemented
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

501 Not Implemented
```

```
$ curl -i -X DELETE http://localhost:8080/hello.txt
HTTP/1.1 501 Not Implemented
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

501 Not Implemented
```

The responses to POST and PUT have **`Connection: close`** because the requests have a body. The response to DELETE, which has none, keeps the connection. A client cannot change a file in the image over HTTP. Replace the image to change the content (“Mounting the Image” on page 70).

A method that HTTPD does not know at all gets **`405 Method Not Allowed`** and the connection is closed. The answer has no **`Allow`** header. HTTPD sends the 405 before it looks at the route, so it is the same for every path:

```
$ curl -i -X OPTIONS http://localhost:8080/hello.txt
HTTP/1.1 405 Method Not Allowed
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

405 Method Not Allowed
```

Ranges. HTTPD does not look at a **`Range`** header. It sends the whole file with status 200 and no **`Content-Range`** header:

```
$ curl -i -H "Range: bytes=0-9" http://localhost:8080/hello.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 61

Hello from the HTTPD document root.
This file is plain text.
```

Conditional requests. HTTPD does not look at **`If-Modified-Since`** either. It never answers with 304, and it sends no **`Last-Modified`** or **`ETag`** header:

```
$ curl -i -H "If-Modified-Since: Tue, 27 Oct 2026 00:00:00 GMT" http://localhost:8080/hello.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 61

Hello from the HTTPD document root.
This file is plain text.
```

A client that sends these headers gets the whole file each time. A server module on a route with a program decides about the methods **GET**, **HEAD**, **POST**, **PUT** and **DELETE** and about the headers for its own paths. Chapter 5, “Routes and Server Modules” on page 47 tells how routes are matched.

Static Locations

A request that no route claims is served from the document root, and nobody needs to log in for it. To ask for a login for a part of the tree, name it in a **LOC=** route. A **LOC=** route has no program. HTTPD serves the path as a static file, as in this chapter, after the check of the route has passed. The Parmlib member of the test system has these lines:

```
  LOC=/pub/*                          AUTH=NONE
  LOC=/private/*                      AUTH=BASIC
  LOC=/admin/*                        AUTH=BASIC RES=FACILITY:HTTPD.ADMIN
  LOC=/denied/*                       AUTH=BASIC RES=FACILITY:HTTPD.DENIED
```

A path under **/pub/** is public:

```
$ curl -i http://localhost:8080/pub/notice.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:52:44 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 23

This notice is public.
```

A path under **/private/** asks for a login, and a request without credentials gets a 401 with a **WWW-Authenticate** header:

```
$ curl -i http://localhost:8080/private/secret.txt
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:52:44 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

A user who logs in but has no access to the resource of the route gets a 403:

```
$ curl -i -u IBMUSER:****** http://localhost:8080/denied/data.txt
HTTP/1.1 403 Forbidden
Date: Tue, 27 Oct 2026 05:52:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

403 Forbidden
```

The syntax of the routes and the order in which HTTPD tries them are in Chapter 5, “Routes and Server Modules” on page 47. The modes of **AUTH=** and the resource check are in Chapter 6, “Authentication and Security” on page 55.

When UFSD Is Not There

HTTPD reads static files through UFSD. It creates the control block of its file system when it starts. This step does not contact UFSD, and HTTPD does not check that UFSD runs. The message **HTTPD044W UNABLE TO INITIALIZE FILE SYSTEM** says that HTTPD could not create that block. It does not say that UFSD is down.

HTTPD asks UFSD for a session when it opens a file, for each request. In each of these cases HTTPD gets no file, and the answer is the 404 of “A File That Is Not There” on page 30:

- HTTPD cannot get a session from UFSD.
- Nothing is mounted at the document root, or the mount is at another path.
- The file or the directory is not in the image.

The Parmlib keyword **UFS=0** switches the file system off. HTTPD then serves no file, and every path that no route claims gets a 404. Leave the keyword at its default if you want to serve files. A route with a program is dispatched without the file system.

When every static request gets a 404, check in this order: that UFSD runs, that the image is mounted (the operator command **F UFSD,MOUNT LIST** shows the mounts), that the mount point is the value of **DOCROOT**, and that the file is in the image. See also Chapter 12, “Troubleshooting” on page 141.

What Can Go Wrong

- **A binary file arrives damaged.** Its extension is not in the table, so HTTPD sent it as text (“Content Types” on page 72).
- **Text shows wrong characters.** The file in the image is not in IBM-1047, or the upload of the image was not binary.
- **A link to a directory gives 404.** The link has no closing slash (“Directory Requests” on page 71).
- **A file with a double dot in its name is not found.** HTTPD refuses any path with two dots in a row (“Paths” on page 74).
- **A browser shows an old copy of a file.** Among the files, only HTML documents get **Cache-Control: no-store**, and HTTPD sends no **Last-Modified** or **ETag** for any file. The browser decides by its own rules for the others.
- **The change of DOCROOT has no effect.** HTTPD reads the Parmlib at start only. Restart it (Chapter 8, “Operating HTTPD” on page 79).

Chapter 8. Operating HTTPD

This chapter is for the operator. It shows how to start and stop HTTPD, what a healthy start and a normal stop look like on the console, and how to look at a running server and change a few of its settings without a restart. It also shows how to read the server from a browser or from **curl** with the display modules, and what to watch on a server that runs for weeks. The complete list of commands and messages is in the *HTTPD Reference*, in the chapters "Console Commands" and "Messages".

The screens in this chapter were taken from a test system. The numbers in them, such as the control block addresses and the times, are those of that system and will differ on yours.

Starting

Start HTTPD with the console command **S**. The installation put the procedure into your procedure library (Chapter 2, "Installing HTTPD" on page 5).

```
S HTTPD
S HTTPD,M=HTTPPRM1
S HTTPD,D='MY.PARMLIB'
```

The first form reads the member that the procedure names, the second reads another member of the same library, and the third reads the member from another library. The two symbolic parameters of the procedure are **M** (the member) and **D** (the data set). The members, the keywords and the route lines are the subject of Chapter 4, "Configuring HTTPD" on page 33 and Chapter 5, "Routes and Server Modules" on page 47.

The server reads the Parmlib member once, at the start. There is no command that reads it again. A change of the member takes effect at the next start ("Restarting" on page 84).

A healthy start

A good start looks like this on the console. The Parmlib member of this server has four routes for the module MVSMF:

```
S HTTPD
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.44.51
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/info
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/services/authenticate
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/restconsoles/*
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CD7C8) TASK(152FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(164D00)   TCB(9BBEB8) TASK(166FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(164C00)   TCB(9BBB88) TASK(177FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(164B80)   TCB(9BB858) TASK(188FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

The first four lines come from the system. The messages of HTTPD come in the order of the table. A message in the table that does not appear above appears only in some cases.

Table 24. The messages of a start, in order

Message	What it tells
HTTPD000I	The version and the commit of the build.
HTTPD005I	The version of LIBC/370 that HTTPD was linked with.
HTTPD006W	Only in a build from a changed working tree. A released build never writes it.
HTTPD002I	How HTTPD got its authority: from the library (module key 0) or by the SVC (key 8). Both are good. Chapter 2, “Installing HTTPD” on page 5 explains the two ways.
HTTPD003W, **HTTPD004I**, **HTTPD004W**	The security identity of the task. **HTTPD004I** says that the task took the user ID and the group of the procedure parameters **STCUSER** and **STCGRP**. The two warnings say that the identity was not changed and the task keeps the one it was started with (Chapter 6, “Authentication and Security” on page 55).
RAKF0004	From HTTPD 4.5.1, one line after **HTTPD004I**: **INVALID ATTEMPT TO ACCESS SYSTEM: USER:HTTPPRB**. It is RAKF refusing the test logon that the server makes. It belongs to a healthy start (“A Security System That Accepts Any User ID” on page 62).
HTTPD042W	Only if the security product accepts any user ID. The server refuses all logins and needs a new start after the cause is gone (“A Security System That Accepts Any User ID” on page 62). A healthy start does not have it.
HTTPD020W	Only if the Parmlib member could not be opened (“A missing Parmlib member is not a refusal” on page 82).
HTTPD036I	One line for each route with a program (**MOD=**). The line names the module and the path.
HTTPD417I	One line for each static location (**LOC=**).
HTTPD430W	Only if the SMF level is above NONE and SMF is not active in the system (“Recording with SMF” on page 94).
HTTPD054I	HTTPD listens on the port.
HTTPD066I	Only if **TRACE=** is set: the request trace is on (“The Request Trace” on page 40).
HTTPD061I	One line for the socket thread and one for each worker that starts. The number of workers is the value of **MINTASK**.
HTTPD001I	HTTPD is ready. The text is **READY - SERVING** and the document root, or **READY - NO DOCUMENT ROOT** on a server that has none.

HTTPD is ready for clients only when **HTTPD001I** has appeared. A start that is stopped before that point ends with a message that says what failed. Chapter 12, “Troubleshooting” on page 141 lists the common ones.

The next start used the lab configuration of this book. It adds the display modules and some static locations:

```
# HTTPD lab configuration (ug-ops, ug-security, ug-routes)
PORT=8080
DOCROOT=/www
REALM=MVS.EXAMPLE
MOD=MVSMF /zosmf/info                   AUTH=NONE
MOD=MVSMF /zosmf/services/authenticate  AUTH=NONE
MOD=MVSMF /zosmf/restconsoles/*         AUTH=TOKEN RES=FACILITY:MVSMF.CONSOLE
```

```
MOD=MVSMF /zosmf/*                      AUTH=TOKEN
MOD=HTTPDSRV /.dsrv                     AUTH=FORM
MOD=HTTPDM /.dm                         AUTH=BASIC
MOD=HTTPDMTT /.dmtt                     AUTH=BASIC
LOC=/pub/*                              AUTH=NONE
LOC=/private/*                          AUTH=BASIC
LOC=/form/*                             AUTH=FORM
LOC=/token/*                            AUTH=TOKEN
LOC=/admin/*                            AUTH=BASIC RES=FACILITY:HTTPD.ADMIN
LOC=/denied/*                           AUTH=BASIC RES=FACILITY:HTTPD.DENIED
```

The routes of this member register seven programs and six locations. The server writes one **HTTPD036I** for each program route and one **HTTPD417I** for each location:

```
S HTTPD,M=HTTPPRM1
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.38.23
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/info
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/services/authenticate
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/restconsoles/*
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD036I MODULE HTTPDSRV REGISTERED FOR /.dsrv
HTTPD036I MODULE HTTPDM REGISTERED FOR /.dm
HTTPD036I MODULE HTTPDMTT REGISTERED FOR /.dmtt
HTTPD417I LOCATION /pub/* REGISTERED
HTTPD417I LOCATION /private/* REGISTERED
HTTPD417I LOCATION /form/* REGISTERED
HTTPD417I LOCATION /token/* REGISTERED
HTTPD417I LOCATION /admin/* REGISTERED
HTTPD417I LOCATION /denied/* REGISTERED
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CD6E8) TASK(1A6FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0FE748)   TCB(9BBA58) TASK(1BAFC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0FE648)   TCB(9BB328) TASK(1CBFC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0FE5C8)   TCB(9BAEB8) TASK(1DCFC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

Count the routes in the messages and compare the count with your Parmlib member. A route that is missing from the messages is a route that the server does not have.

A start that is refused

A second HTTPD must not run on the port of a first one. HTTPD takes an exclusive lock with the name of the port when it starts. When the lock is held by another HTTPD, the new one writes **HTTPD037E**, stops, and the job step ends with return code 8. The member in this example has the port 8081, and another HTTPD was already running on that port:

```
PORT=8081
BIND_TRIES=2
BIND_SLEEP=2
DOCROOT=/www
MOD=MVSMF /zosmf/*  AUTH=TOKEN
```

```
S HTTPD,M=PRMPORT
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.31.14
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD037E HTTPD IS ALREADY ACTIVE ON PORT 8081, THIS INSTANCE ENDS
HTTPD098I HTTPD SHUTTING DOWN
HTTPD416I STATS: 0 REQUESTS, 0 ERRORS, 0 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
IEF404I HTTPD - ENDED - TIME=05.31.14
$HASP395 HTTPD    ENDED
```

The three lines **HTTPD098I**, **HTTPD416I** and **HTTPD099I** are the lines of a normal stop (“Stopping” on page 82). The new server has read the member and registered the route (**HTTPD036I**) before it found the

port taken, but it never listens. The first server is not touched. Look at the running one with **D P** (“The Operator Commands” on page 84) and decide: stop it, or give the new member another **PORT**.

The lock stops only a second HTTPD. A different program that holds the port is found later, when HTTPD binds the port. “Restarting” on page 84 shows what the server does in that case.

A missing Parmlib member is not a refusal

The most dangerous start is the one that looks like a good one. A member name that does not exist, as in **S HTTPD,M=NOSUCH**, does not stop the server. HTTPD writes a warning, runs on its built-in defaults, and writes **HTTPD001I READY**:

```
S HTTPD,M=NOSUCH
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.48.26
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD020W CANNOT OPEN DD:HTTPPRM -- USING DEFAULTS
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

Look at the capture. There is no **HTTPD036I** and no **HTTPD417I**: the server has the port 8080 and no routes. It runs, but it is not the server that the member would have given you. Treat **HTTPD020W** as a failed start: stop the server, correct the member name or the library, and start again. A line in the member that is not of the form **KEYWORD=VALUE** gives **HTTPD026W** in the same way and the server goes on (Chapter 4, “Configuring HTTPD” on page 33).

Stopping

Stop HTTPD with the console command **P**:

```
P HTTPD
```

HTTPD stops accepting connections, waits for its socket thread, ends the workers and gives back what it holds. On the console this is:

```
P HTTPD
HTTPD098I HTTPD SHUTTING DOWN
HTTPD060I SHUTDOWN SOCKET THREAD    TCB(9CD7C8) TASK(14FFC8) STACKSIZE(32768)
HTTPD060I SHUTDOWN WORKER(161B80)   TCB(9BA858) TASK(185FC8) STACKSIZE(65536)
HTTPD060I SHUTDOWN WORKER(161C00)   TCB(9BAB88) TASK(174FC8) STACKSIZE(65536)
HTTPD060I SHUTDOWN WORKER(161D00)   TCB(9BAEB8) TASK(163FC8) STACKSIZE(65536)
HTTPD416I STATS: 30 REQUESTS, 12 ERRORS, 21694 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
$HASP395 HTTPD    ENDED
IEF404I HTTPD - ENDED - TIME=00.42.25
```

The steps are these.

1. **HTTPD098I** starts the stop.
2. HTTPD waits up to 10 seconds for the socket thread to end. From the second second on it writes **HTTPD040I WAITING FOR SOCKET THREAD TO TERMINATE (n)**. If the thread is still there after 10 seconds, **HTTPD041I** says that HTTPD detaches it by force.
3. The listener is closed and every worker ends. Each thread writes one **HTTPD060I**. The count is the socket thread and the workers that were running.
4. The modules that the server held in storage are released, and the clients and the routes are freed.

5. **HTTPD416I** shows the requests, errors and bytes of the run. **HTTPD099I** ends the sequence, and the job ends.

A normal stop ends with return code 0. If the shutdown itself fails, the server writes **HTTPD912E** and the code of the failure. There is no second way to stop HTTPD from the console, and no **MODIFY** command stops it. The comment lines of the procedure that ships with HTTPD name a command **F HTTPD,SHUTDOWN**. The server does not have it. This is what it answers:

```
F HTTPD,SHUTDOWN
HTTPD100I CONS(0) "SHUTDOWN"
HTTPD101E UNKNOWN CONSOLE COMMAND "SHUTDOWN"
HTTPD150I MODIFY COMMANDS ARE:
HTTPD151I DISPLAY Config (D C)
HTTPD151I     DISPLAYS THE CONFIGURATION THIS SERVER IS RUNNING
HTTPD151I DISPLAY Login (D L)
HTTPD151I     DISPLAYS USERS THAT ARE LOGGED INTO THE HTTPD SERVER
HTTPD151I DISPLAY Memory xxxxxx~,nnn~ (D M xxxxxx)
HTTPD151I     DISPLAYS MEMORY AT THE GIVEN ADDRESS xxxxxx
HTTPD151I DISPLAY Ports (D P)
HTTPD151I     DISPLAYS THE PORT NUMBER THIS SERVER IS LISTENING ON
HTTPD151I DISPLAY Stats (D S)
HTTPD151I     DISPLAYS HTTPD CLIENT STATISTICS
HTTPD151I DISPLAY Threads (D T)
HTTPD151I     DISPLAYS INFORMATION ABOUT THE SERVER THREADS
HTTPD151I DISPLAY TIme ~-|+~~minutes~ (D TI)
HTTPD151I     DISPLAYS THE CURRENT TIME IN GMT AND LOCAL TIME
HTTPD151I DISPLAY Version (D V)
HTTPD151I     DISPLAYS THE SERVER VERSION AND BUILD
HTTPD151I
HTTPD151I SET MAxtask n (S MA n)
HTTPD151I     SETS THE WORKER MAXIMUM THREAD COUNT
HTTPD151I SET MIntask n (S MI n)
HTTPD151I     SETS THE WORKER MINIMUM THREAD COUNT
HTTPD151I SET Stats NONE|ERROR|AUTH|ALL ~RESET~
HTTPD151I     SETS THE SMF RECORDING LEVEL
HTTPD151I
HTTPD151I EXAMPLE: /F HTTPD,D C
```

A request that is in progress when the server stops is not guaranteed to complete. The sends of the server look at the stop flags, and a module that runs for a long time has to look at them itself (“Long-Running Handlers” on page 107). Before you stop the server, check with **D T** that no worker serves a client you care about (“The threads” on page 87).

Cancelling

P is the way to stop HTTPD (“Stopping” on page 82). The console command **C** ends the address space at once, whether the server is idle or serving. Cancel a server only when it does not end after **P**:

```
C HTTPD
IEE301I HTTPD    CANCEL COMMAND ACCEPTED
IEF450I HTTPD HTTPD - ABEND S222 U0000 - TIME=12.07.14
$HASP395 HTTPD    ENDED
```

The system ends the task, not HTTPD. **IEF450I** shows the abend code S222, the code of a cancel by the operator. The shutdown of the server does not run. The console shows none of **HTTPD098I**, **HTTPD060I**, **HTTPD416I** and **HTTPD099I**, so you get no counters for the run. An idle server and a server with clients connected give the same lines.

When the job has ended, start the server as after a stop (“Restarting” on page 84).

Lines from libc370. HTTPD 4.3.0 and 4.4.0 wrote lines that start with **libc370 recovery:** and **libc370 @@@try.c failed()** during a cancel, several lines, more when requests were in progress. They said only that a task was ending, which **IEF450I** says as well. HTTPD 4.5.0 is built with libc370 2.26.0, and the server writes none of them.

Note: A server module carries its own copy of libc370. A module that was linked with a level older than 2.26.0 can still write **libc370 @@@try.c failed(): recovery cleanup-only, retry suppressed** when a request is inside it at the cancel. MVSMF 1.3.0, which was linked with libc370 2.12.1, writes

one such line in front of **IEF450I**. The line is harmless. It goes away when the module is built with libc370 2.26.0 or later.

The quiet cancel changes only the line for a task that ends because of the cancel. An abend of a request is reported as before (“Worker Abends” on page 94).

Restarting

There is no restart command. Restart is a stop and a start:

```
P HTTPD
S HTTPD
```

Wait for **HTTPD099I** before you start. Restart the server after

- a change of the Parmlib member, because it is read only at the start;
- a new level of a server module in the load library, because HTTPD holds the modules in storage (“Deploying and Routing” on page 105);
- a new level of HTTPD itself.

Do not mix this up with the three commands that change a setting while the server runs (“Changing the number of workers and the SMF level” on page 88). They cannot do the work of a restart.

After a quick restart the TCP/IP stack may not have released the port yet. The new server then cannot bind it. It writes **HTTPD030E** with the codes of the failure and, when the reason is that the address is in use, it does not give up at once. It writes

```
HTTPD030I EADDRINUSE, WAITING FOR TCPIP TO RELEASE HTTP PORT 8080
```

and tries again after a pause. The number of tries and the pause are the Parmlib keywords **BIND_TRIES** (default 10) and **BIND_SLEEP** (default 10 seconds). If the bind still fails after the last try, the server writes **HTTPD030E** again and the start ends. The Reference chapter “Messages” lists the codes. Wait for the line **HTTPD001I** before you tell the users that the server is back.

The Operator Commands

A running HTTPD takes commands with the **MODIFY** command of the system:

```
F HTTPD,D C
```

The answers are messages on the console, with the prefix **HTTPD**. The answers never go to a web client. The commands are these.

Table 25. The commands for the operator

Command	Task
HELP, **?**	List the commands.
D C	Which configuration is the server running with?
D P	Which port does it listen on?
D V	Which version and build?
D TI [-|+][min]	What time does the server see?

D L	Who is logged in?
D T	What are the threads doing?
D M address[,len]	Dump storage of the server.
D S	How many requests, errors and bytes? How many connections are open?
SET MA n	Change the largest number of workers.
SET MI n	Change the smallest number of workers.
SET STATS level [RESET]	Change the SMF level, or set the counters to zero.

Some rules hold for all commands:

- HTTPD writes every command back as **HTTPD100I CONS(n) "text"**, also a command it does not know. The screens below show this line.
- The words are not case sensitive. A comma and a blank both separate words.
- A word matches a keyword when it is the beginning of the keyword, and the first match wins. **D T** is **DISPLAY THREADS**. The time is **D TI**. After the **F HTTPD,**, the **S** is **SET**, not the start command of the system, and **S MA** is **SET MAXTASK**. A bare **S M** is also **MAXTASK**. Use the forms of the table.
- The server queues at most 5 commands. Do not send a command faster than the answer arrives.

Help

F HTTPD,HELP and **F HTTPD,?** list the commands. The shortest form of a keyword shows in capitals. The brackets that mark an optional part show as tilde characters on the console, as in the capture below.

```
F HTTPD,?
HTTPD100I CONS(0) "?"
HTTPD150I MODIFY COMMANDS ARE:
HTTPD151I DISPLAY Config (D C)
HTTPD151I     DISPLAYS THE CONFIGURATION THIS SERVER IS RUNNING
HTTPD151I DISPLAY Login (D L)
HTTPD151I     DISPLAYS USERS THAT ARE LOGGED INTO THE HTTPD SERVER
HTTPD151I DISPLAY Memory xxxxxx~,nnn~ (D M xxxxxx)
HTTPD151I     DISPLAYS MEMORY AT THE GIVEN ADDRESS xxxxxx
HTTPD151I DISPLAY Ports (D P)
HTTPD151I     DISPLAYS THE PORT NUMBER THIS SERVER IS LISTENING ON
HTTPD151I DISPLAY Stats (D S)
HTTPD151I     DISPLAYS HTTPD CLIENT STATISTICS
HTTPD151I DISPLAY Threads (D T)
HTTPD151I     DISPLAYS INFORMATION ABOUT THE SERVER THREADS
HTTPD151I DISPLAY TIme ~-|+~~minutes~ (D TI)
HTTPD151I     DISPLAYS THE CURRENT TIME IN GMT AND LOCAL TIME
HTTPD151I DISPLAY Version (D V)
HTTPD151I     DISPLAYS THE SERVER VERSION AND BUILD
HTTPD151I
HTTPD151I SET MAxtask n (S MA n)
HTTPD151I     SETS THE WORKER MAXIMUM THREAD COUNT
HTTPD151I SET MIntask n (S MI n)
HTTPD151I     SETS THE WORKER MINIMUM THREAD COUNT
HTTPD151I SET Stats NONE|ERROR|AUTH|ALL ~RESET~
HTTPD151I     SETS THE SMF RECORDING LEVEL
HTTPD151I
HTTPD151I EXAMPLE: /F HTTPD,D C
```

The configuration, the port, the version and the time

D C shows the values that the server holds after it has read the member. Use it to see what the server made of the member, not what you meant it to do:

```
F HTTPD,D C
HTTPD100I CONS(0) "D C"
HTTPD128I PORT 8080  MINTASK 3  MAXTASK 9
HTTPD129I CLIENT_TIMEOUT 10  KEEPALIVE 5/100
HTTPD130I SESSION_TIMEOUT 30 MIN
HTTPD135I SESSION_MAXAGE 480 MIN
HTTPD131I DOCROOT /www
HTTPD134I CODEPAGE CP037
```

```
HTTPD136I REALM MVS.EXAMPLE
HTTPD132I SMF TYPE 243 LEVEL NONE
HTTPD133I CONFIG FROM DD:HTTPPRM
HTTPD137I RESIDENT YES  MODULES HELD 4
HTTPD138I RES_NOPROFILE ALLOW
```

Read the lines in this way.

Table 26. The lines of D C

Message	What it tells
HTTPD128I	The port and the numbers of workers. These are the values from the Parmlib member. **SET MA** and **SET MI** do not change them (“Changing the number of workers and the SMF level” on page 88).
HTTPD129I	The client timeout in seconds, and the keep-alive timeout in seconds and the number of requests on one connection.
HTTPD130I, **HTTPD135I**	The idle time and the maximum age of a login, in minutes (Chapter 6, “Authentication and Security” on page 55).
HTTPD131I	The document root.
HTTPD134I	The code page.
HTTPD136I	The realm of the Basic authentication.
HTTPD132I	The SMF record type and the level in effect now.
HTTPD133I	Where the configuration came from. **DD:HTTPPRM** means the member of the **HTTPPRM** DD of the procedure.
HTTPD137I	Whether the modules are held in storage, and how many different modules are held. The four modules in the screen above are MVSMF, HTTPDSRV, HTTPDM and HTTPDMTT; they have seven routes.
HTTPD138I	The value of **RES_NOPROFILE**, **ALLOW** or **DENY** (“Refusing a Route Whose Profile Is Missing” on page 66).

The command does not list the routes. The display module HTTPDSRV does (“The Display Modules over HTTP” on page 90).

D P shows the port, **D V** the version and the build, and **D TI** the time of the server in GMT and in local time. The offset is in minutes. With a number, as in **D TI -60**, the command shows the time for that offset. This changes only the display.

```
F HTTPD,D P
HTTPD100I CONS(0) "D P"
HTTPD102I HTTPD IS LISTENING ON PORT 8080
```

```
F HTTPD,D V
HTTPD100I CONS(2) "D V"
HTTPD140I HTTPD 4.5.1 (nnnnnnn)
```

```
F HTTPD,D TI
HTTPD100I CONS(0) "D TI"
HTTPD142I TIME 2026/10/27 05:44:39 GMT
HTTPD143I TIME 2026/10/27 00:44:39 LOCAL OFFSET=-300
```

Who is logged in

The server holds a login for a user until it expires (Chapter 6, “Authentication and Security” on page 55). The login has the security environment (the ACEE) of the user. **`D L`** shows how many logins there are, and for each one the user ID, the address of the client and the address of the ACEE:

```
F HTTPD,D L
HTTPD100I CONS(0) "D L"
HTTPD121I USERS LOGGED IN: 1
HTTPD122I USER IBMUSER  IP 192.0.2.17 ACEE(9BA1E0)
```

The threads

`D T` shows each thread of the server in a block of lines that ends with a line of dashes (**`HTTPD199I`**). The blocks come in this order: the main thread, the socket thread, the dispatcher and the workers. The screen shows an idle server with three workers:

```
F HTTPD,D T
HTTPD100I CONS(0) "D T"
HTTPD103I ..THREAD 000FBFC8 MAIN SERVER THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009DC328  ..OWNTCB 00000000
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00010030
HTTPD106I ....FUNC 0009987C  ....ARG1 00000003  ....ARG2 000DB0F0
HTTPD120I ...RNAME HTTPD.0007
HTTPD199I ------------------------------------------------------
HTTPD103I ..THREAD 001A0FC8 SOCKET THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009CD608  ..OWNTCB 009DC328
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00008000
HTTPD106I ....FUNC 00098430  ....ARG1 000DB390  ....ARG2 0009E2E8
HTTPD119I ..SOCKET        3  ....PORT     8080  (HTTP PROTOCOL)
HTTPD199I ------------------------------------------------------
HTTPD103I ..THREAD 001A9FC8 DISPATCHER THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009BAC18  ..OWNTCB 009DC328
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00008000
HTTPD106I ....FUNC 000A15DC  ....ARG1 000DAE18  ....ARG2 00000000
HTTPD107I .....EYE CTHDMGR   ....TASK 001A9FC8  .WAITECB 809CD330
HTTPD108I ....FUNC 00098F1C  ...UDATA 000DB390  STACKSZE 00010000
HTTPD109I ..WORKER 000FB6D4  ..QUEUE 000FB07C  ...STATE 00000004 WAITING
HTTPD110I .MINTASK        3  .MAXTASK        9
HTTPD111I .WORKERS        3  .DISPCNT 18
HTTPD199I ------------------------------------------------------
HTTPD103I ..THREAD 001B4FC8 WORKER THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009BAAA8  ..OWNTCB 009BAC18
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00010000
HTTPD106I ....FUNC 00098F1C  ....ARG1 000DB390  ....ARG2 000FB748
HTTPD112I ..WORKER 000FB748  .....EYE CTHDWORK  .WAITECB 809B9F98
HTTPD113I .....MGR 000DAE18  ....TASK 001B4FC8  ...QUEUE 00000000
HTTPD114I ...STATE 00000002 WAITING
HTTPD115I ...START 000000006AE039DF  Tue Oct 27 00:42:55 2026
HTTPD116I ....WAIT 000000006AE03A28  Tue Oct 27 00:44:08 2026
HTTPD117I ....DISP 000000006AE03A27  Tue Oct 27 00:44:07 2026
HTTPD141I .DISPCNT 6
HTTPD199I ------------------------------------------------------
HTTPD103I ..THREAD 001C5FC8 WORKER THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009BA378  ..OWNTCB 009BAC18
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00010000
HTTPD106I ....FUNC 00098F1C  ....ARG1 000DB390  ....ARG2 000FB648
HTTPD112I ..WORKER 000FB648  .....EYE CTHDWORK  .WAITECB 809CD498
HTTPD113I .....MGR 000DAE18  ....TASK 001C5FC8  ...QUEUE 00000000
HTTPD114I ...STATE 00000002 WAITING
HTTPD115I ...START 000000006AE039DF  Tue Oct 27 00:42:55 2026
HTTPD116I ....WAIT 000000006AE03A27  Tue Oct 27 00:44:07 2026
HTTPD117I ....DISP 000000006AE03A27  Tue Oct 27 00:44:07 2026
HTTPD141I .DISPCNT 6
HTTPD199I ------------------------------------------------------
HTTPD103I ..THREAD 001D6FC8 WORKER THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009BA048  ..OWNTCB 009BAC18
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00010000
HTTPD106I ....FUNC 00098F1C  ....ARG1 000DB390  ....ARG2 000FB5C8
HTTPD112I ..WORKER 000FB5C8  .....EYE CTHDWORK  .WAITECB 809BA538
HTTPD113I .....MGR 000DAE18  ....TASK 001D6FC8  ...QUEUE 00000000
HTTPD114I ...STATE 00000002 WAITING
HTTPD115I ...START 000000006AE039DF  Tue Oct 27 00:42:55 2026
HTTPD116I ....WAIT 000000006AE03A28  Tue Oct 27 00:44:08 2026
HTTPD117I ....DISP 000000006AE03A27  Tue Oct 27 00:44:07 2026
HTTPD141I .DISPCNT 6
HTTPD199I ------------------------------------------------------
```

In the dispatcher block, **`HTTPD110I`** shows the smallest and the largest number of workers that are in effect now, and **`HTTPD111I`** the number of workers and the number of requests the dispatcher has handed out (**`.DISPCNT`**). A worker block shows the state of the worker (**`WAITING`** when it has nothing to do) and three times: when the worker started, when it last waited, and when it last got a request. A

worker that serves a client has three more lines: the user, the address of the client and the socket of the connection (**HTTPD118I** to **HTTPD120I**). That is how you find a request that does not end (“Watching a Long-Running Server” on page 95).

Storage

D M address,length dumps storage of the server. The address is in hex and must be below 16 MB (hex 00FFFFFF), else the server answers with **HTTPD144E**. Without an address the answer is **HTTPD148I**, a line that shows the syntax. The length is a decimal number, or a hex number with the prefix **0x**. A number with a leading 0 is octal. The default is 256 bytes, and the largest dump is 4096. An address that the task cannot read ends the dump with **HTTPD147E**, and the server goes on.

```
F HTTPD,D M 10,16
HTTPD100I CONS(0) "D M 10,16"
Dump of 00000010 "DISPLAY MEMORY" (16 bytes)
+00000 0001D048 00000000 070E0000 00000000 :..}.............:
HTTPD145I END OF MEMORY DUMP FOR 0x000010
```

The dump itself is not a numbered message. **HTTPD145I** ends it. The command is a tool for a person who knows the control blocks. The same dump is easier to read, and has links that follow the pointers, in the display modules (“The Display Modules over HTTP” on page 90).

The statistics

D S shows the SMF level and four counters:

```
F HTTPD,D S
HTTPD100I CONS(0) "D S"
HTTPD123I SMF: NONE (TYPE 243)
HTTPD124I REQUESTS: 18  ERRORS: 3  BYTES: 186629  ACTIVE: 0
```

REQUESTS is the number of requests that the server has handled since the start, **ERRORS** the number of responses with a status of 400 or more, and **BYTES** the number of bytes it has sent. **ACTIVE** is the number of connections that are open now. The three counters count at every SMF level, also at NONE. The same three counters are in the line **HTTPD416I** at the stop.

Changing the number of workers and the SMF level

Three settings can change while the server runs.

The number of workers. **SET MA n** sets the largest number of worker threads, **SET MI n** the smallest. Give the number: without one, and with a negative number, HTTPD answers **HTTPD149I** and changes nothing. HTTPD takes the value 0 as 1 and a value above 100 as 100, and a word that is not a number as 1. The new value holds until the server stops. The Parmlib member does not change, and **D C** still shows the values from the member.

The answer is the dispatcher block of **D T**. It shows the new limits. The count of workers in it is the count from before the server reacted. The server then starts or ends workers, and each one writes **HTTPD061I** or **HTTPD060I**:

```
F HTTPD,SET MA 12
HTTPD100I CONS(0) "SET MA 12"
HTTPD107I .....EYE CTHDMGR   ....TASK 001A9FC8  .WAITECB 40000000
HTTPD108I ....FUNC 00098F1C  ...UDATA 000DB390  STACKSZE 00010000
HTTPD109I ..WORKER 000FB6D4  ...QUEUE 000FB07C  ...STATE 00000004 WAITING
HTTPD110I .MINTASK        3  .MAXTASK       12
HTTPD111I .WORKERS        3  .DISPCNT 18
HTTPD199I ------------------------------------------------------------
```

```
F HTTPD,SET MI 4
HTTPD100I CONS(0) "SET MI 4"
HTTPD107I .....EYE CTHDMGR   ....TASK 001A9FC8  .WAITECB 40000000
HTTPD108I ....FUNC 00098F1C  ...UDATA 000DB390  STACKSZE 00010000
```

```
HTTPD109I ..WORKER 000FB6D4  ...QUEUE 000FB07C  ...STATE 00000004 WAITING
HTTPD110I .MINTASK        4  .MAXTASK      100
HTTPD111I .WORKERS        3  .DISPCNT 18
HTTPD199I ------------------------------------------------------
HTTPD061I STARTING WORKER(0FB0F0)   TCB(9B9C70) TASK(1E7FC8) STACKSIZE(65536)
```

The second screen shows that a larger **MINTASK** starts the missing worker at once. The next screen shows the cut to 100. When MAXTASK is set below MINTASK, MINTASK follows it down, and when MINTASK is set above MAXTASK, MAXTASK follows it up. Every worker has a stack of 64 KB (the value **STACKSIZE(65536)** in **HTTPD061I**) and the region of the server is fixed in the procedure, so do not raise the limits without need.

```
F HTTPD,SET MA 99999
HTTPD100I CONS(0) "SET MA 99999"
HTTPD107I .....EYE CTHDMGR   ....TASK 001A9FC8  .WAITECB 40000000
HTTPD108I ....FUNC 00098F1C  ...UDATA 000DB390  STACKSZE 00010000
HTTPD109I ..WORKER 000FB6D4  ...QUEUE 000FB07C  ...STATE 00000004 WAITING
HTTPD110I .MINTASK        3  .MAXTASK      100
HTTPD111I .WORKERS        3  .DISPCNT 18
HTTPD199I ------------------------------------------------------
```

The SMF level. **SET STATS** with **NONE**, **ERROR**, **AUTH** or **ALL** sets the level of recording at once. Write the name in full. The word **RESET** after the level, or alone, sets the three counters to zero. The number of active connections is not set to zero.

```
F HTTPD,SET STATS ALL
HTTPD100I CONS(0) "SET STATS ALL"
HTTPD126I SMF LEVEL SET TO ALL
```

```
F HTTPD,SET STATS ALL RESET
HTTPD100I CONS(0) "SET STATS ALL RESET"
HTTPD126I SMF LEVEL SET TO ALL
HTTPD125I STATISTICS COUNTERS RESET
```

A level that the server does not know is an error, and it keeps the old level:

```
F HTTPD,SET STATS BOGUS
HTTPD100I CONS(0) "SET STATS BOGUS"
HTTPD126E INVALID SET STATS VALUE "BOGUS"
```

Commands that are not accepted

A word that HTTPD does not know as the first word gives **HTTPD101E** and the help text. Look at the first lines of the answer, the rest is the help:

```
F HTTPD,BOGUS
HTTPD100I CONS(0) "BOGUS"
HTTPD101E UNKNOWN CONSOLE COMMAND "BOGUS"
HTTPD150I MODIFY COMMANDS ARE:
HTTPD151I DISPLAY Config (D C)
HTTPD151I     DISPLAYS THE CONFIGURATION THIS SERVER IS RUNNING
HTTPD151I DISPLAY Login (D L)
HTTPD151I     DISPLAYS USERS THAT ARE LOGGED INTO THE HTTPD SERVER
HTTPD151I DISPLAY Memory xxxxxx~,nnn~ (D M xxxxxx)
HTTPD151I     DISPLAYS MEMORY AT THE GIVEN ADDRESS xxxxxx
HTTPD151I DISPLAY Ports (D P)
HTTPD151I     DISPLAYS THE PORT NUMBER THIS SERVER IS LISTENING ON
HTTPD151I DISPLAY Stats (D S)
HTTPD151I     DISPLAYS HTTPD CLIENT STATISTICS
HTTPD151I DISPLAY Threads (D T)
HTTPD151I     DISPLAYS INFORMATION ABOUT THE SERVER THREADS
HTTPD151I DISPLAY TIme ~-|+~~minutes~ (D TI)
HTTPD151I     DISPLAYS THE CURRENT TIME IN GMT AND LOCAL TIME
HTTPD151I DISPLAY Version (D V)
HTTPD151I     DISPLAYS THE SERVER VERSION AND BUILD
HTTPD151I
HTTPD151I SET MAxtask n (S MA n)
HTTPD151I     SETS THE WORKER MAXIMUM THREAD COUNT
HTTPD151I SET MIntask n (S MI n)
HTTPD151I     SETS THE WORKER MINIMUM THREAD COUNT
HTTPD151I SET Stats NONE|ERROR|AUTH|ALL ~RESET~
HTTPD151I     SETS THE SMF RECORDING LEVEL
```

```
HTTPD151I
HTTPD151I EXAMPLE: /F HTTPD,D C
```

A known first word with an unknown second word gives only the error:

```
F HTTPD,D BOGUS
HTTPD100I CONS(0) "D BOGUS"
HTTPD101E UNKNOWN CONSOLE COMMAND "DISPLAY BOGUS"
```

```
F HTTPD,SET BOGUS 1
HTTPD100I CONS(0) "SET BOGUS 1"
HTTPD101E UNKNOWN CONSOLE COMMAND "SET BOGUS"
```

`DISPLAY` or **`SET`** alone gets no answer except the echo. **`SET STATS`** without a level gives **`HTTPD127W`**, and **`SET MA`** or **`SET MI`** without a number gives **`HTTPD149I`**. Check that the echo **`HTTPD100I`** shows the command that you meant to send, and then that the answer is what you expect. A command that reaches the server and is not understood is not an abend, it is a mistyped command.

The Display Modules over HTTP

HTTPD comes with three small server modules that show the running server through HTTP. HTTPDSRV shows the control blocks of the server, such as the server block, the worker pool and the routes. HTTPDM shows any piece of storage as a dump. HTTPDMTT shows the Master Trace Table of the system. A fourth module, ABEND0C1, abends on purpose for a test of the recovery. The *HTTPD Reference* describes each one in the chapter "The Display Modules". This section shows how an operator uses them.

Setting up the routes

The server registers no route for the modules by itself. The recommended lines are the ones that the lab member above has:

```
MOD=HTTPDSRV /.dsrv     AUTH=FORM
MOD=HTTPDM   /.dm       AUTH=BASIC
MOD=HTTPDMTT /.dmtt     AUTH=BASIC
```

Three rules apply, and they are not optional.

- **Always write the AUTH=.** A route without it is public. A public **`/.dm`** is a window into the storage of the server, and a public **`/.dmtt`** is the console log of the system for everybody.
- **The modules do not look at the user.** Everyone who passes the gate of the route sees everything, including the console log and storage that may hold data of other users. To limit the route to some users, add a RACF resource to the line, for example **`RES=FACILITY:HTTPD.ADMIN`**. The line is described in Chapter 5, "Routes and Server Modules" on page 47 and Chapter 6, "Authentication and Security" on page 55.
- **Use the modules on a test system or for a short time.** They are tools for diagnosis, not endpoints for production. Take the routes out of the member, and restart, when the work is done.

The path of the route for HTTPDM must be exactly **`/.dm`**, because the pages of HTTPDSRV have links to that path.

What authentication they need

Every route takes every kind of credential that HTTPD knows: the cookie of a login, a bearer token, and a user ID and password in Basic form. The mode in the line, **`FORM`** or **`BASIC`**, says only what a client without a credential gets. **`FORM`** sends a browser to a login page. **`BASIC`** answers with the status 401 and the request to log in. For a tool such as **`curl`** this means that **`curl -u`** works against all three routes,

also against the route of HTTPDSRV, which has **AUTH=FORM**. Chapter 6, “Authentication and Security” on page 55 explains the modes.

A request without a credential is answered with the status 401 and the realm of the server (the **REALM** of the member, shown in **D C**):

```
$ curl -i "http://localhost:8080/.dm?m=10&l=16"
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:43:50 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

A user ID with a wrong password gets the same answer. The server does not say which of the two is wrong:

```
$ curl -i -u IBMUSER:wrong "http://localhost:8080/.dm?m=10&l=16"
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:43:50 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

The user ID must be known to the security product. The password is in the command line of **curl** and so in the process list and in the history of the shell. Give the user ID alone (**-u IBMUSER**) and **curl** asks for the password. The Basic form sends the user ID and the password with every request, encoded but not encrypted.

HTTPDM, a dump of storage

The parameters of HTTPDM are the address (**m**), the length (**l**), the size of a line (**c**) and the output form (**d**). The address is in hex, the length is decimal, and the longest dump is 4096 bytes. The default output is an HTML page where every group of four bytes is a link to the storage that the four bytes point to. With **d=1** the answer is plain text. The parameter **t** is a title that goes unchanged into the first line, so do not put text from other people into it.

The first example reads 16 bytes at the address 10 (hex), which is 16 in decimal. The first word there is the pointer to the CVT:

```
$ curl -s -u IBMUSER:****** "http://localhost:8080/.dm?m=10&l=16&t=CVTPTR" | sed -n "/<pre>/,/<\/pre>/p" | sed -e "s/<[^>]*>//g" | tr -cd
"\n -~"

Dump of CVTPTR 00000010 (16 byte chunks) (16 bytes total)
+00000 0001D048 00000000 078D2000 000D1152 :..}.............:
```

The second example follows the pointer. The address 0001D048 is the first word of the first dump. It reads 64 bytes in lines of 16 bytes, and the parameter **t** puts the name CVT in the first line:

```
$ curl -s -u IBMUSER:****** "http://localhost:8080/.dm?m=0001D048&l=64&c=16&t=CVT" | sed -n "/<pre>/,/<\/pre>/p" | sed -e "s/<[^>]*>//g"
| tr -cd "\n -~"

Dump of CVT 0001D048 (16 byte chunks) (64 bytes total)
+00000 00000218 00043C28 0001CD04 00000000 :.................:
+00010 00000000 00034338 00043D64 00038EE6 :..............W:
+00020 00038E60 00006490 000064E6 00044014 :...-.......W.. .:
+00030 000019D8 00044360 0126300F 0001C090 :...Q...-......{.:
```

The two commands in the captures cut the answer down to the dump with **sed**, because the page has a form around it. In a browser you get the whole page and follow the links.

HTTPDSRV, the control blocks

HTTPDSRV is for the browser. The parameter **target** says what to show: **HTTPD** (the default), **MGR**, **MOD**, **WORKERS**, **TASK**, **FILE**, **FS** and **TRACE**. Some of them need an address in the parameter **m**. The other targets answer with HTML pages. They give the offset, the name, the meaning and the value of each field. The target **TRACE** answers in plain text, one line for each request ("The Request Trace" on page 40). The *HTTPD Reference* lists the targets. The help page of HTTPDSRV lists more targets than the module has, and says that some need an address when they do not. The Reference has the correct list, and this guide uses only the targets of the Reference.

The server block shows the state and the counters of the whole server. The flag byte (offset +002C) is the state, the counters of requests, errors and bytes sent and the number of active connections are the same as in **D S**, and the up time is the time of the start. The screen is cut after the counters:

```
$ curl -s -u IBMUSER:****** "http://localhost:8080/.dsrv?target=HTTPD" | sed -n "/<table>/,/<\/table>/p" | sed -e "s/<\/t[dh]><t[dh]>/  /
g" -e "s/<[^>]*>//g" | tr -cd "\n -~"

Offset  Data Name  Description  Contents
+0000  httpd->eye  Eye Catcher  "*HTTPD*"
+0008  httpd->httpx  Function Pointers for CGI modules  0009DAA8
+000C  httpd->httpc  HTTP Client Array (0)  00000000
+0010  httpd->addr  HTTP Address  0.0.0.0
+0014  httpd->port  HTTP Port  8080
+0018  httpd->listen  HTTP Listen Socket  1
+001C  httpd->cfg_realm  Basic realm / server name  MVS.EXAMPLE
+0020  httpd->dbg  HTTP Debug File Handle  00000000
+0024  httpd->tzoffset  Time Zone Offset (seconds)  -18000
+0028  httpd->busy  HTTP Busy Client Array (1)  001B8D54
+002C  httpd->flag  Processing Flag  07 INIT LISTENER READY
+002D  httpd->unused_2d  Reserved (was: global LOGIN policy, retired)  00
+002E  httpd->client  Client Options  E0 TIMEOUT DUMP STATS
+002F  httpd->sockstate  socket thread: 0 starting, 1 running, 2 gone  01
+0030  httpd->socket_thread  Socket Thread Handle  001A6FC8
+0034  httpd->mgr  Thread Manager Handle  000DDE18
+0038  httpd->rname  Resource Name  "HTTPD.0007"
+0044  httpd->route  Route Array (MOD= programs, LOC= static prefixes)  000DD6D4
+0048  httpd->uptime  Server Up Time  Sat Oct 31 05:33:53 2026
+0050  httpd->trace  Request trace (NULL = off)  00000000
+0054  httpd->ufssys  UFS System Handle  000DDBB8
+0058  httpd->readerr_last  Time of Last UFS Read Error Message (#346)  0
+005C  httpd->version  Version String  "4.5.1"
+0060  httpd->readerr_held  UFS Read Errors Not Yet Reported (#346)  0
+0064  httpd->cfg_maxtask  Config Max Task  9
+0065  httpd->cfg_mintask  Config Min Task  3
+0066  httpd->cfg_client_timeout  Config Client Timeout Seconds  10
+0067  httpd->smf_level  SMF Recording Level  0 NONE
+0068  httpd->smf_type  SMF Record Type  243
+0069  httpd->cfg_resident  Config Hold MOD= Modules  YES
+006A  httpd->resident_held  Modules Held Resident  4
+006B  httpd->cfg_cgictx  Config CGI Context Pointers  255
+006C  httpd->ufs_enabled  UFS Filesystem Enabled  1
+006D  httpd->dbg_enabled  Debug Output Enabled  0
+006E  httpd->bind_tries  Socket Bind Retry Count  10
+006F  httpd->bind_sleep  Socket Bind Retry Delay (seconds)  10
+0070  httpd->total_requests  Total HTTP Requests  1
+0074  httpd->total_errors  Total Error Responses  0
+0078  httpd->total_bytes_sent  Total Bytes Sent  25440
+007C  httpd->active_connections  Active Connections  1
```

Further down the page, the row **httpd->sec_open** (offset +0122) says whether the security product accepted the test logon at the start. A healthy server shows **NO**. The value **YES, LOGIN REFUSED** goes with the message **HTTPD042W** ("A Security System That Accepts Any User ID" on page 62). You can read the row in a browser only through a route that lets you reach HTTPDSRV without a login, because a refused login also keeps you out of the display modules.

The block of the manager of the worker pool shows the limits that are in effect now and the number of requests that were handed out:

```
$ curl -s -u IBMUSER:****** "http://localhost:8080/.dsrv?target=MGR" | sed -n "/<table>/,/<\/table>/p" | sed -e "s/<\/t[dh]><t[dh]>/  /g"
-e "s/<[^>]*>//g" | tr -cd "\n -~"

Offset  Data Name  Description  Contents
+0000  mgr->eye  Eye Catcher  "CTHDMGR"
```

```
+0008  mgr->task  Thread Manager Task Handle  001A9FC8
+000C  mgr->wait  Wait For Work ECB  809CD330
+0010  mgr->func  Thread Function Address  00098F1C
+0014  mgr->udata  Thread User Data  000DB390
+0018  mgr->stacksize  Thread Stack Size  65536
+001C  mgr->worker  Worker Threads Array  000FB6D4
+0020  mgr->queue  Worker Queue Array  000FB07C
+0024  mgr->state  Current State  4 "WAITING"
+0028  mgr->mintask  Minimum Worker Threads  3
+002C  mgr->maxtask  Maximum Worker Threads  9
+0030  mgr->dispatched  Dispatched Worker Threads Count  8
+0038  mgr->start  Round Robin Start Index  2
+003C  mgr->rname  Resource Name String  "CTHDMGR.0007.000DAE18"
```

The route array of the server block (offset +0044) is the address for the target **`MOD`**. The page for it shows every route with its path, its program, its authentication mode and its RACF resource. This is the place to check that the server holds the routes that you wrote in the member. The screen is cut after the third route. The lab member has thirteen:

```
$ curl -s -u IBMUSER:****** "http://localhost:8080/.dsrv?target=MOD&m=000DA6D4" | sed -n "/<table>/,/<\/table>/p" | sed -e "s/<\/
t[dh]><t[dh]>/  /g" -e "s/<[^>]*>//g" | tr -cd "\n -~"

Offset  Data Name  Description  Contents
+0000  route->eye  Eye Catcher  "*ROUTE*"
+0008  route->wild  Is Wildcard  0
+0009  route->unused_09  Reserved (was: login required, retired)  0
+000A  route->len  Path Length  11
+000C  route->path  Path Name  "/zosmf/info"
+0010  route->pgm  Program Name (NULL = LOC static route)  "MVSMF"
+0014  route->auth  Auth Mode (AUTH=)  0 NONE (public -- AUTH=NONE or no AUTH=)
+0015  route->resattr  RACF Access Attribute  00 READ (assumed, RESATTR not set)
+0018  route->resclass  RACF Class (RES=, NULL = no resource gate)  "(none)"
+001C  route->resname  RACF Resource Name (RES=)  "(none)"


Offset  Data Name  Description  Contents
+0000  route->eye  Eye Catcher  "*ROUTE*"
+0008  route->wild  Is Wildcard  0
+0009  route->unused_09  Reserved (was: login required, retired)  0
+000A  route->len  Path Length  28
+000C  route->path  Path Name  "/zosmf/services/authenticate"
+0010  route->pgm  Program Name (NULL = LOC static route)  "MVSMF"
+0014  route->auth  Auth Mode (AUTH=)  0 NONE (public -- AUTH=NONE or no AUTH=)
+0015  route->resattr  RACF Access Attribute  00 READ (assumed, RESATTR not set)
+0018  route->resclass  RACF Class (RES=, NULL = no resource gate)  "(none)"
+001C  route->resname  RACF Resource Name (RES=)  "(none)"


Offset  Data Name  Description  Contents
+0000  route->eye  Eye Catcher  "*ROUTE*"
+0008  route->wild  Is Wildcard  1
+0009  route->unused_09  Reserved (was: login required, retired)  0
+000A  route->len  Path Length  21
+000C  route->path  Path Name  "/zosmf/restconsoles/*"
+0010  route->pgm  Program Name (NULL = LOC static route)  "MVSMF"
+0014  route->auth  Auth Mode (AUTH=)  3 TOKEN (bare 401, API route)
+0015  route->resattr  RACF Access Attribute  02 READ
+0018  route->resclass  RACF Class (RES=, NULL = no resource gate)  "FACILITY"
+001C  route->resname  RACF Resource Name (RES=)  "MVSMF.CONSOLE"
```

HTTPDMTT, the Master Trace Table

Every message that HTTPD writes to the console also goes into the Master Trace Table of the system. The table keeps the most recent messages of the console, and HTTPD writes only what an operator can act on. What went wrong with a client is in the response, not here. So the table is the diagnostic channel of the server, and HTTPDMTT shows it in a browser or with **`curl`**, without a TSO session. The module takes one parameter, **`d`**. With **`d=1`** the answer is plain text. This is the end of the table after the start of the lab server (the capture shows the last four lines of the answer):

```
$ curl -s -u IBMUSER:****** http://localhost:8080/.dmtt | sed -n "/<pre>/,/<\/pre>/p" | sed -e "s/<[^>]*>//g" | tail -n 4
FFFF  5.33.53 STC   nn  HTTPD061I STARTING WORKER(0FE648)   TCB(9BB328) TASK(1CBFC8) STACKSIZE(65536)
FFFF  5.33.53 STC   nn  HTTPD061I STARTING WORKER(0FE5C8)   TCB(9BAEB8) TASK(1DCFC8) STACKSIZE(65536)
FFFF  5.33.54 STC   nn  HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

Each line has some fields of the table entry, the time of the entry and the text of the message. The module does not select or search. Use the plain form with a tool of your own, for example **`grep`**.

The table holds only the recent messages, so a server that has run for a long time may have lost its start messages: read the table soon after a failure. It holds the messages of the whole system. It is as sensitive as the console, and the route needs the strongest authentication that you have (Chapter 6, “Authentication and Security” on page 55).

The module ABEND0C1 abends on purpose, to show that the recovery of the server works. Register it only on a test system (“Worker Abends”).

Worker Abends

HTTPD has two layers of recovery, and an abend in a request does not stop the server.

A module abends. A module runs in the task of a worker. If it abends, HTTPD frees the storage the module took from its request area, writes **`HTTPD908E EXTERNAL PROGRAM name failed with Sxxx ABEND`** to the console, and, if the module has not started a response, sends the client the status 503 with a text body. The worker goes on with the next request. If the free of the storage fails, **`HTTPD906W`** tells you. The module chapter says more about what a module has to do for itself (“When a Module Abends” on page 107).

The server code abends. If the abend is in the code of the server and not in a module, the worker writes **`HTTPD062E ABEND code IN WORKER(..) CLIENT(..) SOCKET(n)`**, closes the client, and goes on with the next request. If the worker cannot recover at all, the message is **`HTTPD063E`**, which says that there is no recovery and the client is dropped.

What to do: write down the message, in particular the name of the module and the code. A module that abends with every request is a defect of the module, and a module that abends now and then points to its input. The Master Trace Table keeps the message (“HTTPDMTT, the Master Trace Table” on page 93). Do not restart the server for one **`HTTPD908E`**: the worker lives on. A repeated **`HTTPD062E`** with the same code is different. It points to the server itself, and Chapter 12, “Troubleshooting” on page 141 is the place to go on.

On a test system you can check the recovery. Register ABEND0C1 with **`AUTH=BASIC`** and call it a few times. Every call must give the status 503 and the text “External program ABEND0C1 failed with S0C1 ABEND”, and the other routes must go on working.

Recording with SMF

HTTPD can write an SMF record for each request, and for each connection at the highest level. This chapter gives only what the operator needs. The record layouts and the keywords are in the *HTTPD Reference*, in the chapter “SMF Records”.

The level is one of NONE (the default), ERROR (a record for each request with a status of 400 or more), AUTH (the requests with a status of 401 or 403, and the other errors) and ALL (every request and every connection). The default type is
243. The Parmlib keyword **`SMF`** sets the level at the start, and

`SET STATS` sets it while the server runs. A level that you set with the command lasts until the server stops. The member does not change, so the next start uses the level of the member.

You can see the level in two places: **`HTTPD132I`** in **`D C`** and **`HTTPD123I`** in **`D S`**. The counters of **`D S`** run at every level. They do not tell you whether records are written.

If the member asks for a level above NONE and SMF is not active in the system, the server writes **`HTTPD430W`** at the start and runs, but it writes no record. It checks for every record whether SMF is

active, so the records start when SMF becomes active. An SMF that does not record the type is a matter of the SMF parameters of your installation, which this guide does not cover.

Things to Know

- **Stop with P.** There is no **`F HTTPD,SHUTDOWN`**. The system answers it with **`HTTPD101E`**.
- **A good start ends with HTTPD001I.** A start without **`HTTPD036I`** lines that match the member is not a good start, even with **`HTTPD001I`** (“A missing Parmlib member is not a refusal” on page 82).
- **The member is read once.** The only changes at run time are **`SET MA`**, **`SET MI`** and **`SET STATS`**. They last until the stop.
- **A new module needs a restart.** HTTPD holds the modules in storage (“Deploying and Routing” on page 105).
- **The abbreviations take the first match.** Use **`D TI`** for the time and give the number to **`SET MA`** and **`SET MI`**.
- **Never register a display module without AUTH=.**

Watching a Long-Running Server

A server that runs for weeks needs few commands. The commands show what they show, and no more. There is no command that shows the free storage of the region, and none that keeps a history. Write the readings into your own log if you need one.

Is it up, and busy? **`D S`** shows the requests, the errors, the bytes and the active connections. Look at it at the same time each day, and keep the numbers. The difference between two readings is the work of the period. The number of **`ERRORS`** is the number of responses with a status of 400 or more. A client that sends a wrong password adds to it, so it is not the same as a fault of the server. After **`SET STATS ... RESET`** the counters start again from zero. The count of active connections is not reset.

Are the workers free? **`D T`** shows the state of each worker. At rest all workers are **`WAITING`**. A worker that has served the same client for a long time shows the client in the lines **`HTTPD118I`** to **`HTTPD120I`**. Compare the time of the last start of work (**`HTTPD117I`**) with the time now (**`D TI`**). If all workers up to MAXTASK are busy and the clients wait, raise **`MAXTASK`** with **`SET MA`** and look at the dispatcher block again. If one request is stuck in a module, the module is the cause, and more workers do not cure it (Chapter 12, “Troubleshooting” on page 141).

Memory. The region of the server is fixed in the procedure. The commands show the sizes of the thread stacks, and **`D C`** the number of modules that the server holds. They do not show how much storage is left. Abends of workers and modules (**`HTTPD062E`**, **`HTTPD908E`**) are in the Master Trace Table (“HTTPDMTT, the Master Trace Table” on page 93). Do not raise **`MAXTASK`** far above what the load needs.

Logins. **`D L`** shows the users who are logged in. A login ends after the idle time or the maximum age that **`D C`** shows in **`HTTPD130I`** and **`HTTPD135I`**. Many logins of one user from one address may mean a client that logs in again for every request (Chapter 6, “Authentication and Security” on page 55).

Time. **`D TI`** shows the GMT and the local time of the server. The offset is the one of the system. Compare the time with a clock you trust after a change of the time zone.

The long view. With SMF at ERROR, and SMF active and set to keep the record type, the failed requests can be read later from the SMF data (“Recording with SMF” on page 94). The Master Trace Table shows the messages of the last hours, no more.

Chapter 9. Writing a Server Module

A server module is a program that HTTPD runs to answer the requests for a path. HTTPD itself reads the request, checks the login, and finds the route. The module writes the response. mvsMF, HTTPLUA and HTTPREXX are server modules, and so are the display modules of Chapter 8, "Operating HTTPD" on page 79. This chapter takes you from nothing to a module that builds, runs under HTTPD and answers a request, and then it shows the rules that a module must keep. The exact list of functions, variables and messages is in the *HTTPD Reference*, in the chapter "The Server Module Interface". This chapter points to it where the detail is.

The program in this chapter is complete. It was built with MBT, deployed to a test system and called with **`curl`**. The screens are the real output.

What a Server Module Is

A server module is a load module in a library that the HTTPD started task can read. A route of the Parmlib names it with the keyword **`MOD=`** (Chapter 5, "Routes and Server Modules" on page 47). For every request to the route, the worker task that serves the request runs the module with the LINK service of MVS. The module runs in the address space of HTTPD, in the task of the worker, once for each request. The method does not matter: GET, POST, PUT, DELETE and HEAD all arrive at the same module, and the module has to look at the method itself.

Four facts follow from this, and the rest of the chapter depends on them.

- The server has done its work before the module starts. The request is read and parsed, and the authentication rule of the route has been applied. A request that fails the rule gets its 401 or 403 from the server and never reaches the module.
- The module must complete the response before it returns. When it returns, HTTPD ends the request, whatever the return code is.
- The module has no allocations of its own. A file that the module opens by ddname is looked for in the procedure of the started task, so the procedure is part of the contract of the module (see "Deploying and Routing" on page 105).
- A module that fails does not take the server down ("When a Module Abends" on page 107), but it can hold a worker for as long as it runs ("Long-Running Handlers" on page 107).

The Smallest Complete Module

The module HELLO answers three kinds of request. A GET of **`/hello`** answers with a greeting, and the query parameter **`name`** says whom to greet. A POST answers with the body it received. Any other method gets the status 405.

```
 1 /* HELLO - a small HTTPD server module.
 2  *
 3  *   GET  /hello?name=X   answers "Hello, X!" (X defaults to "world")
 4  *   POST /hello          answers with the request body it received
 5  *   anything else        405 with an Allow header
 6  */
 7 #include <mvs/crt.h>            /* __grtget(), CLIBGRT */
 8 #include "httpcgi.h"
 9
10 /* every http_xxx macro reaches the server through the HTTPX vector */
```

```
 11 #define httpx   http_get_httpx(httpd)
 12
 13 int
 14 main(int argc, char **argv)
 15 {
 16     CLIBGRT     *grt    = __grtget();
 17     HTTPD       *httpd  = grt->grtapp1;     /* the server               */
 18     HTTPC       *httpc  = grt->grtapp2;     /* the client being served  */
 19     UCHAR       *method = http_get_env(httpc, "REQUEST_METHOD");
 20     UCHAR       *text;
 21     (void)argc;
 22     (void)argv;
 23
 24     if (method && http_cmp(method, "GET") == 0) {
 25         text = http_get_env(httpc, "QUERY_name");
 26         http_resp(httpc, 200);
 27         http_printf(httpc, "Content-Type: text/plain\r\n");
 28         http_printf(httpc, "\r\n");     /* the blank line is a call of its own */
 29         http_printf(httpc, "Hello, %s!\n", text ? (char *)text : "world");
 30     }
 31     else if (method && http_cmp(method, "POST") == 0) {
 32         text = http_get_env(httpc, "POST_STRING");
 33         http_resp(httpc, 200);
 34         http_printf(httpc, "Content-Type: text/plain\r\n");
 35         http_printf(httpc, "\r\n");
 36         http_printf(httpc, "You sent: %s\n", text ? (char *)text : "");
 37     }
 38     else {
 39         http_resp(httpc, 405);
 40         http_printf(httpc, "Allow: GET, POST\r\n");
 41         http_printf(httpc, "Content-Type: text/plain\r\n");
 42         http_printf(httpc, "\r\n");
 43         http_printf(httpc, "405 Method Not Allowed\n");
 44     }
 45     return 0;
 46 }
```

The program is the whole module. The lines in it do the following.

Table 27. HELLO, line by line

Lines	What they do
7 and 8	**mvs/crt.h** of the C library declares **__grtget()** and the type **CLIBGRT**. **httpcgi.h** is the header of HTTPD for server modules.
11	Every function of the server is called through a vector of function pointers, the HTTPX. The macros of **httpcgi.h** expect a name **httpx** in scope. The line defines it as the expression that finds the vector in the server block, so the variable **httpd** must be in scope where a macro is used.
13 and 14	The entry point is **main(argc, argv)** (see "The Entry Point" on page 99).
16 to 18	The module finds the server block and the block of the client in the global runtime table: **grtapp1** holds HTTPD and **grtapp2** holds HTTPC.
19	**http_get_env** reads a variable of the request ("Reading the Request" on page 99).
21 and 22	The parameters are not used. The build turns every warning into an error ("Building" on page 103), so the module casts them to void.
24 to 30	GET. **http_resp(httpc, 200)** sends the status line and the standard headers. Line 27 sends one header. Line 28 ends the headers. Line 29 is the body.
31 to 37	POST. The body of the request is the variable **POST_STRING**. The module sends it back.
38 to 44	Any other method. The status 405 and the header **Allow** say what the module does accept.

45	The return code 0.

The listing shows the rules that every module must keep. They have their own sections: how the module is entered, how the headers end, and what the module does with storage.

Note: The listing is complete as a working program, but it leaves out one line that a module you keep should have: the size of the stack. HELLO does not set it, so each request that it serves asks for the default stack of 256 KB. The captures in this chapter show that it works. See “Storage” on page 103 for the line to add.

The Entry Point

The entry point of a server module is the C function **`main(argc, argv)`**. It is not a function with the server blocks as parameters. The module is linked with the startup code **`cgistart`** of the HTTPD library, and the startup code, not the module, reads the parameter list that HTTPD builds for the LINK. It stores the addresses of HTTPD and HTTPC in the global runtime table, opens **`stdout`**, **`stderr`** and **`stdin`** on the DDs **`HTTPDOUT`**, **`HTTPDERR`** and **`HTTPDIN`**, and calls **`main`**.

Note: Older notes on server modules describe an entry **`cgimain(HTTPD *, HTTPC *)`**. There is no such entry. The program in this chapter is the form that works: a plain **`main`**, and the two blocks fetched from the runtime table.

In **`main`** the module finds:

`argc`	2.
`argv[0]`	The name of the module, up to 8 characters.
`argv[1]`	The decoded request path, without quotes.
HTTPD, HTTPC	Read **`grt->grtapp1`** and **`grt->grtapp2`**, as the program does. Compare the eye catcher of each block (**`HTTPD_EYE`** and **`HTTPC_EYE`**) before you use it.
The vector	**`http_get_httpx(httpd)`** returns the HTTPX. In the program the macro **`httpx`** is defined with it.

The return code of **`main`** does not decide the answer. A value of 0 or more is not an error, and the startup code treats a negative value as 0.

If somebody runs the module outside HTTPD, the startup code writes “is an HTTPD server module and cannot run on its own” and ends.

`printf` is not what you expect in a server module. The startup code has a **`printf`** of its own that replaces the library function, and under HTTPD it writes into the response, as **`http_printv`** does. If you need a line in the output of the started task, write it with **`fprintf(stdout, ...)`** or **`fprintf(stderr, ...)`**. The first goes to the DD **`HTTPDOUT`**, the second to **`HTTPDERR`**.

Reading the Request

HTTPD stores the request in variables of the client, and the module reads them with **`http_get_env`**. The value comes back as an EBCDIC string, or NULL if there is no such variable. The names are not case sensitive. These are the variables a module uses most. The *HTTPD Reference* has the full list in the chapter “The Server Module Interface”.

Table 28. The request variables of most use

Variable	Value

REQUEST_METHOD	The method: GET, HEAD, PUT, POST or DELETE.
REQUEST_PATH	The path of the URI, decoded, without the query.
QUERY_STRING	The text after the question mark, not decoded. It exists only if the URI has a query.
QUERY_name	One variable for each query parameter. The value is decoded, the name is not. HELLO reads **QUERY_name** for the parameter **name**.
HTTP_name	One variable for each header of the request, for example **HTTP_Content-Type**.
POST_STRING	The body of a POST request.
POST_name	One variable for each field of a form body, decoded.

Test the result for NULL before you use it. HELLO does this for the parameter **name** and for the body.

HTTPD reads the body of a POST only under conditions. The header **Content-Length** must be present with a length from 1 to 3999, and there must be a **Content-Type** header. Then the whole body is in **POST_STRING**, raw and not decoded. If the type is exactly **application/x-www-form-urlencoded**, the server also splits the body into the **POST_name** variables and decodes their values. A body of 4000 bytes or more is not read, and the module runs without **POST_STRING**. The module reads that body from the socket in **httpc->socket**, as it does for a PUT. The connection closes after the response. On a route without a program, such a body gets the status 413. A chunked body, or a body without a **Content-Type**, gives no **POST_STRING**. The server does not read the body of a PUT: the module reads it from the socket in **httpc->socket**. A request with more than 256 parameters gets the status 400.

HELLO shows the difference between the variables. The first POST is a form, as **curl -d** sends it. **POST_STRING** holds the raw text, with the percent codes:

```
$ curl -i -d "text=hello%20from%20curl" http://localhost:8080/hello
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:57:59 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Content-Type: text/plain
Transfer-Encoding: chunked

You sent: text=hello%20from%20curl
```

The second POST has the type **text/plain**. The body arrives in the same variable:

```
$ curl -i -H "Content-Type: text/plain" --data-raw "plain body" http://localhost:8080/hello
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:57:59 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Content-Type: text/plain
Transfer-Encoding: chunked

You sent: plain body
```

Sending the Response

A response has four parts, and a module sends them in this order.

1. **http_resp(httpc, code)** sends the status line and the standard headers: **Date**, **Server**, **X-Powered-By** and **Connection**. The line has the version of the request.
2. Zero or more header lines, each with **http_printf**.
3. One call that ends the headers. The text of the call is the two characters carriage return and line feed, and nothing else.

4. The body, with **`http_printf`** or, for bytes that are not text, **`http_send`**.

`http_resp` knows a list of status codes: 200, 201, 202, 204, 206, 301, 302, 303, 304, 400, 401, 403, 404, 405, 409, 410, 412, 413, 414, 429, 500, 501, 502, 503, 505 and 507. For any other code it writes **`HTTPD909E`** and sends
500. **`http_printf`** formats its text in a buffer of 5120 bytes, converts it

to ASCII and sends it. Text that is longer than 5119 bytes is cut, so send a long body in pieces. It returns 0, or −1 if the send failed.

Here is what HELLO sends for a GET. The header lines are those of the module and of **`http_resp`**. The answer is chunked, because the module ended the headers in the right way:

```
$ curl -i http://localhost:8080/hello
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:57:59 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
Transfer-Encoding: chunked

Hello, world!
```

The parameter reaches the module decoded:

```
$ curl -i "http://localhost:8080/hello?name=IBMUSER"
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:57:59 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
Transfer-Encoding: chunked

Hello, IBMUSER!
```

A method that the module does not take gets the status 405 and the header that says what is allowed. The answer to a PUT:

```
$ curl -i -X PUT -d x http://localhost:8080/hello
HTTP/1.1 405 Method Not Allowed
Date: Tue, 27 Oct 2026 05:57:59 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Allow: GET, POST
Content-Type: text/plain
Transfer-Encoding: chunked

405 Method Not Allowed
```

The End of the Headers

This is the rule that most often goes wrong. Line 28 of HELLO is a call of its own:

```
   http_printf(httpc, "Content-Type: text/plain\r\n");
   http_printf(httpc, "\r\n");     /* the blank line is a call of its own */
```

HTTPD watches for the call whose text is exactly the two characters carriage return and line feed. That call ends the headers. When the server sees it, and the request is HTTP/1.1, and the status allows a body, and no **`Content-Length`** has been sent, it sends the header **`Transfer-Encoding: chunked`** and turns chunking on. From then on every send is a chunk, and the server sends the last, empty chunk when the request ends. The client knows where the body ends.

The version that looks shorter does not work:

```
   http_printf(httpc, "Content-Type: text/plain\r\n\r\n");   /* WRONG */
```

HTTPD does not recognize the end of the headers in such a call. The headers go out, the blank line goes out, but the response has neither a chunk framing nor a length. The next capture is from HELLO with that one change in each of its three answers. The answer has no **`Transfer-Encoding`**, and the last line of the headers is empty:

```
$ curl -sv -o /dev/null http://localhost:8080/hello 2>&1 | grep "^< "
< HTTP/1.1 200 OK
< Date: Tue, 27 Oct 2026 05:56:59 GMT
< Server: HTTPD Server
< X-Powered-By: MVSLOVERS
< Connection: keep-alive
< Content-Type: text/plain
<
```

With no length and no chunk framing, the client has no way to know where the body ends, so it waits for the connection to close. The measured times show what that costs. With the correct code, one request takes:

```
$ curl -s -o /dev/null -w "%{http_code} %{time_total}s\n" http://localhost:8080/hello
200 0.069525s
```

The same request to the module with the wrong terminator takes:

```
$ curl -s -o /dev/null -w "%{http_code} %{time_total}s\n" http://localhost:8080/hello
200 4.632924s
```

The measured time is close to the keep-alive timeout of the server, which is 5 seconds in the default setting (**`D C`** shows it as **`KEEPALIVE 5/100`**, see “The configuration, the port, the version and the time” on page 85). The capture does not prove why the connection ends when it does. What it shows is the symptom: the answer is right, the call is slow, and the delay is a number of seconds. If you see an answer that arrives after a fixed delay, look at the end of the headers first.

The correct form lets two requests share one connection. The second request of this capture uses the connection of the first:

```
$ curl -sv -o /dev/null -o /dev/null http://localhost:8080/hello http://localhost:8080/hello?name=again 2>&1 | grep -E "Re-using|^< HTTP|
^< Transfer|Connection"
< HTTP/1.1 200 OK
< Connection: keep-alive
< Transfer-Encoding: chunked
* Connection #0 to host localhost left intact
* Re-using existing http: connection with host localhost
< HTTP/1.1 200 OK
< Connection: keep-alive
< Transfer-Encoding: chunked
* Connection #0 to host localhost left intact
```

A request that has a body does not keep the connection. The POST and PUT captures above show **`Connection: close`**. The Reference says why: HTTPD turns keep-alive off for a request with a **`Content-Length`** or a **`Transfer-Encoding`** header, because it cannot be sure that it has read the whole body. This is not a fault of the module.

Three more rules go with the end of the headers.

- Do not send a line break alone as a call of its own before the headers have ended. The call with exactly carriage return and line feed is taken for the end of the headers. After that, a line break alone in the body is data. Put the line break in a longer text anyway.
- If the module sends a header line with **`Content-Length:`** through **`http_printf`**, HTTPD sees it and does not chunk. The module then must send exactly that number of bytes. A header line that goes out with **`http_send`** is not examined.
- Chunking is for HTTP/1.1. A client that sends HTTP/1.0 needs a **`Content-Length`**, and a module that serves such clients sets one.

- A HEAD request has no body in the answer. While the module runs, **`httpc->state`** is **`CSTATE_HEAD`**. After the call that ends the headers, HTTPD drops what **`http_printf`** writes, and it does not chunk the response. A module needs no special case to stay correct, but it can skip the body for a HEAD. Bytes that a module sends with **`http_send`** are not dropped. The *HTTPD Reference* has the rule, in the chapter "The Server Module Interface", section "Output".

Storage

A module runs in the task of a worker of HTTPD, and the rules for its storage follow from that.

Table 29. The storage rules of a module

Subject	Rule
Heap	While **`main`** runs, the startup code points the heap of the module at the request subpool, subpool 5. What the module gets with **`malloc`** belongs to the request. The module still frees what it allocates, on every path. If the module abends, the server frees the whole subpool.
Longer life	Storage that must outlive the request comes from **`__getmsp(size, 0)`** or from the block of **`http_cgictx_get`**. Storage in any other subpool does not survive the request.
No writable statics	A module must not write to a variable at file scope or to a static variable of a function. A module that is fetched from an APF-authorized library or from the link list is in storage of key 0, while its code runs in key 8. The first write to the storage of the module then abends with S0C4. The library decides this, not the attribute of the module.
Where to keep state	In local variables, on the heap, or in the memory that **`__wsaget()`** returns. State of the server belongs in the block of **`http_cgictx_get`**. A table or a string that the module only reads may be static.
Strings you change	Declare **`UCHAR text[] = "..."`** and not **`UCHAR *text = "..."`** if the text is translated or changed in place. The array is a copy on the stack. The pointer points to the constant itself, and a write there is the same S0C4.
Stack	The default of the C runtime asks for 256 KB of contiguous storage in subpool 0 for the stack of every request. A module sets a smaller stack in its own source file, as shown below.
Server allocations	The functions **`http_new_env`**, **`http_open`**, **`http_get_ufs`** and **`http_add_route`** take their storage from subpool 0, inside the server. The reclaim of subpool 5 does not touch it.

A writable static is the commonest mistake. Keep it out from the start. HELLO has no static at all.

The stack is the place where HELLO, as printed, is not what a module should be. Put this line into the source file of the module, not into a library:

```
unsigned __stklen = 64*1024;
```

Building

A module is built on a workstation with the cc370 cross compiler and MBT. It is not compiled on the MVS system. The project file of HELLO is this:

```
schema = 3

[project]
name = "hello"
version = "1.0.0"
kind = "application"

[toolchain]
mbt = "3.0"
cc370 = "1.6.3"
libc370 = "2.25.1"

[dependencies]
"mvslovers/httpd" = ">=4.2.0"

[module.HELLO]
dep_startup = true
rent = true
reus = true
sources = ["src/hello.c"]

[deploy]
target = "HTTPD.DEV.LINKLIB"
```

The file says the following.

Table 30. The keys of the project file

Key	Meaning
[toolchain]	The versions of MBT, of the compiler (cc370) and of the C library (libc370) that the project is built with.
[dependencies]	The product HTTPD, at 4.2.0 or later. MBT fetches the headers and the library of HTTPD from a release. The range names the HTTPD level that the module needs. Use the lowest level that has all the functions you call.
[module.HELLO]	A load module with the name HELLO. The name is the member name in the load library and the name in the **MOD=** keyword.
dep_startup = true	Link the module with the startup code **cgistart** of the HTTPD library, in place of the startup code of the C runtime. Without it the module has no **grtapp1** and no **grtapp2**.
rent, **reus**	Mark the module reentrant and reusable. HTTPD holds a module in storage only if it has one of the two (see “Deploying and Routing” on page 105), and several workers may run it at the same time.
sources	The source files.
[deploy]	The load library that **mbt deploy** writes to.

The headers come from the dependency. The command **mbt deps** resolves the range, picks a release, and stages it into the directory **.mbt/deps/httpd**, with the headers in **include** and the library in **lib**. The following **mbt build** passes that directory to the compiler with **-I** and links **.mbt/deps/httpd/lib/httpd.a**. The log shows both steps:

```
$ mbt deps
[mbt] mvslovers/httpd >=4.2.0 -> 4.4.0
[mbt]   staged -> .mbt/deps/httpd (sha 3528a9744617)
[mbt] Locked 1 dependency(ies) -> mbt.lock
$ mbt build -v
[mbt] libc370 2.25.1 in $HOME/.local/cc370 (>= 2.25.1)
cc370 -O1 -std=gnu99 -Wall -Wextra -Werror -I .mbt/deps/httpd/include -I .mbt/include -I .mbt -MMD -MP -c src/hello.c -o build/hello.o
ld370 -L$HOME/.local/cc370/lib -e @@CRT0 build/hello.o -lcc370rt .mbt/deps/httpd/lib/httpd.a -lc --rent --reus -iebcopy -o build/HELLO
[mbt] Build complete
```

Note the options of the compiler: **`-std=gnu99 -Wall -Wextra -Werror`**. A warning is an error. The result is the file **`build/HELLO`**, a load module with the attributes RENT and REUS (the linker options **`--rent --reus`**). The file **`mbt.lock`** records the release that was picked. Keep it in the repository, so that the next build uses the same headers.

Deploying and Routing

The Library

`mbt deploy` sends the load module to the system and writes it into the library of the project. For HELLO this is the development library of HTTPD:

```
$ mbt deploy --target env
[mbt] Deploy target: HTTPD.DEV.LINKLIB
[mbt] Modules (1): HELLO
[mbt] Packed 1 module(s) -> hello.deploy.xmit
[mbt] Uploading hello.deploy.xmit -> IBMUSER.MBT.XMIT.IN...
[mbt] RECEIVE IBMUSER.MBT.XMIT.IN -> IBMUSER.HELLO.MBTDPLY, IEBCOPY -> HTTPD.DEV.LINKLIB (replace, DISP=SHR)...
[mbt] HTTPD.DEV.LINKLIB: 35% of 248 tracks used, 1 extent(s)
[mbt] Deploy complete: 1 module(s) -> HTTPD.DEV.LINKLIB
```

The command replaces the member and does not delete the library. The library must be in the **`STEPLIB`** of the HTTPD procedure, or the server does not find the module. The development library is put first, so that a deployed level is found before the one that SMP installed:

```
//STEPLIB  DD  DISP=SHR,DSN=HTTPD.DEV.LINKLIB
//         DD  DISP=SHR,DSN=HTTPD.LINKLIB
```

Two rules go with it.

- Every library in an authorized **`STEPLIB`** must be APF authorized. A library that is not authorized takes the authority away from the whole task, and the failure that follows does not name the cause. Where HTTPD is authorized by its library (**`HTTPD002I`** says **`AUTHORIZED BY LIBRARY`**), the development library needs its own entry, beside the one of the SMP library. MVS 3.8j reads the list at the IPL and has no dynamic APF, so the entry needs an IPL (Chapter 2, “Installing HTTPD” on page 5).
- Use the development library only on a test system. It sits first in the concatenation, and it runs whatever was deployed last. The library **`HTTPD.LINKLIB`** belongs to SMP, and only the install job writes it.

The Route

The module runs only when a route names it. For HELLO the member has one route:

```
PORT=8080
DOCROOT=/www
MOD=HELLO /hello               AUTH=NONE
```

The route has **`AUTH=NONE`**, so anybody who can reach the port can call it. That suits a greeting on a test system. A module that does anything of value gets **`AUTH=BASIC`** or stronger (Chapter 6, “Authentication and Security” on page 55), and a route without the keyword is public too. The syntax of the line is in Chapter 5, “Routes and Server Modules” on page 47.

The Restart

HTTPD loads each module of its routes once, when it starts (the Parmlib keyword **`RESIDENT=YES`**, the default). A new level of the module in the library has no effect on a running server. Deploy, then restart:

```
P HTTPD
S HTTPD,M=HELLOPRM
```

The start shows the route:

```
S HTTPD,M=HELLOPRM
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.37.55
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE HELLO REGISTERED FOR /hello
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CDCC0) TASK(115FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD7F0)   TCB(9CD578) TASK(129FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD100)   TCB(9BBEB8) TASK(13AFC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD080)   TCB(9BBB88) TASK(14BFC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

HTTPD036I MODULE HELLO REGISTERED FOR /hello is the line to look for. If the module is neither reentrant nor reusable, the server writes **HTTPD441W** and fetches it for every request. With **RESIDENT=NO** each request fetches the module, so a new level is used at once. That is slower, and it is for the days when you write the module. A module that is in no library of the **STEPLIB** makes every request to the route answer with the status 503 and the text “External program HELLO was not found” (“When a Module Abends” on page 107).

What the Procedure Must Have

A module opens files by ddname in the procedure of the started task, because it has no allocations of its own. If your module needs a DD, the procedure must have it, and every operator who runs HTTPD must keep it. The DDs that come with the procedure are listed in the *HTTPD Reference*, in the chapter “The Server Module Interface”, section “The Started Task Procedure”. Before you remove a DD from the procedure, find out which modules of other products use it.

Testing with curl

Test every module with **curl -i**, which shows the status line and the headers. For each method that the module accepts, and for one that it does not, check four things: the status, the **Content-Type**, the framing (a **Transfer-Encoding: chunked** or a **Content-Length**), and the body.

The tests for HELLO are the screens of this chapter:

Table 31. The tests of HELLO

Test	What it shows
GET without a parameter	200, chunked, the body “Hello, world!”.
GET with **?name=IBMUSER**	The parameter reaches the module.
POST of a form	200, **Connection: close**, **POST_STRING** holds the raw body.
POST of **text/plain**	The same variable holds a text body.
PUT	405 and the header **Allow**.
Two GETs on one connection	The second request re-uses the first connection.
Time of one GET	A fast answer. Seconds mean a fault in the end of the headers.

If the status is 503 and the text starts with “External program”, the module did not run or did not finish. The next section tells what the texts mean.

When a Module Abends

HTTPD runs the module through a routine that sets an ESTAE. If the module abends, the ESTAE returns control to the server, and the worker lives. It goes on with its next request. The server then does this, in order.

1. It frees subpool 5. If the free fails, it writes **`HTTPD906W`**.
2. It builds the text of the answer from the result of the LINK.
3. If the module has not started a response, it sends the status 503 with the type **`text/plain`**. If the module has begun the response, the text goes after what the module sent.
4. It writes the body “External program NAME text”, and the console message **`HTTPD908E EXTERNAL PROGRAM NAME text`**.
5. It ends the request.

The text says what happened.

Table 32. The texts of a failed module

Text	Cause
was not found	The module is in no library of the STEPLIB or link list (S806).
could not be loaded	The LINK failed for another reason, for example a fetch error or too little storage.
could not be started, ESTAE create failed	The ESTAE could not be set.
returned rc n	The module returned the negative code n. Only a module with its own start routine can return it, because the start routine of the shipped library turns a negative code of **`main`** into 0.
failed with Snnn ABEND	A system abend with the code Snnn.
failed with Unnnn ABEND	A user abend with the code Unnnn, in decimal.

The reclaim of subpool 5 gives back what the module got with **`malloc`**. It does not give back what the module holds outside subpool 5: a file that the module opened, a UFS session that it created, or storage from other subpools. A module that has such resources and must survive an abend needs a recovery of its own. The chapter “The Server Module Interface” of the *HTTPD Reference* has the detail, and Chapter 12, “Troubleshooting” on page 141 covers the search for a fault.

Long-Running Handlers

A request that takes long holds a worker. When the server stops, it sets two flags in the flag byte of the server block, **`HTTPD_FLAG_QUIESCE`** and **`HTTPD_FLAG_SHUTDOWN`**, and then waits up to 10 seconds, in steps of one second, for its socket thread. The sends of the server look at the flags: they stop and end the request. A module that loops, waits or sends in pieces must look at the flags itself, in every round, and return when one is set:

```
while (more_to_send) {
    if (http_get_flag(httpd) & (HTTPD_FLAG_QUIESCE | HTTPD_FLAG_SHUTDOWN))
        break;
    if (httpc->state == CSTATE_DONE)
        break;
```

```
    /* ... send the next piece ... */
}
```

A module that does not look keeps its worker busy while the server tries to stop (“Stopping” on page 82). Both flag macros exist from HTTPD 4.0.

HTTPD Levels and the Function Vector

A module calls every function of the server through the HTTPX vector. The vector is append-only: an entry never moves and never changes its meaning, and a new function is added at the end. This is the promise that lets a module that was built a year ago run on a new HTTPD.

The other direction needs care. A module that was built against a newer header and runs on an older server calls an entry that the older server does not have. No function tells a module which entries exist, and there is no run-time test for an entry. The one exception is **`http_cgi_subpool`**, which a module can test because an older server holds 0 in that entry. The rule for the author is therefore plain: name the lowest HTTPD level of the module in its own documentation, and put the same level in the range of the dependency in **`mbt.toml`**.

The function **`http_new_session`** is a good example. It is the last entry of the vector, at offset X’144′, and it exists from HTTPD 4.2:

```
int http_new_session(HTTPC *httpc);
```

It turns the login of the current request into a session of its own, with its own credential and its own token, which **`http_get_token`** returns from then on. It is for an explicit login endpoint of a module. Each call costs a RACF logon and a place in the credential store, and a client holds at most 16 such sessions.

There is a second level. From HTTPD 4.5 the function tells why it failed. Before that release every failure returned −1. The results are:

Table 33. The results of http_new_session

Constant	Value	Meaning
	0	The session exists.
`HTTP_SESSION_NONE`	−1	The request is not authenticated, or carries no password to log on again with.
`HTTP_SESSION_REFUSED`	−2	RACF refused the logon. The module answers 401.
`HTTP_SESSION_BUSY`	−3	The client holds 16 sessions that are in use, or the credential store is full. Retrying later may work. The module answers 503.
`HTTP_SESSION_NOMEM`	−4	No storage, or RACF could not be asked for a reason that is not about the password. A fault of the server. The module answers 500.

A module that must run on both generations of server tests the result with **`!= 0`** or **`< 0`**. It never tests with **`== -1`**: on a new server that reads −2, −3 and −4 as success, and the module then goes on to **`http_get_token`**, which hands out the token of the shared credential that the call exists to avoid. It treats −1 as a failure of unknown reason, because an old server returns −1 for every failure.

```
rc = http_new_session(httpc);
if (rc == 0) {
    /* the session exists: http_get_token() returns its token.
       If the module cannot send the token, it calls http_logout(httpc)
       before it returns, or the session keeps its slot until it times out. */
}
else if (rc == HTTP_SESSION_REFUSED) {
    http_resp(httpc, 401);          /* wrong or revoked password */
}
else if (rc == HTTP_SESSION_BUSY) {
    http_resp(httpc, 503);          /* try again later */
}
else {
    http_resp(httpc, 500);          /* -1, -4, or a value not yet known */
}
```

After −2 the credential of the request is marked unusable. The current request still runs on it and must answer the refusal. The constants are in the header of a release that has them: a module that mentions **`HTTP_SESSION_REFUSED`** must be built with the headers of HTTPD 4.5 or later, and that belongs in the documentation of the module too. The details are in the chapter “The Server Module Interface” of the *HTTPD Reference*.

Things to Know

- **The entry is main.** There is no **`cgimain`**. The blocks come from the global runtime table.
- **End the headers with a call of its own: `http_printf(httpc, "\r\n")`.** A blank line inside a longer text is not recognized, and the client waits for the end of the connection.
- **No writable statics.** Put state in locals, on the heap, or in **`__wsaget()`**. Never change a string constant.
- **Set the stack: `unsigned __stklen = 64*1024;`** in the source file of the module.
- **A new level needs a restart,** unless the server runs with **`RESIDENT=NO`**.
- **Every library of an authorized STEPLIB is APF authorized.** The development library is for a test system.
- **Look at the flags** in a handler that runs long.
- **Name the lowest HTTPD level** that the module needs.

Chapter 10. Upgrading HTTPD

This chapter moves a system that runs an older HTTPD to HTTPD 4.5.1. It is written for a system that HTTPD 4.0.x, 4.1.x, 4.2.x, 4.3.x, 4.4.x or 4.5.0 installed by SMP, and it has a section for a system that still runs HTTPD 3.x. The release that 4.5.1 replaces in one step is 4.5.0. A system at an older release goes to 4.5.0 first (“One Release at a Time”). A system without HTTPD needs Chapter 2, “Installing HTTPD” on page 5, and a system that should lose HTTPD needs Chapter 11, “Removing HTTPD” on page 127.

The upgrade is the same install job as in Chapter 2, “Installing HTTPD” on page 5, with a different effect. Each release names its predecessor in the SYSMOD. When SMP applies the new release, it deletes the load modules of the old one from **HTTPD.LINKLIB** and copies the new ones in. This is what an upgrade does:

- It replaces the five load modules in place. The names of the data sets carry no version, so the **STEPLIB** of the procedure and the APF entry stay as they are.
- It does not run the allocation job. The libraries exist already.
- It does not touch the document root disk.
- It does not touch your procedure, your Parmlib member or your RAKF definitions. SMP never writes to **PARMLIB** or to a procedure library. What changed in the sample members is yours to compare and to carry over (“After the Upgrade: Review Your Own Files” on page 124).
- It leaves no way back. The install job accepts the new level, and the old FMID is deleted. SMP cannot restore the old modules.

The captures in this chapter come from the upgrade from 4.5.0 to 4.5.1 on a TK5 system. Every step of the install job ended with return code 0000 except the APPLY, which ended with return code 0004 and **HMA2461 SYSMOD THTP450 NOT FOUND ON SMPSCDS LIBRARY**. The package of 4.5.1 was built from the source of 4.5.1 before the release was published. The build stamp in the start-up banner is therefore not the one of the published package (“Verifying the Upgrade” on page 120). The file names in your package carry the number of your release.

One Release at a Time

Go from release to release. Do not skip one. The reason is in the way SMP decides who owns a module.

The FMID of a release is **THTP** and the three digits of its version. For a minor release, the package build derives the FMID and the predecessor from the version number. 4.5.1 is a patch release, which the derivation has no id for, so the project file of 4.5.1 names both: **THTP451**, deleting **THTP450**. A release deletes only the FMID of the release before it.

Table 34. Releases, FMIDs and what each one deletes

Release	FMID	Deletes
4.1.0	**THTP410**	**THTP400**
4.2.0	**THTP420**	**THTP410**
4.3.0	**THTP430**	**THTP420**
4.4.0	**THTP440**	**THTP430**

4.5.0	**THTP450**	**THTP440**
4.5.1	**THTP451**	**THTP450**

SMP keys the ownership of a module on its name. The SYSMOD of 4.5.1 can take over **HTTPD** from **THTP450** because it deletes **THTP450**. It cannot take it over from **THTP440** or from an older FMID. By the same rule, if the system is at 4.4.0 and you install 4.5.1, the SYSMOD does not own the module, and the install copies nothing. Every step still ends with return code 0. The inventory then says that 4.5.1 is installed, and the library holds the modules of 4.4.0. This rule is SMP's and holds for every pair of releases. The case of 4.4.0 and 4.5.1 was run on a TK5 system at 4.4.0: every step of the install job of 4.5.1 ended with return code 0000, the **APPLY** had no **HMA2380** line, and the banner of the started server still showed 4.4.0. The library kept the modules of 4.4.0.

A skipped release has a sign in the output. The **APPLY** step has no **HMA2240** lines (a deleted load module) and no **HMA2380** lines (a copied load module). A good upgrade has five of each. Look for them in every upgrade ("Reading the Result" on page 115). The lines **NOT SEL** are no sign: a good upgrade has them too.

Each id is spent once. The inventory of the skipped install is not a state you can repeat. This book does not show a measured repair for it. Check the level first ("Before You Start"), so that the case does not arise.

The accepted copy has its own zone, the ACDS, and the ACCEPT step deletes there on its own. An upgrade is complete when both zones have moved ("Verifying the Upgrade" on page 120).

A system at 4.0.x needs the packages of 4.1.0, 4.2.0, 4.3.0, 4.4.0 and 4.5.0 before the package of 4.5.1. The hop from 4.0.x to 4.1.0 is different from the others, and "Upgrading from HTTPD 4.0.x" on page 122 describes it.

Before You Start

Goal: you know the present level, you have the packages you need, and you have kept what the upgrade replaces.

The present level. Ask SMP for the level of the load module **HTTPD** and for the two SYSMODs. The job is the procedure **SMPAPP** with these control statements. Name the FMID of your present level and the FMID of the release you install:

```
//LISTCDS  EXEC SMPAPP
//SMPCNTL  DD  *
 RESETRC .
 LIST CDS SYSMOD(THTP450) .
 LIST CDS LMOD(HTTPD) .
 LIST CDS SYSMOD(THTP451) .
/*
```

The output of this job, on the system before the upgrade:

```
     DATE 26.303  TIME 23:27:14      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                    PAGE 0001

HMA4240    HMASMP EXEC PARM = 'DATE=U'
 RESETRC .
HMA2050    RESETRC PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


 LIST CDS SYSMOD(THTP450) .
     DATE 26.303  TIME 23:27:14      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                    PAGE 0002

SMPCDS   SYSMOD ENTRIES


  NAME

THTP450   TYPE           = FUNCTION
```

```
          STATUS          = REC  APP  ACC
          FMID            = THTP450
          JCLIN           = YES
          DATE/TIME REC   = 26.303  23:13:01
                    APP   = 26.303  23:13:37
                    ACC   = 26.303  23:13:55
          DELETE VER(001) = THTP440
          MOD             = ABEND0C1  HTTPD     HTTPDM    HTTPDMTT  HTTPDSRV

     DATE 26.303  TIME 23:27:14      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0003

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


 LIST CDS LMOD(HTTPD) .
     DATE 26.303  TIME 23:27:14      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0004

SMPCDS  LOAD MODULE ENTRIES


  NAME

HTTPD     LASTUPD         = THTP450  TYPE=ADD
          SYSTEM LIBRARY  = LINKLIB
          LKED ATTRIBUTES = COPY

     DATE 26.303  TIME 23:27:14      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0005

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


 LIST CDS SYSMOD(THTP451) .
     DATE 26.303  TIME 23:27:14      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0006

LIST     SELECT SUMMARY REPORT FOR SMPCDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING


 TYPE        NAME
 SYSMOD      THTP451
     DATE 26.303  TIME 23:27:14      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0007

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04


HMA2050    HMASMP PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04
```

Read three facts. The line **`LASTUPD = THTP450`** under the load module **`HTTPD`** is the present level: the module in the library comes from that FMID. The SYSMOD **`THTP450`** has the status **`REC APP ACC`** and the line **`DELETE VER(001) = THTP440`**, the one it replaced. And the new SYSMOD **`THTP451`** is not known: the answer is the list of selected entries that were not found or not eligible, and the step ends with return code 04. That is the answer you want before the upgrade.

If **`LASTUPD`** does not name the predecessor of the release you are about to install (“One Release at a Time” on page 111), install the missing releases first.

The library members. The library before the upgrade, listed with the utility IEHLIST. The list is the reference for the comparison after the upgrade:

```
                            SYSTEMS SUPPORT UTILITIES---IEHLIST                              PAGE    1
 DIRECTORY INFO FOR SPECIFIED PDS ON VOL TSO001
HTTPD.LINKLIB
           MEMBERS     TTRC     VARIABLE USER DATA ---(USER DATA AND TTRC ARE IN HEX)
          ABEND0C1  00292E2C  002A070000 000000C2C2 0152583060 0027B08800 00010000
          HTTPD     002C1F2C  002D210000 000000C2C2 0478C80558 012CC08800 00010100
          HTTPDM    00352B2C  0036050000 000000C2C2 016DD830C8 003BF88800 00010000
          HTTPDMTT  0038252C  0039070000 000000C2C2 01669833A0 002D888800 00010000
          HTTPDSRV  003B252C  003C0B0000 000000C2C2 02437009F8 00B9D88800 00010000

OF THE 00020 DIRECTORY BLOCKS ALLOCATED TO THIS PDS, 00019 ARE(IS) COMPLETELY UNUSED
```

The five members are the five load modules of the package. The column **`TTRC`** is the place of the member in the library.

Copies. The install job scratches **`HTTPD.SAMPLIB`** and receives it again (“Edit and Submit the Install Job” on page 115). If you edited a member there, copy the member out before you submit the job. Your procedure and your Parmlib member belong in **`PROCLIB`** and **`PARMLIB`**, where SMP does not look.

A way back is yours to build, outside SMP. Copy **HTTPD.LINKLIB** to a data set of another name before the upgrade, with the utility you prefer. In an emergency a **STEPLIB** that names the copy starts the old modules. The copy needs an APF entry of its own for that. This book shows no session of it.

The STEPLIB. Look at the **STEPLIB** of your procedure. If it names a development library before **HTTPD.LINKLIB**, that library wins, and the server keeps running the module of the development build after the upgrade. The system takes the first library of the concatenation that has the module. "The Started Task and the Package" in the *HTTPD Reference*, section "The STEPLIB", has the rule.

Upgrading by One Release

An upgrade has the same tasks as an installation, without the allocation.

Stop the Server

Goal: nothing has **HTTPD.LINKLIB** open. The started task loads from the library, and the install job writes into it. A job that cannot get the library waits (Chapter 2, "Installing HTTPD" on page 5 describes the wait).

Stop the server with the console command **P HTTPD**:

```
P HTTPD
HTTPD098I HTTPD SHUTTING DOWN
HTTPD060I SHUTDOWN SOCKET THREAD    TCB(9AD080) TASK(162FC8) STACKSIZE(32768)
HTTPD060I SHUTDOWN WORKER(174C48)   TCB(9A09C8) TASK(198FC8) STACKSIZE(65536)
HTTPD060I SHUTDOWN WORKER(174CC8)   TCB(9A0CF8) TASK(187FC8) STACKSIZE(65536)
HTTPD060I SHUTDOWN WORKER(0ED7B0)   TCB(9AD828) TASK(176FC8) STACKSIZE(65536)
HTTPD416I STATS: 0 REQUESTS, 0 ERRORS, 0 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
IEF404I HTTPD - ENDED - TIME=23.23.14
$HASP395 HTTPD    ENDED
$HASP150 HTTPD    ON PRINTER2       151 LINES
$HASP160 PRINTER2 INACTIVE - CLASS=Z
$HASP250 HTTPD    IS PURGED
```

The message **HTTPD098I** starts the stop. The threads end one by one (**HTTPD060I**), and **HTTPD416I** prints the counters of the run. The message **HTTPD099I** says the shutdown is complete. The line **$HASP395 HTTPD ENDED** says that the address space is gone. Wait for it. There is no **F HTTPD,SHUTDOWN**, although the comments in the sample procedure list it ("Console Commands" in the *HTTPD Reference*).

Send the XMIT Files

Goal: the two XMIT files of the new release are on the system as data sets with fixed records of 80 bytes, sent in binary. The session is the same as for an installation:

```
$ ftp -v -n mvs.example.org 2121
Connected to mvs.example.org.
220 MVS 3.8j FTPD Server
ftp> user HERC01
331 Send password please.
Password:
230 HERC01 is logged on.  Working directory is "HERC01.".
Remote system type is MVS.
ftp> progress off
Progress bar off.
ftp> binary
200 Representation type is Image
ftp> put httpd-4.5.1-load.xmit 'HERC01.HTTPD.LOAD.XMIT'
local: httpd-4.5.1-load.xmit remote: 'HERC01.HTTPD.LOAD.XMIT'
229 Entering Extended Passive Mode (|||2122|)
125 Storing data set HERC01.HTTPD.LOAD.XMIT
250 Transfer completed successfully.
843520 bytes sent in 00:03 (246.84 KiB/s)
ftp> put httpd-4.5.1-samplib.xmit 'HERC01.HTTPD.SAMP.XMIT'
local: httpd-4.5.1-samplib.xmit remote: 'HERC01.HTTPD.SAMP.XMIT'
229 Entering Extended Passive Mode (|||2122|)
125 Storing data set HERC01.HTTPD.SAMP.XMIT
250 Transfer completed successfully.
22240 bytes sent in 00:00 (278.88 KiB/s)
ftp> ls HTTPD.LOAD.XMIT
229 Entering Extended Passive Mode (|||2122|)
150 Opening data connection for file list
```

```
Volume Unit    Referred Ext Used Recfm Lrecl BlkSz Dsorg Dsname
TSO001 3390 2026/10/30  1   19  FB       80  3120  PS   HTTPD.LOAD.XMIT
226 List completed successfully.
ftp> ls HTTPD.SAMP.XMIT
229 Entering Extended Passive Mode (|||2122|)
150 Opening data connection for file list
Volume Unit    Referred Ext Used Recfm Lrecl BlkSz Dsorg Dsname
TSO001 3390 2026/10/30  1    1  FB       80  3120  PS   HTTPD.SAMP.XMIT
226 List completed successfully.
ftp> quit
221 Quit command received. Goodbye.
```

The names of the data sets are yours. The install job takes them in the next task. Chapter 2, “Installing HTTPD” on page 5 explains the upload, and what to do with a text transfer. Use FTP or another route that does not need the HTTP server. If mvsMF is your upload path, it is stopped now.

Edit and Submit the Install Job

Goal: the install job names your two data sets. Do not run the allocation job. The package README says to run it before the install job. For an upgrade you skip that step: the libraries exist and hold the modules that the SYSMOD replaces.

Change the two **`RECEIVE INDSN`** lines of the install job and nothing else. The listing is the difference between the original and the edited job:

```
$ diff httpd-4.5.1-inst.jcl.orig httpd-4.5.1-inst.jcl
57c57
<   RECEIVE INDSN('CHANGE.ME.HTTPLOAD') -
---
>   RECEIVE INDSN('HERC01.HTTPD.LOAD.XMIT') -
66c66
<   RECEIVE INDSN('CHANGE.ME.SAMPLIB') -
---
>   RECEIVE INDSN('HERC01.HTTPD.SAMP.XMIT') -
```

Submit the job as in Chapter 2, “Installing HTTPD” on page 5. The job takes about a minute.

Reading the Result

Goal: know whether the upgrade did its work. The console shows that the job started and that it ended. It does not show the steps:

```
$HASP100 HTTPDINS ON INTRDR      HTTPD INSTALL
$HASP373 HTTPDINS STARTED - INIT  1 - CLASS A - SYS TK5R
IEF403I HTTPDINS - STARTED - TIME=23.28.04
IEF404I HTTPDINS - ENDED - TIME=23.29.00
$HASP395 HTTPDINS ENDED
$HASP309    INIT  1 INACTIVE ******** C=A
IKJ144I   UNDEFINED USERID(S)DUMMY
```

The step table is in the job log, with the time of each step:

```
                                        J E S 2   J O B   L O G
23.28.04 JOB   nn  $HASP373 HTTPDINS STARTED - INIT  1 - CLASS A - SYS TK5R
23.28.04 JOB   nn  IEF403I HTTPDINS - STARTED - TIME=23.28.04
23.28.04 JOB   nn  IEFACTRT - Stepname  Procstep  Program   Retcode
23.28.04 JOB   nn  HTTPDINS   DELOLD              IDCAMS    RC= 0000
23.28.04 JOB   nn  HTTPDINS   RECV1               IKJEFT01  RC= 0000
23.28.04 JOB   nn  HTTPDINS   RECV2               IKJEFT01  RC= 0000
23.28.05 JOB   nn  HTTPDINS   RECV      HMASMP    HMASMP    RC= 0000
23.28.22 JOB   nn  HTTPDINS   APPLYCHK  HMASMP    HMASMP    RC= 0000
23.28.48 JOB   nn  HTTPDINS   APPLY     HMASMP    HMASMP    RC= 0004
23.29.00 JOB   nn  HTTPDINS   ACCEPT    HMASMP    HMASMP    RC= 0000
23.29.00 JOB   nn  HTTPDINS   CLEANUP             IDCAMS    RC= 0000
23.29.00 JOB   nn  IEF404I HTTPDINS - ENDED - TIME=23.29.00
23.29.00 JOB   nn  $HASP395 HTTPDINS ENDED
```

The step **`APPLY`** ends with 0004. That is the normal result of an upgrade. The next paragraphs say why. Every other step ends with 0000.

DELOLD. The step makes the two RECEIVE targets absent. Both are data sets of the previous run, so the capture looks different from the installation:

```
IDCAMS  SYSTEM SERVICES                                    TIME: 23:28:04       10/30/26     PAGE   1

  DELETE HTTPD.HTTPLOAD NONVSAM SCRATCH PURGE
IKJ56228I DATA SET HTTPD.HTTPLOAD NOT IN CATALOG OR CATALOG CAN NOT BE
IKJ56228I ACCESSED
IDC0551I **ENTRY HTTPD.HTTPLOAD NOT DELETED
IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 8

  DELETE HTTPD.SAMPLIB NONVSAM SCRATCH PURGE
IDC0550I ENTRY (A) HTTPD.SAMPLIB DELETED
IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 0

  SET MAXCC=0

IDC0002I IDCAMS PROCESSING COMPLETE. MAXIMUM CONDITION CODE WAS 0
```

The data set **`HTTPD.HTTPLOAD`** is not in the catalog. The previous install job scratched it in its last step, and the delete ends with condition code 8. The data set **`HTTPD.SAMPLIB`** exists, and the message **`IDC0550I`** says it is deleted. Your copies of the old members are gone with it. The statement **`SET MAXCC=0`** makes the step end with 0.

RECV1 and RECV2. TSO RECEIVE unpacks the two XMIT files:

```
  RECEIVE INDSN('HERC01.HTTPD.LOAD.XMIT')     DATASET('HTTPD.HTTPLOAD')
NJE38 RECEIVE v2.5.0

Receiving 'HTTPD.V4R5M1.LINKLIB' from ORIGNODE(IBMUSER)

Receive into 'HTTPD.HTTPLOAD' successful
END
```

```
  RECEIVE INDSN('HERC01.HTTPD.SAMP.XMIT')     DATASET('HTTPD.SAMPLIB')
NJE38 RECEIVE v2.5.0

Receiving 'HTTPD.SAMPLIB' from ORIGNODE(HTTPD)

Receive into 'HTTPD.SAMPLIB' successful
END
```

The line to find is **`Receive into ... successful`**. The name after **`Receiving`** is the name the XMIT had where it was made. It is not the name of your data set.

RECV. SMP stores the SYSMOD. The output repeats it and ends with **`HMA3930`**:

```
     DATE 26.303  TIME 23:28:05      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                    PAGE 0001

HMA4240    HMASMP EXEC PARM = 'DATE=U'
 RESETRC .
HMA2050    RESETRC PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


 RECEIVE SELECT(THTP451) .




----------++FUNCTION(THTP451) .

----------++VER(Z038) DELETE(THTP450)
----------   /* httpd 4.5.1                                          */
----------   /* Load modules are copied from the LKLIB, not re-bound.      */ .

----------++JCLIN .

----------++MOD(ABEND0C1) LKLIB(HTTPLOAD) DISTLIB(AHTTPLOD) .

----------++MOD(HTTPD) LKLIB(HTTPLOAD) DISTLIB(AHTTPLOD) .

----------++MOD(HTTPDM) LKLIB(HTTPLOAD) DISTLIB(AHTTPLOD) .

----------++MOD(HTTPDMTT) LKLIB(HTTPLOAD) DISTLIB(AHTTPLOD) .

----------++MOD(HTTPDSRV) LKLIB(HTTPLOAD) DISTLIB(AHTTPLOD) .

HMA3930    SYSMOD THTP451 SUCCESSFULLY RECEIVED
```

```
     DATE 26.303  TIME 23:28:05      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0002

                       RECEIVE  SUMMARY  REPORT


     SYSMOD   STATUS     TYPE      ------------------------

     THTP451  RECEIVED   FUNCTION
     DATE 26.303  TIME 23:28:05      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0003

HMA2050    RECEIVE PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


HMA2050    HMASMP PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
```

The second line of the SYSMOD is the one that makes the upgrade: **++VER(Z038) DELETE(THTP450)**. It names the predecessor.

APPLYCHK. A dry run. It changes nothing. The summary names the predecessor as deleted:

```
     DATE 26.303  TIME 23:28:05      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0001

HMA4240    HMASMP EXEC PARM = 'DATE=U'
 APPLY S(THTP451) CHECK .


HMA4180    INLINE JCLIN PROCESSING SUCCESSFUL FOR SYSMOD=THTP451

HMA2050    APPLY PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
     DATE 26.303  TIME 23:28:22      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0002

SYSMOD STATUS REPORT FOR APPLY CHECK PROCESSING

  NOTE:  '-' INDICATES THE REQUISITE SYSMOD CONDITION IS NOT SATISFIED
         '*' INDICATES THE NON SATISFIED REQUISITE SYSMOD CONDITION IS BYPASSED


SYSMOD   STATUS    TYPE      FMID     REQUISITE AND SUPEDBY SYSMODS

THTP450  DELETED   FUNCTION  THTP450

THTP451  APPLIED   FUNCTION  THTP451
     DATE 26.303  TIME 23:28:22      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0003

ELEMENT SUMMARY REPORT FOR APPLY CHECK PROCESSING


ELEM   ELEMENT   ELEM      CURRENT  CURRENT  MAC/SRC   DISTSRC   ASSEM    LOAD MOD  ---LMOD SYSLIB---  SYSMOD   SYSMOD
TYPE   NAME      STATUS    FMID     RMID     SYSLIB    LIBRARY   NAMES                                 NAME     STATUS

MOD    ABEND0C1  APPLIED   THTP451  THTP451                              ABEND0C1  LINKLIB            THTP451  APPLIED
                 NOT SEL                                                                             THTP450  DELETED

MOD    HTTPD     APPLIED   THTP451  THTP451                              HTTPD     LINKLIB            THTP451  APPLIED
                 NOT SEL                                                                             THTP450  DELETED

MOD    HTTPDM    APPLIED   THTP451  THTP451                              HTTPDM    LINKLIB            THTP451  APPLIED
                 NOT SEL                                                                             THTP450  DELETED

MOD    HTTPDMTT  APPLIED   THTP451  THTP451                              HTTPDMTT  LINKLIB            THTP451  APPLIED
                 NOT SEL                                                                             THTP450  DELETED

MOD    HTTPDSRV  APPLIED   THTP451  THTP451                              HTTPDSRV  LINKLIB            THTP451  APPLIED
                 NOT SEL                                                                             THTP450  DELETED
     DATE 26.303  TIME 23:28:22      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0004

DELETED FUNCTION REPORT FOR  APPLY CHECK PROCESSING

SYSMOD          DELETED THE
CAUSING THE     FOLLOWING SYSMODS
DELETION        TYPE       SYSMOD

THTP451         FUNCTION   THTP450

     DATE 26.303  TIME 23:28:22      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0005

HMA2050    HMASMP PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
```

The return code is 00. The lines **THTP450 DELETED** show that SMP knows the predecessor and will delete it. The step has no **HMA2240** and no **HMA2380**, because a check does not do the work.

APPLY. This step does the work:

```
     DATE 26.303  TIME 23:28:23      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0001

HMA4240    HMASMP EXEC PARM = 'DATE=U'
 APPLY S(THTP451) DIS(WRITE) .
HMA4140    SMPCDS DIRECTORY SUCCESSFULLY LOADED FOR IN-STORAGE UPDATE OPERATIONS


HMA2240    SUCCESSFULLY DELETED LMOD ABEND0C1 ON LINKLIB LIBRARY
HMA2240    SUCCESSFULLY DELETED LMOD HTTPD ON LINKLIB LIBRARY
HMA2240    SUCCESSFULLY DELETED LMOD HTTPDM ON LINKLIB LIBRARY
HMA2240    SUCCESSFULLY DELETED LMOD HTTPDMTT ON LINKLIB LIBRARY
HMA2240    SUCCESSFULLY DELETED LMOD HTTPDSRV ON LINKLIB LIBRARY
HMA4180    INLINE JCLIN PROCESSING SUCCESSFUL FOR SYSMOD=THTP451

HMA2380    COPY SUCCESSFUL - MOD=ABEND0C1 - LMOD=ABEND0C1 - LIBRARY=LINKLIB - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPD - LMOD=HTTPD - LIBRARY=LINKLIB - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDM - LMOD=HTTPDM - LIBRARY=LINKLIB - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDMTT - LMOD=HTTPDMTT - LIBRARY=LINKLIB - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDSRV - LMOD=HTTPDSRV - LIBRARY=LINKLIB - SYSMOD=THTP451 - RETURN CODE=00
HMA2270    APPLY PROCESSING SUCCESSFULLY COMPLETED FOR SYSMOD THTP451
HMA2461    SYSMOD THTP450 NOT FOUND ON SMPSCDS LIBRARY
HMA3680    SMPCDS IN STORAGE DIRECTORY SUCCESSFULLY REWRITTEN
HMA2050    APPLY PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04
     DATE 26.303  TIME 23:28:47      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0002

SYSMOD STATUS REPORT FOR APPLY PROCESSING

  NOTE:  '-' INDICATES THE REQUISITE SYSMOD CONDITION IS NOT SATISFIED
         '*' INDICATES THE NON SATISFIED REQUISITE SYSMOD CONDITION IS BYPASSED


SYSMOD   STATUS    TYPE      FMID     REQUISITE AND SUPEDBY SYSMODS

THTP450  DELETED   FUNCTION  THTP450

THTP451  APPLIED   FUNCTION  THTP451
     DATE 26.303  TIME 23:28:47      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0003

ELEMENT SUMMARY REPORT FOR APPLY PROCESSING


ELEM   ELEMENT   ELEM      CURRENT  CURRENT  MAC/SRC   DISTSRC   ASSEM    LOAD MOD  ---LMOD SYSLIB---  SYSMOD   SYSMOD
TYPE   NAME      STATUS    FMID     RMID     SYSLIB    LIBRARY   NAMES                                 NAME     STATUS

MOD    ABEND0C1  APPLIED   THTP451  THTP451                                ABEND0C1  LINKLIB            THTP451  APPLIED
                 NOT SEL                                                                               THTP450  DELETED

MOD    HTTPD     APPLIED   THTP451  THTP451                                HTTPD     LINKLIB            THTP451  APPLIED
                 NOT SEL                                                                               THTP450  DELETED

MOD    HTTPDM    APPLIED   THTP451  THTP451                                HTTPDM    LINKLIB            THTP451  APPLIED
                 NOT SEL                                                                               THTP450  DELETED

MOD    HTTPDMTT  APPLIED   THTP451  THTP451                                HTTPDMTT  LINKLIB            THTP451  APPLIED
                 NOT SEL                                                                               THTP450  DELETED

MOD    HTTPDSRV  APPLIED   THTP451  THTP451                                HTTPDSRV  LINKLIB            THTP451  APPLIED
                 NOT SEL                                                                               THTP450  DELETED
     DATE 26.303  TIME 23:28:47      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0004

DELETED FUNCTION REPORT FOR  APPLY PROCESSING

SYSMOD          DELETED THE
CAUSING THE     FOLLOWING SYSMODS
DELETION        TYPE       SYSMOD

THTP451         FUNCTION   THTP450

     DATE 26.303  TIME 23:28:47      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0005

HMA2050    HMASMP PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04
```

Read these lines in order:

1. **HMA2240 SUCCESSFULLY DELETED LMOD ... ON LINKLIB LIBRARY**, five times. SMP deletes the load modules of the old FMID from **HTTPD.LINKLIB**. This is the delete that the SYSMOD asked for.
2. **HMA4180 INLINE JCLIN PROCESSING SUCCESSFUL**. SMP has read the JCLIN of the new SYSMOD.
3. **HMA2380 COPY SUCCESSFUL - MOD=... - LIBRARY=LINKLIB**, five times. SMP copies the new load modules in. This is the line that separates a real upgrade from a silent one. Count it.
4. **HMA2270 APPLY PROCESSING SUCCESSFULLY COMPLETED FOR SYSMOD THTP451**.
5. **HMA2461 SYSMOD THTP450 NOT FOUND ON SMPSCDS LIBRARY**, and the return code 04. SMP looks for a backup entry of the deleted FMID, and a deleted FMID has none. The message comes after the success message and does not undo it.

The page at the end, **DELETED FUNCTION REPORT**, shows **THTP451** as the function that deleted **THTP450**. The report is the second proof that the predecessor is gone.

The copy runs through the utility IEBCOPY. Its listing from the same step:

```
                                  IEBCOPY MESSAGES AND CONTROL STATEMENTS                          PAGE 0001
                    COPY OUTDD=LINKLIB,INDD=HTTPLOAD
                    SELECT MEMBER=((ABEND0C1,,R))          THTP451
                    SELECT MEMBER=((HTTPD,,R))             THTP451
                    SELECT MEMBER=((HTTPDM,,R))            THTP451
                    SELECT MEMBER=((HTTPDMTT,,R))          THTP451
                    SELECT MEMBER=((HTTPDSRV,,R))          THTP451
IEB167I  FOLLOWING MEMBER(S)  COPIED  FROM INPUT DATA SET REFERENCED BY HTTPLOAD -
IEB154I  ABEND0C1 HAS BEEN SUCCESSFULLY  COPIED
IEB154I  HTTPD    HAS BEEN SUCCESSFULLY  COPIED
IEB154I  HTTPDM   HAS BEEN SUCCESSFULLY  COPIED
IEB154I  HTTPDMTT HAS BEEN SUCCESSFULLY  COPIED
IEB154I  HTTPDSRV HAS BEEN SUCCESSFULLY  COPIED
IEB144I  THERE ARE 0000018 UNUSED TRACKS IN OUTPUT DATA SET REFERENCED BY LINKLIB
IEB149I  THERE ARE 0000019  UNUSED DIRECTORY BLOCKS IN OUTPUT DIRECTORY
IEB147I  END OF JOB -00 WAS HIGHEST SEVERITY CODE
```

The marker **R** after each member name is the replace option: the new module replaces the member of the same name. A load module with the attribute AC(1) keeps it, because SMP copies the module and does not link it again.

ACCEPT. The step runs although the APPLY ended with 0004. The install job lets it run for a code of 4 or less. The step does the same work on the distribution library:

```
     DATE 26.303  TIME 23:28:48      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                 PAGE 0001

HMA4240    HMASMP EXEC PARM = 'DATE=U'
 ACCEPT S(THTP451) DIS(WRITE) .
HMA4140    SMPACDS DIRECTORY SUCCESSFULLY LOADED FOR IN-STORAGE UPDATE OPERATIONS


HMA2240    SUCCESSFULLY DELETED MODULE ABEND0C1 ON AHTTPLOD LIBRARY
HMA2240    SUCCESSFULLY DELETED MODULE HTTPD ON AHTTPLOD LIBRARY
HMA2240    SUCCESSFULLY DELETED MODULE HTTPDM ON AHTTPLOD LIBRARY
HMA2240    SUCCESSFULLY DELETED MODULE HTTPDMTT ON AHTTPLOD LIBRARY
HMA2240    SUCCESSFULLY DELETED MODULE HTTPDSRV ON AHTTPLOD LIBRARY

HMA2380    COPY SUCCESSFUL - MOD=ABEND0C1 - LMOD=ABEND0C1 - LIBRARY=AHTTPLOD - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPD - LMOD=HTTPD - LIBRARY=AHTTPLOD - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDM - LMOD=HTTPDM - LIBRARY=AHTTPLOD - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDMTT - LMOD=HTTPDMTT - LIBRARY=AHTTPLOD - SYSMOD=THTP451 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=HTTPDSRV - LMOD=HTTPDSRV - LIBRARY=AHTTPLOD - SYSMOD=THTP451 - RETURN CODE=00
HMA2270    ACCEPT PROCESSING SUCCESSFULLY COMPLETED FOR SYSMOD THTP451
HMA3680    SMPACDS IN STORAGE DIRECTORY SUCCESSFULLY REWRITTEN
HMA2050    ACCEPT PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
     DATE 26.303  TIME 23:29:00      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                 PAGE 0002

SYSMOD STATUS REPORT FOR ACCEPT PROCESSING

  NOTE:  '-' INDICATES THE REQUISITE SYSMOD CONDITION IS NOT SATISFIED
         '*' INDICATES THE NON SATISFIED REQUISITE SYSMOD CONDITION IS BYPASSED


SYSMOD   STATUS    TYPE      FMID     REQUISITE AND SUPEDBY SYSMODS

THTP450  DELETED   FUNCTION  THTP450

THTP451  ACCEPTED  FUNCTION  THTP451
     DATE 26.303  TIME 23:29:00      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                 PAGE 0003

ELEMENT SUMMARY REPORT FOR ACCEPT PROCESSING


ELEM   ELEMENT   ELEM      CURRENT  CURRENT  MAC/SRC   DISTSRC   ASSEM    LOAD MOD  ---MOD DISTLIB---  SYSMOD   SYSMOD
TYPE   NAME      STATUS    FMID     RMID     DISTLIB   LIBRARY   NAMES                                 NAME     STATUS

MOD    ABEND0C1  ACCEPTED  THTP451  THTP451                                          AHTTPLOD          THTP451  ACCEPTED
                 NOT SEL                                                                              THTP450  DELETED

MOD    HTTPD     ACCEPTED  THTP451  THTP451                                          AHTTPLOD          THTP451  ACCEPTED
                 NOT SEL                                                                              THTP450  DELETED

MOD    HTTPDM    ACCEPTED  THTP451  THTP451                                          AHTTPLOD          THTP451  ACCEPTED
                 NOT SEL                                                                              THTP450  DELETED

MOD    HTTPDMTT  ACCEPTED  THTP451  THTP451                                          AHTTPLOD          THTP451  ACCEPTED
                 NOT SEL                                                                              THTP450  DELETED

MOD    HTTPDSRV  ACCEPTED  THTP451  THTP451                                          AHTTPLOD          THTP451  ACCEPTED
                 NOT SEL                                                                              THTP450  DELETED
     DATE 26.303  TIME 23:29:00      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                       PAGE 0004

DELETED FUNCTION REPORT FOR  ACCEPT PROCESSING

SYSMOD          DELETED THE
CAUSING THE     FOLLOWING SYSMODS
```

```
DELETION        TYPE       SYSMOD

THTP451         FUNCTION   THTP450

     DATE 26.303  TIME 23:29:00      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0005

HMA2050    HMASMP PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
```

Five **HMA2240** lines delete the old modules from **AHTTPLOD**, and five **HMA2380** lines copy the new ones in. The step ends with 00. The gate in the job lets the **ACCEPT** run for an **APPLY** code of 4 or less. A code of 8 or more stops it.

CLEANUP. The last step scratches the staging library:

```
IDCAMS  SYSTEM SERVICES                                        TIME: 23:29:00       10/30/26     PAGE   1

  DELETE HTTPD.HTTPLOAD NONVSAM SCRATCH PURGE
IDC0550I ENTRY (A) HTTPD.HTTPLOAD DELETED
IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 0

IDC0002I IDCAMS PROCESSING COMPLETE. MAXIMUM CONDITION CODE WAS 0
```

The return codes mean what they meant in Chapter 2, “Installing HTTPD” on page 5. The one difference is the 0004 of the **APPLY**. A 0004 in another step, or in a job that has no delete, needs a look at the messages. And a return code of 0 in every step does not show that the work happened. Count the **HMA2240** and **HMA2380** lines, and then verify (“Verifying the Upgrade”).

Verifying the Upgrade

Goal: the inventory and the library both show the new level.

The SMP record. Run the list job again. The control statements are the same as before the upgrade. The output:

```
     DATE 26.303  TIME 23:30:46      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0001

HMA4240    HMASMP EXEC PARM = 'DATE=U'
 RESETRC .
HMA2050    RESETRC PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


 LIST CDS SYSMOD(THTP450) .
     DATE 26.303  TIME 23:30:46      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0002

SMPCDS   SYSMOD ENTRIES


  NAME

THTP450   TYPE            = FUNCTION
          DELBY           = THTP451

     DATE 26.303  TIME 23:30:46      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0003

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


 LIST CDS LMOD(HTTPD) .
     DATE 26.303  TIME 23:30:46      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0004

SMPCDS  LOAD MODULE ENTRIES


  NAME

HTTPD     LASTUPD         = THTP451  TYPE=ADD
          SYSTEM LIBRARY  = LINKLIB
          LKED ATTRIBUTES = COPY

     DATE 26.303  TIME 23:30:46      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0005

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


 LIST CDS SYSMOD(THTP451) .
     DATE 26.303  TIME 23:30:46      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0006
```

```
SMPCDS   SYSMOD ENTRIES


  NAME

THTP451   TYPE            = FUNCTION
          STATUS          = REC  APP  ACC
          FMID            = THTP451
          JCLIN           = YES
          DATE/TIME REC   = 26.303  23:28:05
                    APP   = 26.303  23:28:43
                    ACC   = 26.303  23:28:57
          DELETE VER(001) = THTP450
          MOD             = ABEND0C1  HTTPD     HTTPDM    HTTPDMTT  HTTPDSRV

     DATE 26.303  TIME 23:30:46      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0007

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


HMA2050    HMASMP PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
```

Four facts changed. The old SYSMOD **THTP450** has lost its status, and it shows only the line **DELBY = THTP451**: the FMID that deleted it. The new SYSMOD **THTP451** has the status **REC APP ACC**, the line **DELETE VER(001) = THTP450** and the five modules. And the load module **HTTPD** has **LASTUPD = THTP451**. The old FMID stays in the inventory as a mark and is never used again.

The library. The IEHLIST listing after the upgrade:

```
                            SYSTEMS SUPPORT UTILITIES---IEHLIST                              PAGE    1
 DIRECTORY INFO FOR SPECIFIED PDS ON VOL TSO001
HTTPD.LINKLIB
           MEMBERS     TTRC     VARIABLE USER DATA ---(USER DATA AND TTRC ARE IN HEX)
          ABEND0C1  00400B2C  0040240000 000000C2C2 0152583060 0027B08800 00010000
          HTTPD     0042252C  0044010000 000000C2C2 047E480558 012FD88800 00010100
          HTTPDM    004B3B2C  004C150000 000000C2C2 016DD830C8 003BF88800 00010000
          HTTPDMTT  004E272C  004F180000 000000C2C2 01669833A0 002D888800 00010000
          HTTPDSRV  0051272C  00521A0000 000000C2C2 02442009F8 00BA888800 00010000

OF THE 00020 DIRECTORY BLOCKS ALLOCATED TO THIS PDS, 00019 ARE(IS) COMPLETELY UNUSED
```

Compare it with the listing before. The five names are the same. The **TTRC** and the user data of each member are not. The members are new copies. If the listing is the same as before, the copy did not happen.

The start. Start the server with the old procedure and the old Parmlib member:

```
S HTTPD
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=23.31.22
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/info
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/services/authenticate
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9AD080) TASK(162FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0ED7B0)   TCB(9AD828) TASK(176FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(174CC8)   TCB(9A0CF8) TASK(187FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(174C48)   TCB(9A09C8) TASK(198FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

The message **HTTPD000I** carries the version of the build, and **HTTPD001I** says that the server is ready. The three **HTTPD036I** lines are the routes of the member. If the version is not the one you installed, look at the **STEPLIB** (“Before You Start” on page 112). Send a request to the server, as in Chapter 3, “A First Request” on page 27.

Two things in the capture are new in 4.5.1. The stamp in **HTTPD000I** is shown as **nnnnnnn**: the package of the capture was built from the source of 4.5.1 before the release was published, and a start of the published package was not captured for this book. The line **RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM: USER:HTTPPRB** comes from the test of the security system at the start. It is expected on TK5 (RAKF 1.2.x) and MVS/CE (RAKF 2.0.0), and “Starting

HTTPD" in Chapter 2, "Installing HTTPD" on page 5 explains it. If the line **HTTPD042W** appears instead, the upgrade did not cause it: look at the tables of RAKF ("A Security System That Accepts Any User ID" on page 62).

What the SMP Upgrade Deletes

The delete covers the load modules of the predecessor and nothing more.

- **In the target library.** The five load modules **HTTPD**, **HTTPDSRV**, **HTTPDM**, **HTTPDMTT** and **ABEND0C1**, from **HTTPD.LINKLIB**, then the new ones are copied in (the **HMA2240** and **HMA2380** lines of the APPLY).
- **In the distribution library.** The same five, from **HTTPD.AHTTPLOD** (the same messages in the ACCEPT).
- **In the inventory.** The predecessor **THTP450** becomes a mark with the line **DELBY**. The listing shows it for the CDS, and the summary of the ACCEPT shows the same for the ACDS. The load module entries move to the new FMID.

The upgrade does not delete the procedure, the Parmlib member, the webroot disk, the RAKF entries and the mounts of UFSD. The sample library is not SMP's: the first step of the job scratches it, and the package brings the new one. The capture shows a library of only five members, so this book does not show what SMP does with a foreign member in **HTTPD.LINKLIB**. The library belongs to SMP. Keep other modules, such as mvsMF, in a library of their own.

What was verified. The FMID rule comes from the project file. The entry **[smp] prefix = "THTP"** makes the build derive the FMID and the predecessor from the version, and the project file of 4.5.1 names **fmid = "THTP451"** and **delete = ["THTP450"]** itself, because 4.5.1 is a patch release. A package build of 4.5.1 produced **++FUNCTION(THTP451)** and **++VER(Z038) DELETE(THTP450)**. The other rows of the table in "One Release at a Time" on page 111 come from the project file and from the measured installs of the releases. The step output in this chapter is from the upgrade 4.5.0 to 4.5.1, where **THTP451** deleted **THTP450**: **THTP451** is a **FUNCTION** with the line **DELETE VER(001) = THTP450**, **THTP450** shows **DELBY = THTP451** and no modules, and **HTTPD.LINKLIB** and **HTTPD.AHTTPLOD** each hold the five members of the package. Earlier upgrades, from 4.3.0 to 4.4.0 and from 4.4.0 to 4.5.0, had the same shape.

Upgrading from HTTPD 4.0.x

The libraries of 4.0.x carry the version in their names. One of them is **HTTPD.V4R0M1.LINKLIB**. From 4.1.0 on, the names carry no version. The hop from 4.0.x to 4.1.0 is the one where the names change, and so it is the one hop that needs the allocation job. Every hop after it does not.

1. Take the package of 4.1.0. Run its allocation job. It creates **HTTPD.LINKLIB** and **HTTPD.AHTTPLOD** (Chapter 2, "Installing HTTPD" on page 5).
2. Make the new **HTTPD.LINKLIB** available to the started task. Put it into the **STEPLIB** of the procedure in place of the old library, and make it authorized, with an APF entry or through RAKF ("Authority" in Chapter 2, "Installing HTTPD" on page 5). The volume matters: the APF entry names it.
3. Send the XMIT files, edit the install job and submit it, as in "Upgrading by One Release" on page 114. The **APPLY** step deletes **THTP400** from the new, empty library without a complaint, copies the five modules in, and ends with 0004. The **ACCEPT** step ends with 0000.
4. Install 4.2.0, 4.3.0, 4.4.0, 4.5.0 and 4.5.1 in turn, each as in "Upgrading by One Release" on page 114. These hops skip the allocation job.

The install of 4.1.0 leaves the old libraries as they were. After the crossing, check these points:

- **mvsMF.** The old library held the module **MVSMF**. The new **HTTPD.LINKLIB** does not, and it should not. Install mvsMF from its own package into a library of its own, and put the library into the **STEPLIB**

behind **HTTPD.LINKLIB** or into the link list. Without it, a request for a route of mvsMF gets an error response (Chapter 2, “Installing HTTPD” on page 5).

- **The old libraries.** The data sets **HTTPD.V4R0M*.*** stay on the volume. Scratch them yourself, when the new server runs and nothing names them any more. List the data sets first, and check the **STEPLIB**, the APF list and the link list. Look at the names before you scratch: an old webroot data set may carry a name of the same pattern. The document root disk is not a library of the product. It holds your content and the UFSD mount names it. Do not scratch it with them.
- **The procedure.** Compare it with the sample (“After the Upgrade: Review Your Own Files” on page 124).

The JES2 DDs. The procedure of 4.0.0 had no **HASPCKPT** and **HASPACE1** statements. mvsMF 1.3.0 and earlier open the two data sets by their DD names in the task of the server, and without the statements they answer a job list with status 500 and a request for one job with status 404. The sample procedure of 4.0.1 has the statements back. From 4.5.0 on, the sample procedure has them as comment lines, because mvsMF 1.4.0 and later do not need them. A procedure that you copied from 4.0.0 needs them active when you run mvsMF 1.3.0 or earlier or a JES2 with a prefix other than **SYS1** (Chapter 2, “Installing HTTPD” on page 5).

A clean cut instead. If you have all the packages, but would rather not install five releases in turn, you can remove HTTPD with Chapter 11, “Removing HTTPD” on page 127 and then install the current release with Chapter 2, “Installing HTTPD” on page 5. The removal job frees the FMIDs and scratches the libraries, and the installation is the one for a new system. Keep a copy of your procedure, your Parmlib member and your webroot disk first, because Chapter 11, “Removing HTTPD” on page 127 treats the parts that SMP never owned in a step of its own. This way has not been run for this book as a whole.

Upgrading from HTTPD 3.x

HTTPD 3.x was not an SMP install, and the step from 3.x to 4.x is a migration and not an upgrade. The new server is a new install (Chapter 2, “Installing HTTPD” on page 5). It has a new procedure and a new configuration. This section lists what a reader of the old documentation needs to know.

The data sets of the new install have the names of Chapter 2, “Installing HTTPD” on page 5. If a data set of the old installation has one of those names, the allocation job cannot create it. Rename the old data set first. This book does not show the rename.

The configuration. The Lua configuration file is gone. The server reads the Parmlib member that the DD name **HTTPPRM** of the procedure names, with lines of the form **KEYWORD=VALUE**. Start from the sample member of the package (**HTTPPRM0** in **HTTPD.SAMPLIB**) and move your settings into it. “The HTTPPRM Keywords” in the *HTTPD Reference* lists every keyword. Chapter 4, “Configuring HTTPD” on page 33 explains the file.

The routes. A program route is a line **MOD=PROGRAM /path** with an optional **AUTH=mode**. A route without **AUTH=** is public. Write the **AUTH=** on every route (Chapter 5, “Routes and Server Modules” on page 47, Chapter 6, “Authentication and Security” on page 55). The 3.x keyword **LOGIN** has no place any more. HTTPD ends the start with **HTTPD420E** and CC 0008 if the value would have required a login (**HTTPD048E**), because the routes would be open without it. An empty value, or the value **NONE**, gives **HTTPD048W** and the start goes on. HTTPD does not turn the value into route policies: change the routes (“Routes and Authentication Policy” in the *HTTPD Reference*).

The parameter CONFIG. A procedure that passes **PARM='CONFIG=...'** still starts. HTTPD ignores the parameter and writes **HTTPD024W** and **HTTPD024I**, which says to use **S HTTPD,M=MEMBER** to choose a different member. Remove the parameter from the procedure. The command **F HTTPD,D CONFIG** shows with **HTTPD133I** which member HTTPD read.

Static files. The files are served from a UFS file system through UFSD, and from nowhere else. A data set as document root is not supported. Put the files into the webroot and have UFSD mount it (Chapter 2, “Installing HTTPD” on page 5, Chapter 7, “Serving Files” on page 69).

What is gone. These features of 3.x are not in 4.x:

Table 35. Features of HTTPD 3.x that are not in 4.x

Feature	What to use now
The embedded FTP server	The standalone FTPD, as a started task of its own.
MQTT telemetry	Nothing replaces it. The server writes SMF records.
The Lua engine	HTTPLUA, a module of its own project that you add as a route.
The demo modules	Nothing. The module ABEND0C1 is still in the package, and it is inert unless a route names it.
A data set as document root	A UFS file system, through UFSD.
The in-memory statistics	SMF records. The keyword **SMF** has the default **NONE**, so set it if you want records (“The HTTPPRM Keywords” and the SMF chapter of the Reference).
HTTPJES2 and HTTPDSL	The dataset and jobs APIs of mvsMF.

A route that names **HTTPJES2** or **HTTPDSL** loads nothing. The start writes **HTTPD440W** for a module that is not found, and a request to the route gets status 503 with **HTTPD908E** on the console. Delete the line.

The DD names. Keep the statements **HASPCKPT** and **HASPACE1** in the procedure when you run mvsMF 1.3.0 or earlier. A module runs in the task of the server, and these releases of mvsMF open the two data sets by DD name. From mvsMF 1.4.0 on, the statements are optional: the jobs API allocates **SYS1.HASPCKPT** and **SYS1.HASPACE** itself when the procedure has no DD. They stay needed on a JES2 whose prefix is not **SYS1**. Leaving them in is safe.

What is new. HTTP/1.1 with persistent connections and chunked transfer, a strict parser, and routing by the extension of a file with **MOD=** (Chapter 5, “Routes and Server Modules” on page 47).

The checklist. In this order:

1. Install the 4.x package (Chapter 2, “Installing HTTPD” on page 5).
2. Copy the sample Parmlib member and the sample procedure, and edit them for your system. Remove **CONFIG=** from the procedure.
3. Start UFSD, and mount the webroot.
4. Move your routes into the member, each with an **AUTH=**.
5. Install FTPD as a separate task, if you used the FTP server.
6. Decide on **SMF**.
7. Start HTTPD and test with a client (Chapter 3, “A First Request” on page 27).

After the Upgrade: Review Your Own Files

The upgrade replaces the sample library and leaves your copies alone. Nothing in an old member is wrong because a newer sample exists, but a newer sample can contain a thing that your member lacks. Compare your member with the new new sample **HTTPD.SAMPLIB(HTTPPRM0)**, and your procedure with

HTTPD.SAMPLIB(HTTPD), by eye or with the compare tool of your system. These are the changes that a reader of an older release needs to look for. The first two matter for the safety of the system.

- **The console route of mvsMF.** Since 4.4.0 the sample member has the route **MOD=MVSMF /zosmf/restconsoles/*** with **AUTH=TOKEN RES=FACILITY:MVSMF.CONSOLE**, before the route **/zosmf/***. The console service issues operator commands for every caller who can log in, and mvsMF up to 1.3.0 checks nothing itself. From mvsMF 1.4.0 on it checks READ on the same profile too and refuses every user until the profile exists; the route stays as a first layer. The route must come before the catch-all, because the first match wins. A member that has only the line **/zosmf/*** has no check on the console service. Add the route, and define the profile in RAKF (Chapter 6, "Authentication and Security" on page 55). Without a profile the check permits everybody under RAKF, and **HTTPD425W** says so at the start.
- **The login checks of 4.5.1.** Nothing in your files has to change. The server now compares the user that the security system returns with the userid that was typed, and refuses the login on a mismatch. It refuses every login of the started-task account. At the start it tests the security system once with a logon that must fail. On TK5 and MVS/CE this writes **RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM: USER:HTTPPRB** to the console, and the line is expected. On **HTTPD042W** the server refuses every login until the security software runs correctly and HTTPD has been restarted, while routes without a login still answer ("A Security System That Accepts Any User ID" on page 62).
- **The AUTH values.** A route without **AUTH=** is public. Open the member and read every **MOD** and **LOC** line for it.
- **A route whose profile is missing.** From 4.5.0 the keyword **RES_NOPROFILE=DENY** makes a route with **RES=** refuse every request while its profile does not exist, instead of letting every user who is logged in pass. The sample member has it as a comment line, and without it nothing changes. Turn it on when the start prints no **HTTPD425W** ("Refusing a Route Whose Profile Is Missing" on page 66).
- **Resident modules.** From 4.4.0 the keyword **RESIDENT** has the default **YES**: HTTPD loads each module of a route once at the start and holds it until the end. A new level of a module then takes effect at the next restart. This matters when you install a new mvsMF. With **RESIDENT=NO** every request fetches the module. A module that is neither reentrant nor serially reusable is not held, and the start says so with **HTTPD441W**.
- **The procedure.** It names the library in the **STEPLIB** and the Parmlib member in the **HTTPPRM** statement. From 4.0.1 on it had the two JES2 data sets, and from 4.5.0 on it has them as comment lines. Your own active statements keep working, so you need not change them. If you run mvsMF 1.4.0 or later on a JES2 with the prefix **SYS1**, you may comment them out. If you keep them, check that **UNIT** and **VOL=SER** match your own JES2 procedure.

"The HTTPPRM Keywords" in the *HTTPD Reference* has a section on the keywords that are no longer valid. A member from an earlier release starts, except for a retired **LOGIN** that required a login.

Problems at Upgrade

Chapter 12, "Troubleshooting" on page 141 has the general table. These are the ones that belong to an upgrade.

Table 36. Problems at upgrade

What you see	What it is and what to do
The **APPLY** step ends with 0004 and the **ACCEPT** ran.	The normal result. The deleted FMID has no backup entry (**HMA2461**).
Every step ends with 0000, but the **APPLY** has no **HMA2240** and no **HMA2380**.	A release was skipped. The SYSMOD does not own the modules and copied nothing. The library still holds the old modules. List the level ("Before You Start" on page 112) and do not go on.

The job waits, and the console shows **IEF861I** and **IEF099I**.	HTTPD still runs and has the library open. Stop it with **P HTTPD** and wait for **$HASP395 HTTPD ENDED**.
A RECEIVE step fails.	The upload was not binary, or the record format is wrong. Send the file again (Chapter 2, “Installing HTTPD” on page 5).
The version in **HTTPD000I** is the old one.	The **STEPLIB** names another library first, or an old copy of the module is in the link list. Check both.
The start ends with **HTTPD420E** and CC 0008.	A route lost its policy: a retired **LOGIN** (**HTTPD048E**), an unknown **AUTH=** value or a route that cannot be registered. Read all the messages before it.
RAKF0004 ... at the start. **USER:HTTPPRB**	Expected on TK5 and MVS/CE since 4.5.1: the test of the security system. No user **HTTPPRB** needs to exist.
HTTPD042W at the start, and every login is refused.	The security software accepts any user ID. On RAKF 1.2.x the user table is missing or was rejected at the IPL. Correct the tables, IPL, and restart HTTPD (“A Security System That Accepts Any User ID” on page 62). Until then the server refuses every login.
HTTPD440W at the start.	A route names a module that HTTPD does not find in the **STEPLIB** or the link list. After a move from 4.0.x, mvsMF is the likely one.
HTTPD002I says **AUTHORIZED BY SVC**, and you expect **AUTHORIZED BY LIBRARY**. Or **HTTPD012E** at the start.	The library is not the one the APF entry names, for example the new library of a move from 4.0.x on another volume. HTTPD falls back to the SVC of RAKF. Without RAKF it ends with **HTTPD012E** (“Authority” in Chapter 2, “Installing HTTPD” on page 5).
A job list of mvsMF answers 500. A request for one job or its spool answers 404 “job not found” although the job exists.	mvsMF 1.3.0 or earlier, or a JES2 with a prefix other than **SYS1**, and the procedure lacks **HASPCKPT** and **HASPACE1**. Add the two statements (Chapter 2, “Installing HTTPD” on page 5).

To take HTTPD out of the system, follow Chapter 11, “Removing HTTPD” on page 127.

Chapter 11. Removing HTTPD

This chapter takes HTTPD off an MVS 3.8j system and frees its FMID. It covers the SMP inventory, the load and distribution libraries, and the parts that SMP never owned: the procedure, the Parmlib member, the RAKF entries and the document root disk. The last section shows how to check that the work happened.

If you want a newer release, you are in the wrong chapter. A release deletes its predecessor when it is installed (Chapter 10, "Upgrading HTTPD" on page 111). Running this chapter first would scratch the libraries that the new release installs into.

Note: Which release the sessions come from. The sessions in this chapter were measured on release 4.4.0 (the FMID **THTP440**) on a TK5 system. The FMID of a release is **THTP** and the three digits of its version. HTTPD 4.5.0 is **THTP450**, and it deletes **THTP440**. HTTPD 4.5.1 is **THTP451**, and it deletes **THTP450** the rule is in the *HTTPD Reference*, in "The Started Task and the Package", section "The FMID". The job in this chapter is the 4.4.0 job. "What Changes for HTTPD 4.5.0" on page 136 tells what changes for 4.5.0, and what the 4.5.0 form of the job did on a TK5 system. "What Changes for HTTPD 4.5.1" on page 136 tells what changes for 4.5.1, and what the 4.5.1 form of the job did on an MVS/CE system that held 4.5.0 and on a TK5 system that held 4.5.1. If you installed a development build of HTTPD, read "A Development Build with a Later FMID" on page 137 too.

What HTTPD Put on the System

Table 37. What release 4.4.0 installed

Item	Name
FMID	**THTP440**
Load modules	**HTTPD**, **HTTPDSRV**, **HTTPDM**, **HTTPDMTT**, **ABEND0C1**
Target library	**HTTPD.LINKLIB**
Distribution library	**HTTPD.AHTTPLOD**
Sample library	**HTTPD.SAMPLIB** (SMP does not own it)
Document root disk	**HTTPD.WEBROOT.UFS** (SMP does not own it)

The names of the data sets carry no version, so there is one HTTPD installation on a system, whichever release put it there. The staging library **HTTPD.HTTPLOAD** is not on the list. The last step of the install job scratches it, but only if the ACCEPT ended with return code 0. If a failed install left it, scratch it as well.

Earlier releases left their FMIDs behind. One release has one FMID, and an id is spent once. Each release deletes its predecessor, and SMP keeps the id of the deleted function in both zones as a mark. SMP lists the mark as **TYPE FUNCTION** with **DELBY** naming the later id, and its **LIST** of such an id ends with return code 0, not 4. A system that took four upgrades carries four of these marks. A removal that names only the current release and its predecessor leaves the older marks in the inventory for good. The SYSMOD of 4.5.0 carries **DELETE(THTP440)**, so a system that took 4.5.0 as its first install holds **THTP440** as a mark as well, with **DELBY = THTP450** in the CDS and in the ACDS. The same holds for **THTP450** on a system that took 4.5.1 as its first install, with **DELBY = THTP451**. The job in this chapter

therefore names every FMID that HTTPD has ever spent, from **THTP400** to **THTP440** (to **THTP450** in the form of “What Changes for HTTPD 4.5.0” on page 136, to **THTP451** in the form of “What Changes for HTTPD 4.5.1” on page 136). You need not know which releases your system has seen.

Step 1: Stop the Server

Stop the started task from the console. The server writes its shutdown messages and the statistics of the run, and the task ends.

```
P HTTPD
HTTPD098I HTTPD SHUTTING DOWN
HTTPD040I WAITING FOR SOCKET THREAD TO TERMINATE (1)
HTTPD060I SHUTDOWN SOCKET THREAD    TCB(9AD080) TASK(15EFC8) STACKSIZE(32768)
HTTPD060I SHUTDOWN WORKER(170C48)   TCB(9A09C8) TASK(194FC8) STACKSIZE(65536)
HTTPD060I SHUTDOWN WORKER(170CC8)   TCB(9A0CF8) TASK(183FC8) STACKSIZE(65536)
HTTPD060I SHUTDOWN WORKER(0E97B0)   TCB(9AD828) TASK(172FC8) STACKSIZE(65536)
HTTPD416I STATS: 39 REQUESTS, 1 ERRORS, 156405 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
$HASP395 HTTPD    ENDED
IEF404I HTTPD - ENDED - TIME=07.07.17
$HASP150 HTTPD    ON PRINTER2       158 LINES
$HASP160 PRINTER2 INACTIVE - CLASS=Z
$HASP250 HTTPD    IS PURGED
```

HTTPD098I starts the stop and **HTTPD099I** ends it. Between them the server writes one line for the listener thread and one for each worker. The messages are in the *HTTPD Reference*, in ‘Messages’, section “A Healthy Start and Stop”. **$HASP250 HTTPD IS PURGED** shows that JES2 has dropped the task.

If you run a second HTTPD on another port, stop that one as well. It loads from a library that step 4 scratches.

Step 2: Cut the FMIDs out of the SMP Inventory

The job below edits the CDS (what is applied) and the ACDS (what is accepted). It touches no library. It is long on purpose. This is the job as it ran on the measured system. **SMPAPP** is the SMP procedure of TK5. If your system names it differently, change the two **EXEC** statements.

```
//HTTPDUCL JOB (SYS),'HTTPD UNINSTALL',
//            CLASS=A,MSGCLASS=H,MSGLEVEL=(1,1),
//            REGION=4096K
//UCLIN   EXEC SMPAPP
//SMPCNTL  DD  *
 UCLIN CDS .
  DEL SYSMOD(THTP440) MOD(HTTPD) .
 ENDUCL .
 RESETRC .
 UCLIN CDS .
  DEL SYSMOD(THTP440) MOD(HTTPDSRV) .
 ENDUCL .
 RESETRC .
 UCLIN CDS .
  DEL SYSMOD(THTP440) MOD(HTTPDM) .
 ENDUCL .
 RESETRC .
 UCLIN CDS .
  DEL SYSMOD(THTP440) MOD(HTTPDMTT) .
 ENDUCL .
 RESETRC .
 UCLIN CDS .
  DEL SYSMOD(THTP440) MOD(ABEND0C1) .
 ENDUCL .
 RESETRC .
 UCLIN CDS .
  DEL MOD(HTTPD) .
 ENDUCL .
 RESETRC .
 UCLIN CDS .
  DEL MOD(HTTPDSRV) .
 ENDUCL .
 RESETRC .
 UCLIN CDS .
  DEL MOD(HTTPDM) .
 ENDUCL .
 RESETRC .
 UCLIN CDS .
```

```
 DEL MOD(HTTPDMTT) .
ENDUCL .
RESETRC .
UCLIN CDS .
 DEL MOD(ABEND0C1) .
ENDUCL .
RESETRC .
UCLIN CDS .
 DEL LMOD(HTTPD) .
ENDUCL .
RESETRC .
UCLIN CDS .
 DEL LMOD(HTTPDSRV) .
ENDUCL .
RESETRC .
UCLIN CDS .
 DEL LMOD(HTTPDM) .
ENDUCL .
RESETRC .
UCLIN CDS .
 DEL LMOD(HTTPDMTT) .
ENDUCL .
RESETRC .
UCLIN CDS .
 DEL LMOD(ABEND0C1) .
ENDUCL .
RESETRC .
UCLIN CDS .
 DEL SYSMOD(THTP440) .
ENDUCL .
RESETRC .
UCLIN CDS .
 DEL SYSMOD(THTP430) .
ENDUCL .
RESETRC .
UCLIN CDS .
 DEL SYSMOD(THTP420) .
ENDUCL .
RESETRC .
UCLIN CDS .
 DEL SYSMOD(THTP410) .
ENDUCL .
RESETRC .
UCLIN CDS .
 DEL SYSMOD(THTP400) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL SYSMOD(THTP440) MOD(HTTPD) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL SYSMOD(THTP440) MOD(HTTPDSRV) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL SYSMOD(THTP440) MOD(HTTPDM) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL SYSMOD(THTP440) MOD(HTTPDMTT) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL SYSMOD(THTP440) MOD(ABEND0C1) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL MOD(HTTPD) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL MOD(HTTPDSRV) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL MOD(HTTPDM) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL MOD(HTTPDMTT) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL MOD(ABEND0C1) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL SYSMOD(THTP440) .
ENDUCL .
RESETRC .
UCLIN ACDS .
 DEL SYSMOD(THTP430) .
ENDUCL .
RESETRC .
UCLIN ACDS .
```

```
  DEL SYSMOD(THTP420) .
 ENDUCL .
 RESETRC .
 UCLIN ACDS .
  DEL SYSMOD(THTP410) .
 ENDUCL .
 RESETRC .
 UCLIN ACDS .
  DEL SYSMOD(THTP400) .
 ENDUCL .
 RESETRC .
/*
//LIST    EXEC SMPAPP
//SMPCNTL  DD  *
 RESETRC .
 LIST CDS  SYSMOD(THTP440) .
 LIST CDS  SYSMOD(THTP430) .
 LIST CDS  SYSMOD(THTP420) .
 LIST CDS  SYSMOD(THTP410) .
 LIST CDS  SYSMOD(THTP400) .
 LIST ACDS SYSMOD(THTP440) .
 LIST ACDS SYSMOD(THTP430) .
 LIST ACDS SYSMOD(THTP420) .
 LIST ACDS SYSMOD(THTP410) .
 LIST ACDS SYSMOD(THTP400) .
/*
//
```

Save the job as **`uclin.jcl`** and submit it the way the install job is submitted (“Editing and Submitting the Install Job” on page 12). With FTP:

```
$ ftp -v -n mvs.example.org 2121
Connected to mvs.example.org.
220 MVS 3.8j FTPD Server
ftp> user HERC01
331 Send password please.
Password:
230 HERC01 is logged on.  Working directory is "HERC01.".
Remote system type is MVS.
ftp> progress off
Progress bar off.
ftp> quote site filetype=jes
200 SITE command was accepted
ftp> put uclin.jcl
local: uclin.jcl remote: uclin.jcl
229 Entering Extended Passive Mode (|||22000|)
125 Sending Job to JES internal reader FIXrecfm 80
250-It is known to JES as JOBnnnnn
250 Transfer completed successfully.
2802 bytes sent in 00:00 (57.47 KiB/s)
ftp> quit
221 Quit command received. Goodbye.
```

The job has 35 **`DEL`** statements, 20 against the CDS and 15 against the ACDS. In each zone they fall into these groups.

1. Five **`DEL SYSMOD(THTP440) MOD(...)`** statements. They take the five modules out of the entry of the FMID.
2. Five **`DEL MOD(...)`** statements. They remove the module entries.
3. Five **`DEL LMOD(...)`** statements, in the CDS only. They remove the load module entries.
4. Five **`DEL SYSMOD(...)`** statements, one for each id from **`THTP440`** down to **`THTP400`**. They remove the function entries, and with them the marks of the deleted releases.

Every **`DEL`** stands in a **`UCLIN`**/**`ENDUCL`** pair of its own, and a **`RESETRC .`** follows every **`ENDUCL .`**. The reason is the way SMP treats a **`DEL`** for an entry that the zone does not hold.

- The **`DEL`** is an error. SMP writes **`HMA2532`** (the entry does not exist) and **`HMA2560`** (update not complete), and the block ends with return code 8.
- An error stays in force for the rest of the step. Every later **`UCLIN`** of the same step is answered with **`HMA2293`** (control statement ignored due to previous error), and its **`DEL`** statements are discarded without a trace.
- **`RESETRC .`** after each block lifts the error. A new **`EXEC`** step starts clean as well.

The job needs this because it names every FMID that HTTPD has spent, and no system has seen all of them.

Reading the return codes

On the measured system, the job log says this.

```
                                          J E S 2   J O B   L O G
 7.07.25 JOB   nn  $HASP373 HTTPDUCL STARTED - INIT  1 - CLASS A - SYS TK5R
 7.07.25 JOB   nn  IEF403I HTTPDUCL - STARTED - TIME=07.07.25
 7.07.58 JOB   nn  IEFACTRT - Stepname  Procstep  Program   Retcode
 7.07.58 JOB   nn  HTTPDUCL   UCLIN     HMASMP    HMASMP    RC= 0008
 7.07.58 JOB   nn  HTTPDUCL   LIST      HMASMP    HMASMP    RC= 0004
 7.07.58 JOB   nn  IEF404I HTTPDUCL - ENDED - TIME=07.07.58
 7.07.58 JOB   nn  $HASP395 HTTPDUCL ENDED
```

The **UCLIN** step ends with return code 8 and the **LIST** step with
4. Both are the expected result on this system. The system held **THTP440**,

THTP430 and the mark of **THTP420**, but it had never seen **THTP410** or **THTP400**. The next screen counts the messages in the SMP output of the **UCLIN** step.

```
$ grep -E 'HMA2(532|550|560)' uclin-smp-uclin.txt
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR HTTPD
HMA2550    UPDATE COMPLETE FOR HTTPDSRV
HMA2550    UPDATE COMPLETE FOR HTTPDM
HMA2550    UPDATE COMPLETE FOR HTTPDMTT
HMA2550    UPDATE COMPLETE FOR ABEND0C1
HMA2550    UPDATE COMPLETE FOR HTTPD
HMA2550    UPDATE COMPLETE FOR HTTPDSRV
HMA2550    UPDATE COMPLETE FOR HTTPDM
HMA2550    UPDATE COMPLETE FOR HTTPDMTT
HMA2550    UPDATE COMPLETE FOR ABEND0C1
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR THTP430
HMA2550    UPDATE COMPLETE FOR THTP420
HMA2532 ** SYSMOD ENTRY THTP410 TO BE DELETED DOES NOT EXIST
HMA2560    UPDATE PROCESSING TERMINATED - UPDATE NOT COMPLETE
HMA2532 ** SYSMOD ENTRY THTP400 TO BE DELETED DOES NOT EXIST
HMA2560    UPDATE PROCESSING TERMINATED - UPDATE NOT COMPLETE
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR HTTPD
HMA2550    UPDATE COMPLETE FOR HTTPDSRV
HMA2550    UPDATE COMPLETE FOR HTTPDM
HMA2550    UPDATE COMPLETE FOR HTTPDMTT
HMA2550    UPDATE COMPLETE FOR ABEND0C1
HMA2550    UPDATE COMPLETE FOR THTP440
HMA2550    UPDATE COMPLETE FOR THTP430
HMA2550    UPDATE COMPLETE FOR THTP420
HMA2532 ** SYSMOD ENTRY THTP410 TO BE DELETED DOES NOT EXIST
HMA2560    UPDATE PROCESSING TERMINATED - UPDATE NOT COMPLETE
HMA2532 ** SYSMOD ENTRY THTP400 TO BE DELETED DOES NOT EXIST
HMA2560    UPDATE PROCESSING TERMINATED - UPDATE NOT COMPLETE
```

Of the 35 statements, 31 ended with **HMA2550** (update complete). Four ended with **HMA2532** and **HMA2560**: **THTP410** and **THTP400**, in each of the two zones. There is no **HMA2293**, because each block has its own **RESETRC .**.

A return code of 8 here does not say that the removal failed, and a return code of 0 would not say that it worked. The **LIST** of the next step tells. If a **DEL** that you expected to work shows an error other than **HMA2532**, read the complete SMP output of the **UCLIN** step. It holds the messages between the statements.

Step 3: Read the LIST

The second step of the job lists the five ids in both zones. This listing is the result of the removal, not the return code of the job. This is the output on the measured system.

```
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0001

HMA4240    HMASMP EXEC PARM = 'DATE=U'
 RESETRC .
HMA2050    RESETRC PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00


 LIST CDS  SYSMOD(THTP440) .
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0002

LIST     SELECT SUMMARY REPORT FOR SMPCDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING


 TYPE        NAME
 SYSMOD      THTP440
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0003

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04


 LIST CDS  SYSMOD(THTP430) .
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0004

LIST     SELECT SUMMARY REPORT FOR SMPCDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING


 TYPE        NAME
 SYSMOD      THTP430
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0005

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04


 LIST CDS  SYSMOD(THTP420) .
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0006

LIST     SELECT SUMMARY REPORT FOR SMPCDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING


 TYPE        NAME
 SYSMOD      THTP420
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0007

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04


 LIST CDS  SYSMOD(THTP410) .
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0008

LIST     SELECT SUMMARY REPORT FOR SMPCDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING


 TYPE        NAME
 SYSMOD      THTP410
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0009

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04


 LIST CDS  SYSMOD(THTP400) .
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0010

LIST     SELECT SUMMARY REPORT FOR SMPCDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING


 TYPE        NAME
 SYSMOD      THTP400
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0011

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04


 LIST ACDS SYSMOD(THTP440) .
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0012

LIST     SELECT SUMMARY REPORT FOR SMPACDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING


 TYPE        NAME
 SYSMOD      THTP440
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0013

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04
```

```
 LIST ACDS SYSMOD(THTP430) .
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0014

LIST     SELECT SUMMARY REPORT FOR SMPACDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING


 TYPE        NAME
 SYSMOD      THTP430
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0015

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04


 LIST ACDS SYSMOD(THTP420) .
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0016

LIST     SELECT SUMMARY REPORT FOR SMPACDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING


 TYPE        NAME
 SYSMOD      THTP420
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0017

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04


 LIST ACDS SYSMOD(THTP410) .
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0018

LIST     SELECT SUMMARY REPORT FOR SMPACDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING


 TYPE        NAME
 SYSMOD      THTP410
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0019

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04


 LIST ACDS SYSMOD(THTP400) .
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0020

LIST     SELECT SUMMARY REPORT FOR SMPACDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING


 TYPE        NAME
 SYSMOD      THTP400
     DATE 26.300  TIME 07:07:58      HMASMP LVL 04.48 MESSAGES AND CONTROL STATEMENTS                      PAGE 0021

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04


HMA2050    HMASMP PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04
```

Each of the ten **`LIST`** statements must answer with the stanza “THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING”, naming the id, and the step ends with return code 4. Return code 4 and an empty list mean that the FMID is free. Read all ten. The CDS and the ACDS are separate inventories, and an id that is gone from one and present in the other is not free.

A stanza that lists **`TYPE FUNCTION`** with **`DELBY`** is an id that is still there. SMP answers it with return code 0, so it does not lift the return code of the step above the 4 that you expect. This is why you read the stanzas and not the return code of the step.

Step 4: Scratch the Libraries

Note: List the members first. **`HTTPD.LINKLIB`** is the STEPLIB of the started task, and it holds whatever was copied into it. The package puts five modules there. A server module of another product, for example **`MVSMF`**, can be in the same library. SMP does not know such a module, and the scratch takes it with it. List the members (ISPF 3.4 or IEHLIST) and install again what belongs to something else.

The UCLIN job edited the inventory only. Both libraries are still catalogued, and the allocation job of a later install would fail on them. This job scratches them.

```
//HTTPDDEL JOB (SYS),'HTTPD SCRATCH',
//            CLASS=A,MSGCLASS=H,MSGLEVEL=(1,1),
//            REGION=4096K
//SCRATCH EXEC PGM=IDCAMS
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  *
  DELETE HTTPD.LINKLIB  NONVSAM SCRATCH PURGE
  DELETE HTTPD.AHTTPLOD NONVSAM SCRATCH PURGE
/*
//
```

The job log shows that the step ran.

```
                                          J E S 2   J O B   L O G
 7.09.00 JOB   nn  $HASP373 HTTPDDEL STARTED - INIT  1 - CLASS A - SYS TK5R
 7.09.00 JOB   nn  IEF403I HTTPDDEL - STARTED - TIME=07.09.00
 7.09.00 JOB   nn  IEFACTRT - Stepname  Procstep  Program   Retcode
 7.09.00 JOB   nn  HTTPDDEL   SCRATCH             IDCAMS    RC= 0000
 7.09.00 JOB   nn  IEF404I HTTPDDEL - ENDED - TIME=07.09.00
 7.09.00 JOB   nn  $HASP395 HTTPDDEL ENDED
```

The return code says that the job ran. IDCAMS says what it did.

```
IDCAMS  SYSTEM SERVICES                                          TIME: 07:09:00       10/27/26     PAGE   1

  DELETE HTTPD.LINKLIB  NONVSAM SCRATCH PURGE
IDC0550I ENTRY (A) HTTPD.LINKLIB DELETED
IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 0

  DELETE HTTPD.AHTTPLOD NONVSAM SCRATCH PURGE
IDC0550I ENTRY (A) HTTPD.AHTTPLOD DELETED
IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 0

IDC0002I IDCAMS PROCESSING COMPLETE. MAXIMUM CONDITION CODE WAS 0
 !! END OF JES SPOOL FILE !!
```

IDC0550I names the entry that was deleted, once for each library. **HTTPD.SAMPLIB** is not in this job. The next install scratches it by itself (the step **DELOLD** of the install job). Scratch it too if you want a clean system, once you have copied what you want out of it.

If you put the library into the APF list (the entry in **SYS1.PARMLIB(IEAAPF00)**), remove the entry for **HTTPD.LINKLIB** too.

Step 5: The Parts that SMP Never Owned

The UCLIN job and the scratch job do not touch these. They are yours.

The procedure. Delete the member **HTTPD** from the procedure library that you copied it to, for example **SYS2.PROCLIB(HTTPD)**.

The Parmlib member. Delete the member that the procedure names in its symbol **M** (**HTTPPRM0** in the sample) from the data set that its symbol **D** names. The *HTTPD Reference* shows the procedure in "The Started Task and the Package", section "The Procedure". Keep a copy if you want to look at your routes again later.

The RAKF entries. The user of the started task (**HTTPD** in the group **USER** in the sample procedure) has a line in **SYS1.SECURE.CNTL(USERS)**. The routes with **RES=** can have profiles in **SYS1.SECURE.CNTL(PROFILES)**. Chapter 6, "Authentication and Security" on page 55 shows the definitions that HTTPD needs. Take care in this step.

- Remove the line of the HTTPD user from **USERS** only if no other product uses the user. Remove a profile only if no other product uses it. A profile of a route can belong to a server module of another product, for example mvsMF.

- Reload the tables with the RAKF started tasks, **`S RAKFPROF`** for the profiles and **`S RAKFUSER`** for the users.

The document root disk. The mount of **`HTTPD.WEBROOT.UFS`** belongs to UFSD, not to HTTPD. Remove the **`MOUNT`** statement that names it from the Parmlib member of UFSD. Unmount the disk or stop UFSD before you scratch the data set (see the documentation of UFSD). Then scratch it with IDCAMS, as you scratched the libraries.

```
    DELETE HTTPD.WEBROOT.UFS NONVSAM SCRATCH PURGE
```

No SMP job removes the document root disk. The files that you put into it are in it. Copy out what you want before you scratch it.

Step 6: Check That It Is Gone

A return code says that a job ran. It does not say that the work happened. Check the result in two places.

SMP. The **`LIST`** of step 3 is the check for the inventory. You can run it again at any time. On a clean system every stanza answers “not found”.

The catalog. List what is left under the high-level qualifier.

```
    LISTCAT LEVEL(HTTPD)
```

This is the answer on the measured system, after the scratch job.

```
IDCAMS  SYSTEM SERVICES                                    TIME: 07:10:11       10/27/26     PAGE   1

  LISTCAT LEVEL(HTTPD)
NONVSAM ------- HTTPD.BREXX
     IN-CAT --- SYS1.UCAT.TK5
NONVSAM ------- HTTPD.HTML
     IN-CAT --- SYS1.UCAT.TK5
NONVSAM ------- HTTPD.HTML.TK5
     IN-CAT --- SYS1.UCAT.TK5
NONVSAM ------- HTTPD.ICON
     IN-CAT --- SYS1.UCAT.TK5
NONVSAM ------- HTTPD.LUA
     IN-CAT --- SYS1.UCAT.TK5
NONVSAM ------- HTTPD.SAMPLIB
     IN-CAT --- SYS1.UCAT.TK5
NONVSAM ------- HTTPD.STATS
     IN-CAT --- SYS1.UCAT.TK5
NONVSAM ------- HTTPD.UFSDISK0
     IN-CAT --- SYS1.UCAT.TK5
NONVSAM ------- HTTPD.UFSDISK1
     IN-CAT --- SYS1.UCAT.TK5
NONVSAM ------- HTTPD.V3R3M0.LINKLIB
     IN-CAT --- SYS1.UCAT.TK5
IDCAMS  SYSTEM SERVICES                                    TIME: 07:10:11       10/27/26     PAGE   2
         THE NUMBER OF ENTRIES PROCESSED WAS:
                   AIX ------------------0
                   ALIAS ----------------0
                   CLUSTER --------------0
                   DATA -----------------0
                   GDG ------------------0
                   INDEX ----------------0
                   NONVSAM --------------10
                   PAGESPACE ------------0
                   PATH -----------------0
                   SPACE ----------------0
                   USERCATALOG ----------0
                   TOTAL ----------------10
         THE NUMBER OF PROTECTED ENTRIES SUPPRESSED WAS 0
IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 0

IDC0002I IDCAMS PROCESSING COMPLETE. MAXIMUM CONDITION CODE WAS 0
```

Neither **`HTTPD.LINKLIB`** nor **`HTTPD.AHTTPLOD`** is in the list. The list still shows ten entries. Only **`HTTPD.SAMPLIB`** is a data set of the package, and step 4 left it in place on purpose. The other nine are not data sets that the package creates, and this chapter does not touch them. On your system, check

each remaining name before you scratch it. Expect **HTTPD.WEBROOT.UFS** and **HTTPD.SAMPLIB** in the list until you scratch them yourself.

Running the Job Twice

The job can run again. On a system where no id is left, the **UCLIN** step ends with return code 8 for the missing entries and writes no **HMA2550**, and the **LIST** gives the same answer. After the removal no zone holds the id, so the same release can be installed again.

What Changes for HTTPD 4.5.0

HTTPD 4.5.0 has the FMID **THTP450** and deletes **THTP440**. The job above is the 4.4.0 job. For 4.5.0, make these changes in both zones.

- Add one block for **THTP450** to each zone. It has four lines: **UCLIN CDS .** (**UCLIN ACDS .** in the ACDS), **DEL SYSMOD(THTP450) .**, **ENDUCL .** and **RESETRC .**.
- In the five **DEL SYSMOD(THTP440) MOD(...)** blocks, name **THTP450** in place of **THTP440**. The modules of a 4.5.0 system belong to the newest FMID. The block **DEL SYSMOD(THTP440) .** stays.
- Add **LIST CDS SYSMOD(THTP450) .** and **LIST ACDS SYSMOD(THTP450) .** to the **LIST** step.

What was measured. The 4.5.0 form of the job has 37 **DEL** statements. It was run on a TK5 system that held **THTP450**, **THTP440**, **THTP430** and **THTP420**. The **UCLIN** step ended with return code 8, from four **HMA2532** for **THTP400** and **THTP410**, two for each. 33 statements ended with **HMA2550**, and there was no **HMA2293**. The **LIST** step ended with return code 4, and all six ids were not found in the CDS and in the ACDS. After the scratch of the two libraries, the install of a package build of 4.5.0 (Chapter 2, “Installing HTTPD” on page 5) ran without an error.

The next section is for HTTPD 4.5.1. The section after it is about the case where a system holds an FMID that the job does not name, because a development build put it there.

What Changes for HTTPD 4.5.1

HTTPD 4.5.1 has the FMID **THTP451** and deletes **THTP450**. The job of “What Changes for HTTPD 4.5.0” is the 4.5.0 job. For 4.5.1, make these changes in both zones.

- Add one block for **THTP451** to each zone, with the four lines of the block for **THTP450**, and put it ahead of the block **DEL SYSMOD(THTP450) .**.
- Add five **DEL SYSMOD(THTP451) MOD(...)** blocks, ahead of the five blocks that name **THTP450**, and keep those five. The modules of a 4.5.1 system belong to **THTP451**, and those of a 4.5.0 system to **THTP450**. The job names both, so that you need not know which release the system holds.
- Add **LIST CDS SYSMOD(THTP451) .** and **LIST ACDS SYSMOD(THTP451) .** to the **LIST** step, each ahead of the **THTP450** statement of its zone.

The job has 49 **DEL** statements then, 27 against the CDS and 22 against the ACDS, and the **LIST** step has fourteen statements. The README of the installation package prints the job in full.

What was measured. The 4.5.1 form of the job was run on an MVS/CE system with SMP 4 (**HMASMP LVL 04.48**). The system held the released **THTP450** with its five modules, and the marks of **THTP440** and **THTP430**. It had been brought there by installing the 4.4.0 package and then the 4.5.0 package. It held no entry for **THTP451**, **THTP420**, **THTP410** or **THTP400**. The **UCLIN** step ended with return code

8. 31 statements ended with **HMA2550**, and 18 ended with **HMA2532**

and **HMA2560**, nine in each zone: the five **DEL SYSMOD(THTP451) MOD(...)** blocks, **DEL SYSMOD(THTP451)**, and the blocks for **THTP420**, **THTP410** and **THTP400**. There was no **HMA2293**. The **LIST** step ended with

return code 4, and all 14 statements answered "not found". **LIST CDS MOD(...)**, **LIST CDS LMOD(...)** and **LIST ACDS MOD(...)** for the five modules answered "not found" as well. On this system the **THTP450** blocks did the work and the **THTP451** blocks found nothing.

A system that holds THTP451. The 4.5.1 form of the job was run on a TK5 system that held **THTP451**, installed from the package of 4.5.1 as an upgrade from 4.5.0. The modules belong to **THTP451** and **THTP450** is a mark, so the five **DEL SYSMOD(THTP450) MOD(...)** blocks of each zone are the ones that find nothing, ten in all. They do not end with **HMA2532** like the blocks of the system above. SMP writes **HMA3572** for them, as it does for a mark that lists no module ("A Development Build with a Later FMID"). The **UCLIN** step ended with return code 8 and the **LIST** step with return code 4. The job reported **HMA2550** 35 times, **HMA2532** 4 times and **HMA3572** 10 times, and there was no **HMA2293**. All 14 **LIST** statements of the job and all 15 statements for the modules and load modules answered "not found", so the removal was complete. Scratching the libraries, allocating them again and installing the same package ended with return code 0000 in every step, with ten **HMA2380** lines and the banner of 4.5.1, and **THTP450** was a mark again.

What was not measured. A system at 4.4.0 or older was not run. Its modules belong to an FMID that no **MOD** block names, and whether the **DEL MOD** and **DEL SYSMOD** statements remove them cleanly is not known.

The next section is about the case where a system holds an FMID that the job does not name, because a development build put it there.

A Development Build with a Later FMID

A development build of the source tree is packaged under the FMID of the release that the tree is heading for. A build of 4.5.0-dev carries **THTP450**, the id of the release that the tree becomes. If you installed such a package with SMP, the system holds an id that the job of "Step 2: Cut the FMIDs out of the SMP Inventory" on page 128 does not name. The job of "What Changes for HTTPD 4.5.1" on page 136 names **THTP450** and **THTP451**. The sessions of this section were measured with the changes of "What Changes for HTTPD 4.5.0" on page 136, which the job of "What Changes for HTTPD 4.5.1" on page 136 contains, on a development build under **THTP450**. A development build under **THTP451** is covered by the same blocks, and that was not run. The rest of the section is for the job of "Step 2: Cut the FMIDs out of the SMP Inventory" on page 128 as printed, and for an id later than **THTP451**.

What the job does on such a system. The sessions of this section were measured with SMP 4 (**HMASMP LVL 04.48**) on two systems. The TK5 system held 4.3.0 and 4.4.0, installed by SMP, before the development build was installed over them. The MVS/CE system was fresh. On both, the job of the 4.4.0 chapter ends with condition code 8, and it removes the modules but not the development build. The five **DEL SYSMOD(THTP440) MOD(...)** blocks of each zone fail, because on such a system **THTP440** is only a mark with **DELBY** naming **THTP450** and lists no modules.

```
   DEL SYSMOD(THTP440) MOD(HTTPD) .
HMA3572 ** MOD HTTPD TO BE DELETED FROM ENTRY DOES NOT EXIST
HMA2560    UPDATE PROCESSING TERMINATED - UPDATE NOT COMPLETE
 ENDUCL .
HMA2050    UCLIN PROCESSING COMPLETED - HIGHEST RETURN CODE IS 08
```

The **DEL MOD** and **DEL LMOD** blocks end with **HMA2550**. The module entries are gone from both zones, and so are the ids that the job names and the system held. **THTP450** stays in both zones, and its **MOD** list still names the five modules, whose entries no longer exist. The libraries are not touched. **LIST CDS MOD(...)** now answers that the module is not found, so the modules look free, and only the function itself shows what is left. This is the TK5 system, without the page headers of the listing.

```
 LIST CDS SYSMOD(THTP450) .

SMPCDS   SYSMOD ENTRIES
```

```
  NAME

THTP450   TYPE           = FUNCTION
          STATUS         = REC  APP  ACC
          FMID           = THTP450
          JCLIN          = YES
          DATE/TIME REC  = 26.301  18:17:20
                    APP  = 26.301  18:17:50
                    ACC  = 26.301  18:18:04
          DELETE VER(001) = THTP440
          MOD            = ABEND0C1  HTTPD     HTTPDM    HTTPDMTT  HTTPDSRV

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
```

Why the leftover matters. The install job of the 4.4.0 package receives the SYSMOD, and then **APPLY** and **ACCEPT** both end with return code 12. Nothing is installed. The cause is the mark: after the development build, **THTP440** is a function that **THTP450** deleted. The server still starts, and it prints the banner of the development build. This is the control run on the TK5 system: the 4.4.0 package installed over the development build, with no removal in between.

```
 APPLY S(THTP440) CHECK .

HMA3792 ** SYSMOD THTP440 SELECTED FOR APPLY IS DELETED
HMA3703 ** APPLY PROCESSING TERMINATED BECAUSE FUNCTION SYSMOD THTP440 FAILED
HMA2050    APPLY PROCESSING COMPLETED - HIGHEST RETURN CODE IS 12

 ACCEPT S(THTP440) DIS(WRITE) .
HMA4140    SMPACDS DIRECTORY SUCCESSFULLY LOADED FOR IN-STORAGE UPDATE OPERATIONS

HMA3792 ** SYSMOD THTP440 SELECTED FOR ACCEPT IS DELETED
HMA3703 ** ACCEPT PROCESSING TERMINATED BECAUSE FUNCTION SYSMOD THTP440 FAILED
HMA3680    SMPACDS IN STORAGE DIRECTORY SUCCESSFULLY REWRITTEN
HMA2050    ACCEPT PROCESSING COMPLETED - HIGHEST RETURN CODE IS 12
```

Find out which id the system carries. Run this under the SMP procedure of your system (**SMPAPP** on TK5).

```
    //HTTPDLST JOB (SYS),'HTTPD LIST',
    //             CLASS=A,MSGCLASS=H,MSGLEVEL=(1,1),
    //             REGION=4096K
    //LISTM   EXEC SMPAPP
    //SMPCNTL  DD  *
     LIST CDS MOD(HTTPD) .
    /*
    //
```

This is the answer of the TK5 system with the development build, without the page headers.

```
 LIST CDS MOD(HTTPD) .

SMPCDS  MODULE ENTRIES

  NAME

HTTPD     LASTUPD        = THTP450  TYPE=ADD
          LIBRARIES      = DISTLIB=AHTTPLOD
          FMID           = THTP450
          RMID           = THTP450
          LMOD           = HTTPD

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 00
```

FMID = THTP450 puts you in this section. So does any other id except **THTP400**, **THTP410**, **THTP420**, **THTP430** and **THTP440**. Those five are releases, the job of “Step 2: Cut the FMIDs out of the SMP Inventory” on page 128 removes them, and this section is not yours. Only **THTP450** has been measured. A package that was built with a throwaway id of its own is the same case with another name, and that was not measured. Read your id wherever this section says **THTP450**.

Change the job in three places, in both zones. They are the changes of “What Changes for HTTPD 4.5.0” on page 136, with the placement that the measured job had.

1. Add a block for **THTP450**: **UCLIN CDS .** (**UCLIN ACDS .** in the ACDS), **DEL SYSMOD(THTP450) .**, **ENDUCL .** and **RESETRC .**. Put it ahead of the block **DEL SYSMOD(THTP440) .**. Each zone has one more block then, and the job has 37 **DEL** statements, 21 against the CDS and 16 against the ACDS.
2. In the five **DEL SYSMOD(THTP440) MOD(...)** blocks, name **THTP450** in place of **THTP440**. The modules of such a system belong to the newest id. The block **DEL SYSMOD(THTP440) .** stays.
3. Add **LIST CDS SYSMOD(THTP450) .** and **LIST ACDS SYSMOD(THTP450) .** to the **LIST** step, each ahead of the **THTP440** statement of its zone. The step has twelve statements then.

What the changed job does. It was measured on the two systems above and in four further runs. Every statement that names **THTP450** ends with **HMA2550**, in both zones: the ten **DEL SYSMOD(THTP450) MOD(...)** blocks and the two **DEL SYSMOD(THTP450)**. The **UCLIN** step still ends with condition code 8 whenever the job names an id that the system does not hold, as “Reading the return codes” on page 131 describes. How many such ids there are depends on the marks the system carries. The runs gave four (**THTP410** and **THTP400** in each zone) on TK5 after 4.3.0, 4.4.0 and the development build, six on MVS/CE with 4.4.0 before the development build, and eight on fresh systems. Because of **RESETRC .** the job runs to its end in every case. Do not wait for condition code 0. The **LIST** is the result: all twelve statements answer that the id is not found, as in “Step 3: Read the LIST” on page 131, and the eleven **LIST** statements for modules and load modules do the same. This is the first of the twelve, without the page headers.

```
 LIST CDS SYSMOD(THTP450) .

LIST     SELECT SUMMARY REPORT FOR SMPCDS

THE FOLLOWING SELECTED ENTRIES WERE NOT FOUND OR WERE NOT ELIGIBLE FOR PROCESSING

 TYPE        NAME
 SYSMOD      THTP450

HMA2050    LIST PROCESSING COMPLETED - HIGHEST RETURN CODE IS 04
```

The libraries still hold the development build. **UCLIN** does not touch them. On the TK5 system **HTTPD.LINKLIB** and **HTTPD.AHTTPLOD** each still listed the five modules **ABEND0C1**, **HTTPD**, **HTTPDM**, **HTTPDMTT** and **HTTPDSRV** after the job. If you install a release next, do not scratch the libraries and do not run the allocation job. Run the install job of the release package. If you only remove HTTPD, go on with “Step 4: Scratch the Libraries” on page 133. That step scratches those two libraries and not **HTTPD.DEV.LINKLIB**, which is yours to scratch, and it was not run after a development build.

Installing a release afterwards. Measured with the 4.4.0 package on both systems, after the changed job and without the allocation job. Every step of the install job ended with condition code 0, **APPLY** included. **APPLY** reported ten **HMA2380** lines, one for each of the five modules in each of the two libraries. These are two of them.

```
HMA2380    COPY SUCCESSFUL - MOD=ABEND0C1 - LMOD=ABEND0C1 - LIBRARY=LINKLIB - SYSMOD=THTP440 - RETURN CODE=00
HMA2380    COPY SUCCESSFUL - MOD=ABEND0C1 - LMOD=ABEND0C1 - LIBRARY=AHTTPLOD - SYSMOD=THTP440 - RETURN CODE=00
```

The **LIST** of the module then shows **FMID = THTP440** and **RMID = THTP440**, and **THTP450** is not found. **THTP430** is now a function with **DELBY = THTP440**, also on a system that never held it. That mark is the release’s own and not a leftover of the development build. After the start, the console shows the release.

```
/18.21.46 STC   43  HTTPD000I HTTPD 4.4.0 (93CD307) STARTING
/18.21.46 STC   43  HTTPD001I HTTPD 4.4.0 READY - SERVING /www
```

A banner with -DEV after a successful install means that the server was loaded from another library. This is the banner of a development build, from the start after the control run, where the install had failed.

```
/18.31.11 STC   43  HTTPD000I HTTPD 4.5.0-DEV (9116F34-DIRTY) STARTING
/18.31.11 STC   43  HTTPD001I HTTPD 4.5.0-DEV READY - SERVING /www
```

A development build is deployed to **HTTPD.DEV.LINKLIB** by default, and a procedure whose **STEPLIB** names that library ahead of **HTTPD.LINKLIB** loads the module from there. Neither the SMP deletion nor the install of the release touches **HTTPD.DEV.LINKLIB**. Read the **STEPLIB** of your procedure. If it names that library, take it out, or scratch the library when you no longer need it, and start HTTPD again. This was not measured with HTTPD: the development builds of the runs were installed from the SMP package.

Note: What has not been measured. The runs did not stop HTTPD before the jobs (“Step 1: Stop the Server” on page 128), so the server ran while SMP replaced the modules, and it was restarted after the install. A development build installed directly over a system that holds 4.3.0 or older, and a system with a history older than 4.3.0, were not run, and neither were the steps that scratch the libraries and finish the removal after a development build. The release install over a system that only the unchanged job of “Step 2: Cut the FMIDs out of the SMP Inventory” on page 128 has cleaned, with **THTP450** still in the inventory, was not run either; the control run above is the nearest case.

Chapter 12. Troubleshooting

This chapter lists what goes wrong in the order in which you meet it. First the server does not start. Then it starts, but not as the Parmlib member says. Then a client gets no answer, or an answer with the wrong status. Last, a module fails, or a response is slow. Each entry has the symptom, the cause and the cure.

The causes come from the messages and from the behaviour of the server. The *HTTPD Reference* has them in full: "Messages" lists every message, "Routes and Authentication Policy" lists what each login mode answers, and "HTTP Behaviour and Limits" lists the status codes. This chapter points to them by title.

The console screens and client sessions in this chapter were captured on a test system. The user id is IBMUSER, the host is **`localhost`**, and the addresses in the console lines are test addresses.

Where to Look First

Two places tell you what is wrong.

The console. HTTPD writes a message for what an operator can act on. Every message has an id of the form **`HTTPDnnnX`**. The last letter is I, W or E for information, warning or error. The letter does not tell you whether the server stops. **`HTTPD023E`** and **`HTTPD026E`** are errors that let the server start. The Reference lists, for every message, what the server does afterwards. Every message that goes to the console also goes into the Master Trace Table of the system. "Reading the Master Trace Table" on page 159 shows how to read it without a TSO session.

The response. A request that a client got wrong is answered with a status and a text of one line. It does not go to the console. A wrong password, for example, is not written to the console by HTTPD. So read the status line that the client got, and use **`curl -i`** to see it.

The next table sends you to the right section.

Table 38. From the symptom to the section

Symptom	Section
The job ends and **`HTTPD001I`** never appears	"A Start That Ends" on page 142
The server starts, but the port, the routes or the limits are not the ones in the member	"The Start Goes On, but Not as Intended" on page 145
The client cannot connect, or the connection closes without an answer	"The Client Cannot Connect" on page 149
The client gets 302, 400, 401, 403, 404, 405, 413, 414, 501 or 505	"The Status Is Not 200" on page 149
A user cannot log in, or a changed password has no effect	"Logins" on page 154

The client gets 503, or a module works on one system and abends on another	“A Module Does Not Run” on page 155
A response of a module takes about five seconds	“A Response Takes Five Seconds” on page 156
Requests wait, or a worker abends	“Requests Wait or Workers Abend” on page 159
One request takes long, and you want to know where the time goes	“A Request Is Slow” on page 157

A Start That Ends

A healthy start ends with **HTTPD001I ... READY** (see Chapter 8, “Operating HTTPD” on page 79). Some messages end the start before that. For most of them the server then writes **HTTPD098I HTTPD SHUTTING DOWN**, **HTTPD416I** with the statistics and **HTTPD099I HTTPD SHUTDOWN COMPLETE**, and the step ends with return code 8. The message that names the cause comes before these three. In “A Start That Ends” in “Messages”, the Reference lists all of them.

Table 39. The messages that end the start

Message	RC	Cause and cure
HTTPD037E	8	Another HTTPD holds the port (“The Port Is Busy”). Stop it, or use another port.
HTTPD030E	8	The bind to the port failed (“The Port Is Busy”).
HTTPD028E, **HTTPD031E**	8	The call to create the socket or to listen failed. The message shows the return code and the error number of the call.
HTTPD038E	8	The server opened the member of the Parmlib and could not read it to the end. Check the data set of the DD **HTTPPRM** and start again.
HTTPD411E, **HTTPD419E**, **HTTPD420E**, **HTTPD048E**, **HTTPD427E**	8	A route has no valid policy, or the value of **RES_NOPROFILE** is not valid (“A Route Without a Valid Policy” on page 143).
HTTPD033E, **HTTPD064E**	8	The socket thread could not be created, or it is not running.
HTTPD090E	8	The server could not set up the console interface. It stops at once, and no stop message follows.
HTTPD014E to **HTTPD019W**	12	A DD name that the procedure must not have is there, or a DD of the server is missing (“The Procedure Has the Wrong DD Names” on page 144). The program ends before the banner.
HTTPD012E		The server could not obtain its authorization (“Authorization Fails” on page 145). The step ends with the return code of the failed call.

The Port Is Busy

Symptom. The operator starts a second server, or restarts the first one, and the start ends.

The server takes a lock on the port before it binds. The lock is named **HTTPD.PORT.nnnnn**, where **nnnnn** is the port with five digits. When another HTTPD address space holds it, the server writes **HTTPD037E** and ends at once. It does not try again. A second server on a different port starts. This is the Parmlib member of the second server:

```
PORT=8081
BIND_TRIES=2
BIND_SLEEP=2
DOCROOT=/www
MOD=MVSMF /zosmf/*  AUTH=TOKEN
```

The first server serves the port 8081 already. This is the start of the second:

```
S HTTPD,M=PRMPORT
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.31.14
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD037E HTTPD IS ALREADY ACTIVE ON PORT 8081, THIS INSTANCE ENDS
HTTPD098I HTTPD SHUTTING DOWN
HTTPD416I STATS: 0 REQUESTS, 0 ERRORS, 0 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
IEF404I HTTPD - ENDED - TIME=05.31.14
$HASP395 HTTPD    ENDED
```

Read the capture from the top. The server logged on and registered its route (**HTTPD036I**) before it came to the port. Then **HTTPD037E** named the port, and the three stop messages followed. The cure is in the message: stop the other server, or give this one another **PORT**.

When the lock is free and the bind still fails, the server writes **HTTPD030E** with the return code and the error number of the call. If the error is EADDRINUSE, for example after a quick restart, the server waits and tries again. Each try gives **HTTPD030I EADDRINUSE, WAITING FOR TCPIP**. The keywords **BIND_TRIES** and **BIND_SLEEP** set the number of tries and the seconds between them. When no try works, the server writes **HTTPD030E** a second time and the start ends. Any other error of the bind ends the start with the second **HTTPD030E** and no try. Chapter 4, “Configuring HTTPD” on page 33 shows how to change the port. “The HTTPPRM Keywords” in the *HTTPD Reference* has the two keywords and their ranges.

TO RELEASE HTTP PORT n

A Route Without a Valid Policy

Symptom. The server writes **HTTPD420E ... HTTPD WILL NOT START** and ends.

A route that has no **AUTH=** is public. A route that loses its policy by a mistake would become public, so the server refuses to start instead. Three mistakes cause it. The first is a mode that is not one of the four. This member has a typing error in the mode:

```
PORT=8080
DOCROOT=/www
LOC=/pub/*  AUTH=BASC
```

The start ends with three messages:

```
S HTTPD,M=PRMAUTH
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.28.50
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD411E UNKNOWN AUTH MODE 'BASC' -- USE NONE, FORM, BASIC OR TOKEN
HTTPD419E LOC=/pub/* COULD NOT BE REGISTERED -- ITS AUTH POLICY IS LOST
HTTPD420E ROUTE AUTHORIZATION POLICY INCOMPLETE -- HTTPD WILL NOT START
```

```
HTTPD098I HTTPD SHUTTING DOWN
HTTPD416I STATS: 0 REQUESTS, 0 ERRORS, 0 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
IEF404I HTTPD - ENDED - TIME=05.28.50
$HASP395 HTTPD    ENDED
```

HTTPD411E names the wrong value and the four right ones. **HTTPD419E** names the route that could not be registered. **HTTPD420E** is written at the end of the member and ends the start. Write **NONE**, **FORM**, **BASIC** or **TOKEN** and start again.

The second mistake is the old keyword **LOGIN** with an operand that asked for a login. The keyword is retired. A member with **LOGIN=BASIC** would make every route without **AUTH=** public, so the server stops:

```
PORT=8080
DOCROOT=/www
LOGIN=BASIC
```

```
S HTTPD,M=PRMLOGB
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.27.44
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD048E LOGIN BASIC IS RETIRED -- ROUTES WITHOUT AUTH= WOULD BECOME PUBLIC
HTTPD420E ROUTE AUTHORIZATION POLICY INCOMPLETE -- HTTPD WILL NOT START
HTTPD098I HTTPD SHUTTING DOWN
HTTPD416I STATS: 0 REQUESTS, 0 ERRORS, 0 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
IEF404I HTTPD - ENDED - TIME=05.27.44
$HASP395 HTTPD    ENDED
```

Remove the **LOGIN** line and write **AUTH=** on each route. Chapter 5, “Routes and Server Modules” on page 47 shows the four modes. A **LOGIN** line with no operand, or with **NONE**, gives only the warning **HTTPD048W** (“The Start Goes On, but Not as Intended” on page 145).

The third mistake is a value of **RES_NOPROFILE** that is neither **ALLOW** nor **DENY**. The keyword exists to close a gate. If the server read a misspelled value as **ALLOW**, the gate would stay open without a word, so it stops. The lines of the member that matter:

```
RES_NOPROFILE=DENNY
LOC=/pub/*  AUTH=BASIC RES=FACILITY:HTTPD.TESTRES
```

The start ends with this console, the JES lines left out:

```
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD427E INVALID RES_NOPROFILE VALUE "DENNY" (ALLOW|DENY)
HTTPD417I LOCATION /pub/* REGISTERED
HTTPD420E ROUTE AUTHORIZATION POLICY INCOMPLETE -- HTTPD WILL NOT START
HTTPD098I HTTPD SHUTTING DOWN
HTTPD416I STATS: 0 REQUESTS, 0 ERRORS, 0 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
```

HTTPD427E shows the value, cut after 16 characters. The text of **HTTPD420E** is the same for every cause, so the cause is in the message before it. Write **ALLOW** or **DENY** and start again.

The Procedure Has the Wrong DD Names

Symptom. A message from **HTTPD014E** to **HTTPD019W** appears, there is no banner, and the step ends with return code 12.

The server uses the DD names **HTTPDOUT**, **HTTPDERR** and **HTTPDIN** for its standard streams. The procedure must not have the DD names **SYSPRINT**, **SYSTERM** or **SYSIN**. The routine in front of the server tests for them first, and it ends the program when it finds one (**HTTPD014E**, **HTTPD015E**, **HTTPD016E**). Then it opens the three DDs of the server. A DD that cannot be opened gives **HTTPD017W**, **HTTPD018W** or **HTTPD019W**, and the letter W does not save the start: all six conditions end the program. Compare your procedure with the sample member. The Reference has it in "The Started Task and the Package".

Authorization Fails

Symptom. The console shows **HTTPD012E name UNABLE TO DYNAMICALLY OBTAIN APF AUTHORIZATION** first. The banner and the C library line follow, each with a leading plus sign because the step is not authorized, and there is no **HTTPD002I**. The step ends.

The server needs to be authorized. At the entry it tests whether the step is authorized. If it is not, the server asks for the authorization with SVC 244, and tests again. The message says that the step is still not authorized. The profile **SVC244** in the class **FACILITY** gates the SVC. The account that counts is the one the task has when it starts, before the server logs on as **STCUSER**. Check the permissions of that account. "Authorization" in "The Started Task and the Package" in the *HTTPD Reference* describes the call.

When the start works, **HTTPD002I** tells how the step was authorized.

Table 40. The two forms of HTTPD002I

Message	Meaning
HTTPD002I AUTHORIZED BY LIBRARY (MODULE KEY 0)	The step was authorized at the entry, by the APF authorization of its library. The system fetches the modules with the protection key 0.
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)	The step was not authorized at the entry. The server authorized the task itself. The system fetched the modules with the key 8.

Both forms are a good start. They differ for the modules that the server runs ("A Module Abends with S0C4" on page 156). If you expected the first form and got the second, check the APF list for the libraries of the STEPLIB.

The Start Goes On, but Not as Intended

A warning does not stop the start. The server skips the bad line or keeps the default, and goes on. Read the console from the banner down to **HTTPD001I** after every change of the member. The next table lists the common ones.

Table 41. Messages that let the start go on

Message	Meaning and cure
HTTPD020W	The DD **HTTPPRM** cannot be opened. The server starts on its defaults: port 8080, 3 to 9 workers, the document root **/www** and no routes. Add the DD, or correct the member name ("The Member Is Not There" on page 146).
HTTPD023E	The value of **PORT** is not a number from 1 to 65535. The server keeps the earlier value or the default.
HTTPD026W	A line is not a comment and has no equals sign. The server skips it.

HTTPD029W	The keyword is not known. The server skips the line. A typing error in a keyword gives this message.
HTTPD026E, **HTTPD032E**	The idle time or the maximum age of a login is not longer than **CLIENT_TIMEOUT**. The letter is E, but it is a warning. Raise **SESSION_TIMEOUT** or **SESSION_MAXAGE**.
HTTPD048W	The retired keyword **LOGIN** has no effect. Remove it and write **AUTH=** on each route.
HTTPD412W, **HTTPD413W**, **HTTPD414W**	A route option is malformed (**HTTPD412W**), unknown (**HTTPD413W**), or given on a public route (**HTTPD414W**). The server ignores the option.
HTTPD425W	The resource of **RES=** has no profile. The route is served with no gate (“Logins” on page 154).
HTTPD426W	The same, with **RES_NOPROFILE=DENY** in the member. The route answers every request with 403 until the profile exists (“403, the Resource Is Denied” on page 152).
HTTPD440W, **HTTPD441W**, **HTTPD442W**	The module of a route could not be held at the start. The route works. The module is fetched for each request (“A Module Does Not Run” on page 155).
HTTPD003W, **HTTPD004W**	The server could not log on as **STCUSER**. It runs with the identity the task inherited (“Logins” on page 154).
HTTPD042W	The security product accepts any user ID. The server refuses all logins until it is started again (“Every Login Is Refused, HTTPD042W” on page 154).
HTTPD444W	The value of **TRACE** is not 0 and not a number from 16 to 1024. The request trace stays off (“The Request Trace” on page 40).
HTTPD067W	The server could not get the storage for the request trace. It runs without it (“The Request Trace” on page 40).
HTTPD044W	The server could not create its handle to the file system. It serves no static files (“Static Files Are Not Served” on page 158).
HTTPD047E	The server could not set up its credential key. Logins do not work.
HTTPD034W	The server could not create the worker threads. The routes that run programs are disabled.

The Member Is Not There

Symptom. The server answers on port 8080 although the member says another port, and it has none of your routes.

The start below used the member name **NOSUCH** on the **S** command. The server could not open the DD, wrote **HTTPD020W**, and started on its defaults:

```
S HTTPD,M=NOSUCH
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.48.26
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD020W CANNOT OPEN DD:HTTPPRM -- USING DEFAULTS
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

The server listened on 8080 and had no routes. Static files from the document root are all that it served. The cure is to start with the right member, or to add the missing DD to the procedure. The file

name of the capture says “refuse”, but the server did not refuse: a missing member is not an error for HTTPD.

Lines the Server Skips

A keyword that HTTPD does not know is skipped with **HTTPD029W**, and the keyword is shown in upper case:

```
S HTTPD,M=PRMKEY
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.26.28
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD029W UNKNOWN CONFIGURATION KEYWORD: COLOUR
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

A line without an equals sign gives **HTTPD026W** with the first 40 characters of the line:

```
S HTTPD,M=PRMLINE
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.26.43
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD026W UNRECOGNIZED CONFIGURATION LINE: this line has no equals sign
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

In both cases the server went on. The setting that you meant to make is not made, so a keyword with a typing error leaves the default in force. The line **PORT=0** is the same kind of case:

```
S HTTPD,M=PRMBADP
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.26.58
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD023E INVALID PORT VALUE (0)
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

The server wrote **HTTPD023E** and went on, and the capture shows that it listened on the default 8080. The message has the letter E, and the server still started. Chapter 4, “Configuring HTTPD” on page 33 describes what a bad line or value does.

The same happens with the options of a route. This member has one of each mistake, and the next capture shows what the server said:

```
PORT=8080
DOCROOT=/www
RESIDENT=MAYBE
MOD=NOSUCH /nosuch/*            AUTH=NONE
LOC=/a/*                        AUTH=NONE RES=FACILITY:HTTPD.ADMIN
LOC=/b/*                        AUTH=BASIC RES=NOCOLON
```

```
LOC=/c/*                        AUTH=NONE BOGUS=1
LOC=/d/*                        RES=FACILITY:HTTPD.NOPROFILE
```

```
S HTTPD,M=PRMRTE
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.35.14
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD443W INVALID RESIDENT VALUE "MAYBE" (YES|NO)
HTTPD036I MODULE NOSUCH REGISTERED FOR /nosuch/*
HTTPD414W AUTH=NONE IGNORES RES=FACILITY:HTTPD.ADMIN (PUBLIC ROUTE)
HTTPD417I LOCATION /a/* REGISTERED
HTTPD412W IGNORING MALFORMED RES= 'NOCOLON' (NEED CLASS:RESOURCE)
HTTPD417I LOCATION /b/* REGISTERED
HTTPD413W IGNORING UNKNOWN ROUTE OPTION 'BOGUS=1'
HTTPD417I LOCATION /c/* REGISTERED
HTTPD417I LOCATION /d/* REGISTERED
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD425W NO PROFILE FOR FACILITY:HTTPD.NOPROFILE -- /d/* NOT GATED
HTTPD440W MODULE NOSUCH NOT FOUND AT STARTUP, NOT HELD
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD188)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD088)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD008)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

The first route (**MOD=NOSUCH**) is the one of "A Module Does Not Run" on page 155. Then follow **HTTPD414W** (a resource on a public route is ignored), **HTTPD412W** (the resource has no colon), **HTTPD413W** (an option that does not exist) and **HTTPD425W** (the profile does not exist, so the route is not gated; with **RES_NOPROFILE=DENY** it is **HTTPD426W** and the route refuses every request). The server registered all four routes in the end, and it did not stop.

Sessions That Are Too Short

The server compares the idle time and the maximum age of a login with **CLIENT_TIMEOUT** at the end of the start. A login that a request is still using must not be dropped. This member sets both values below the client time-out:

```
PORT=8080
DOCROOT=/www
CLIENT_TIMEOUT=120
SESSION_TIMEOUT=1
SESSION_MAXAGE=2
```

```
S HTTPD,M=PRMSESS
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.30.24
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD026E SESSION_TIMEOUT (1 MIN) <= CLIENT_TIMEOUT (120 SEC), RAISE IT
HTTPD032E SESSION_MAXAGE (2 MIN) <= CLIENT_TIMEOUT (120 SEC), RAISE IT
HTTPD061I STARTING SOCKET THREAD    TCB(9CDB00) TASK(100FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0DD1A8)   TCB(9CD410) TASK(114FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD0A8)   TCB(9CD0E0) TASK(125FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0DD028)   TCB(9BBCF8) TASK(136FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

Both messages are warnings, and the server started. The messages show the idle time and the maximum age in minutes and the client time-out in seconds. Raise **SESSION_TIMEOUT** and **SESSION_MAXAGE**, or lower **CLIENT_TIMEOUT**.

The Client Cannot Connect

The connection is refused or does not open. The server is not running, it did not reach **HTTPD001I**, or the client uses another port. Look on the console for **HTTPD054I LISTENING ON ANY PORT n**. The command **F HTTPD,D P** shows the port that the server serves:

```
F HTTPD,D P
HTTPD100I CONS(0) "D P"
HTTPD102I HTTPD IS LISTENING ON PORT 8080
```

The server binds to all interfaces of the host. If the port is right and the connection still fails, the cause is outside HTTPD.

The connection closes and the client gets no answer. The server closes a connection without a response when a time-out ends. The limit is **CLIENT_TIMEOUT** seconds (default 10) for the first request of a connection and **KEEPALIVE_TIMEOUT** seconds (default 5) for each later one. A client that sends part of a request line and then waits is closed. If **CLIENT_TIMEOUT_MSG** is on, which is the default, and the server has received at least one character, it writes a line to the console. The line has no message id. The next screens use a time-out of 3 seconds. The client sends 13 characters of a request line and waits:

```
$ exec 3<>/dev/tcp/localhost/8080
$ printf 'GET /hello.tx' >&3
$ sleep 6
$ exec 3>&-
```

```
client 00144FC8 on socket 1 timed out after 3 seconds
Dump of 00145028 "Receive Buffer" (13 bytes)
+00000 C7C5E340 61888593 93964BA3 A7       :GET /hello.tx   :
```

The line names the client block and the socket, and the dump shows what the server had received. A client that reuses a connection after more than the keep-alive time finds it closed. That is not a fault of the server. Send the next request on a new connection.

The client waits for the data and the connection ends. When the client does not take data, the server tries again every 0.1 second. After 100 tries without progress, that is 10 seconds, it gives up on the client and writes **HTTPD065W**. This holds for a chunked response and for the body of a static file. The server does not use the connection again, and the worker is free. It writes this message at most once in 60 seconds. If the server cannot read a file from the file system in the middle of a response, it writes **HTTPD045E**, cuts the response and closes the connection. It writes this message at most once in 60 seconds, and a count at its end says how many errors it held back. A client that sees a response that stops in the middle should look for these two messages.

The Status Is Not 200

The error responses of the server have a text body of one line. The next table lists the status codes and where to read on. “Status Codes” in “HTTP Behaviour and Limits” in the *HTTPD Reference* has the full list.

Table 42. The status codes that point to a problem

Code	Cause
302	A route with form login and no session. This is the login form working (“302, the Login Form” on page 151).
400	The request is malformed: a bad request line, header or parameter.

401	The route needs a login, and the request has none or a wrong one (“401, the Route Needs a Login”).
403	The user is known, but the resource check of the route denies the access (“403, the Resource Is Denied” on page 152).
404	There is no file for the path (“404, There Is No File” on page 153).
405	The method is not one that the server knows (“405 and 501, the Method” on page 153).
413	The body of a POST is larger than 3999 bytes, and the route has no program. A module route gets such a body unread.
414	The request line is longer than 3998 characters.
500	The server cannot build the response, or a module asked for a status code that the server does not know. The server then writes **HTTPD909E** to the console.
501	POST, PUT or DELETE for a static file (“405 and 501, the Method” on page 153).
503	A module does not run (“A Module Does Not Run” on page 155).
505	The version of the request is not HTTP/1.0 and not HTTP/1.1.

401, the Route Needs a Login

A route with **AUTH=BASIC** answers a request without credentials with 401 and a challenge. The header **WWW-Authenticate** carries the realm, which is the value of **REALM**:

```
$ curl -i http://localhost:8080/private/secret.txt
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:52:44 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

A route with **AUTH=TOKEN** answers with 401 and no challenge:

```
$ curl -i http://localhost:8080/token/api.txt
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:52:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

The server never sends the header for a token route, whatever the request holds. A browser then shows no password dialog. The same is true for a request that has the header **X-CSRF-ZOSMF-HEADER**, also on a Basic route: the browser would open its own dialog, so the server leaves the header out. This is the answer of the mvsMF route, which is a token route, to a request with no login:

```
$ curl -i http://localhost:8080/zosmf/restfiles/ds?dslevel=SYS1.PARMLIB
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:53:01 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

A 401 does not say why. A user that does not exist and a user with a wrong password get the same answer as a request with no credentials:

```
$ curl -i -u NOSUCH:wrongpw http://localhost:8080/private/secret.txt
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:52:44 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

```
$ curl -i -u IBMUSER:wrongpw http://localhost:8080/private/secret.txt
HTTP/1.1 401 Unauthorized
Date: Tue, 27 Oct 2026 05:52:44 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
WWW-Authenticate: Basic realm="MVS.EXAMPLE"
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

401 Unauthorized
```

So the status line cannot tell the three apart. These are the causes to check.

- **No credential.** The server looks for the login in this order: the cookie **`Sec-Token`**, the cookie **`LtpaToken2`**, a Bearer token, and Basic credentials. A client that sends none of them gets the 401.
- **The user is not known, or the password is wrong.** An unknown user leaves a line from the security product on the console. HTTPD writes none. The next screen shows the lines that one session left on the test system: the security product refused the user **`NOSUCH`**, then a login and a logout of IBMUSER were recorded.

  ```
  HTTPD001I HTTPD 4.5.1 READY - SERVING /www
  RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:NOSUCH
  HTTPD007I USER IBMUSER  IP 192.0.2.17 LOGIN SUCCESSFUL ACEE(9BB190)
  HTTPD008I USER IBMUSER  IP 192.0.2.17 LOGOUT SUCCESSFUL
  ```

  A wrong password for a known user does not appear on the console. The password of the client is not logged.
- **The client has another address.** The login is bound to the address of the client. A user that logged in from another address does not count. A client behind a proxy or a gateway that changes the address between two requests loses its login.
- **The login has ended.** HTTPD drops a login that is idle for **`SESSION_TIMEOUT`** minutes (default 30), and one that is older than **`SESSION_MAXAGE`** minutes (default 480). A client that sends user and password with every request does not notice this. A client with a cookie or a token must log in again. **`F HTTPD,D L`** shows who is logged in:

  ```
  F HTTPD,D L
  HTTPD100I CONS(0) "D L"
  HTTPD121I USERS LOGGED IN: 1
  HTTPD122I USER IBMUSER  IP 192.0.2.17 ACEE(9BA1E0)
  ```

Chapter 6, “Authentication and Security” on page 55 describes the four modes and the sources of a login.

302, the Login Form

A route with **`AUTH=FORM`** answers a GET without a login with 302 and **`Location: /login`**. It also sets the cookie **`Sec-Uri`**, which holds the address of the request, so that the form can send the user back:

```
$ curl -i http://localhost:8080/form/page.html
HTTP/1.1 302 Moved Temporarily
Date: Tue, 27 Oct 2026 05:52:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Location: /login
Set-Cookie: Sec-Uri=YYaWmZRhl4GHhUuIo5ST; path=/
Transfer-Encoding: chunked
```

This is not a fault. The client is meant to follow the redirect and show the form. A request to a form route with a method other than GET and POST gets status 200 and the text **`Invalid REQUEST_METHOD`**, because the server cannot send such a request to a form.

403, the Resource Is Denied

The route has **`RES=class:resource`**. The user is logged in, but the security product does not give the user READ access to the resource. The answer is 403 with the body **`403 Forbidden`**:

```
$ curl -i -u IBMUSER:****** http://localhost:8080/denied/data.txt
HTTP/1.1 403 Forbidden
Date: Tue, 27 Oct 2026 05:52:48 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

403 Forbidden
```

The resource of this route is a profile that nobody may read. To cure a 403, permit the user to read the profile. Then activate the change. A profile that you define or permit is not in force until RAKF has read it. The tool that defines the profiles says so, in its own words, “Run RAKFPROF to activate the changes”. The started task **`RAKFPROF`** does it:

```
S RAKFPROF
$HASP100 RAKFPROF ON STCINRDR
RAKF0010I STC RAKFPROF STARTED USING DEFAULT STC ACCOUNT
$HASP373 RAKFPROF STARTED
IEF403I RAKFPROF - STARTED - TIME=00.40.36
RAKFPROF7  RAKF PROFILES UPDATED
$HASP395 RAKFPROF ENDED
IEF404I RAKFPROF - ENDED - TIME=00.40.36
```

The opposite fault is silent. Under RAKF a resource that has no profile is not protected, and the product answers that the access is allowed. The route then lets everybody pass. At the start the server tests every route that has **`RES=`** and a mode other than **`NONE`**, and it writes **`HTTPD425W NO PROFILE FOR class:resource -- path NOT GATED`** for each one that has no profile (see the capture in “Lines the Server Skips” on page 147). A typing error in the resource name looks the same. A class that RAKF does not define gives the same with RAKF 1.2.x (measured on TK5). RAKF 2.0.0 (MVS/CE) gives the return code 8 instead, with no **`HTTPD425W`**, and no user passes. Treat **`HTTPD425W`** as an error of your configuration. “What the Server Asks of the Security Product” in the *HTTPD Reference* describes the check.

With **`RES_NOPROFILE=DENY`** in the member, the route is not silent. The start writes **`HTTPD426W NO PROFILE FOR class:resource -- path REFUSES ALL`**, and every logged-in user gets **`REQUESTS`**
403 with the body **`403 Forbidden: no RACF profile protects this resource`**. That body is the difference from the 403 above: it says that no profile exists, not that the profile does not permit the user. Define the profile and run **`RAKFPROF`**. The server needs no restart (“Refusing a Route Whose Profile Is Missing” on page 66).

For the console route of mvsMF there is a second answer. mvsMF 1.4.0 and later check READ on the same profile themselves. HTTPD answers 403 first to a user without READ. A user who passes

the route when the profile is not defined gets status 500 from mvsMF, with the error category 4, because mvsMF refuses every user until the profile exists. That is the answer with the default **RES_NOPROFILE=ALLOW**. With **RES_NOPROFILE=DENY** HTTPD answers 403 with the body above before mvsMF is called. Define the profile.

404, There Is No File

A request for a path that no route claims is served from the document root. A file that is not there, and a file that HTTPD cannot open, get the same answer:

```
$ curl -i http://localhost:8080/nothere.html
HTTP/1.1 404 Not Found
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

404 document "/nothere.html" not found
```

The body names the decoded path. If only one file is missing, check the name and the directory in the image. If every request for a file gets a 404, the cause is more likely the file system. “Static Files Are Not Served” on page 158 has the checks.

405 and 501, the Method

A static file answers GET and HEAD. A POST, PUT or DELETE for a static file gets 501:

```
$ curl -i -X POST -d x=1 http://localhost:8080/hello.txt
HTTP/1.1 501 Not Implemented
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

501 Not Implemented
```

The answer to the POST says **Connection: close**. Use GET to read a file.

If **curl -I** gets 501, the server is HTTPD 4.4.0 or earlier. These releases do not answer a HEAD for a static file with the headers of a GET. The console message **HTTPD000I** at the start names the version (“What the Console Shows” on page 27). HTTPD 4.5.0 answers the HEAD (“What Is Not Supported” on page 74).

A method that the server does not know gets 405:

```
$ curl -i -X OPTIONS http://localhost:8080/hello.txt
HTTP/1.1 405 Method Not Allowed
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

405 Method Not Allowed
```

Chapter 7, “Serving Files” on page 69 has more on files.

Logins

"401, the Route Needs a Login" on page 150 and "403, the Resource Is Denied" on page 152 cover what a client sees. Three more faults concern the server and its security product.

The Server Has Not Its Own Identity

Symptom. The console shows **HTTPD004W RACINIT ENVIR=CREATE FAILED FOR** at the start.
user/group RC=n

At the start the server logs on as the user id of **STCUSER** and the group of **STCGROUP**, which the procedure passes in the PARM string. When the logon fails, the server writes **HTTPD004W** and keeps the identity that the task inherited. A cause is that the user id is not defined. The server goes on, and that is the danger: on a standard RAKF profile set the inherited account holds ALTER authority on every data set of the system. Define the user id and the group, then restart. **HTTPD003W** says that the server could not enter supervisor state and so skipped the logon. "The Identity of the Task" in "The Started Task and the Package" in the *HTTPD Reference* has the steps. On the healthy start the line is **HTTPD004I STC IDENTITY SET TO**, as the captures in this chapter show.
user/group VIA RACINIT

Every Login Is Refused, HTTPD042W

Symptom. The console shows **HTTPD042W SECURITY SYSTEM ACCEPTS ANY** at the start. Every login gets
USERID, ALL LOGINS REFUSED
401, also for users that are defined and use the right password. The server serves the paths that need no login. Before the message the console shows **RAKF0001** and **RAKF0002** for the logon of the task and for the user ID **HTTPPRB**, and **RAKF0003** and **RAKF0009** if the profile table is missing as well.

Cause. RAKF 1.2.x has no valid user table, for example because the load of **SYS1.SECURE.CNTL(USERS)** was rejected at the IPL (**RAKFUIDS2**, **RAKFUIDSX**). RAKF then accepts every user ID. HTTPD 4.5.1 and later find this at the start and refuse all logins.

Cure. Correct the tables before the next IPL and read the lines of the load in the log. Then restart HTTPD. A running server does not test again, and a server that has written **HTTPD042W** keeps refusing until it is started again. A good start has **RAKF0004** for the user ID **HTTPPRB** and no **HTTPD042W**. "A Security System That Accepts Any User ID" on page 62 has the details.

A Changed Password Does Not Take Effect

Symptom. An administrator changes a password in the security product, or revokes a user. A client that logged in earlier is still accepted with the old password, and the new password works as well.

HTTPD keeps a login in a table in its storage. A request with the same user, password and address finds the entry and is answered from it. The server does not ask the security product again, so the product's answer at the time of the login is what counts. An entry ends in one of these ways:

- It is idle for **SESSION_TIMEOUT** minutes (default 30).
- It is older than **SESSION_MAXAGE** minutes (default 480), counted from the login. Activity does not renew it. The next request then asks the security product again, and a revoked user or a changed password take effect at that moment.
- The client logs out.
- The server stops.

The new password is a different key for the table, so a request with it asks the security product and works. Only the old password is served from the table.

To end a login at once, restart the server, or have the user log out. If a shorter life is what the site needs, lower **SESSION_TIMEOUT** and **SESSION_MAXAGE**. Keep both above **CLIENT_TIMEOUT** in seconds, or

the server warns (“Sessions That Are Too Short” on page 148). Chapter 4, “Configuring HTTPD” on page 33 shows both keywords.

From HTTPD 4.5 a module that asks for a new session can find that the security product refuses the logon. Then the server marks the entry as stale. It does not answer by password or token from that entry any more, and it frees the entry 600 seconds later. The module answers 401. “Sessions” in “The Server Module Interface” in the *HTTPD Reference* has the result codes.

A Module Does Not Run

Symptom. The client gets 503 and a line that starts with **`External program`**, and the console shows **`HTTPD908E`**.

The route names a module, and the server could not run it, or the module ended badly. This is a route to a module that is not in any library:

```
PORT=8080
DOCROOT=/www
MOD=NOSUCH /nosuch/*            AUTH=NONE
```

```
$ curl -i http://localhost:8080/nosuch/x
HTTP/1.1 503 Service Unavailable
Date: Tue, 27 Oct 2026 05:50:25 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
Transfer-Encoding: chunked

External program NOSUCH was not found
```

The body says **`External program NOSUCH was not found`**. The server also wrote **`HTTPD908E EXTERNAL PROGRAM NOSUCH was not found`** to the console. The start had already warned about this module with **`HTTPD440W MODULE NOSUCH NOT FOUND AT STARTUP, NOT HELD`** (see the capture in “Lines the Server Skips” on page 147). The text after the module name is the reason, and it is the same for the client and for the operator.

Table 43. The reasons for a failed module

Text	Cause and cure
was not found	The module is in neither the STEPLIB nor the link list. The completion code is S806. Check the name in the route, and check that the member is in a library of the STEPLIB of the procedure.
could not be loaded	The LINK failed, and the cause is not S806. The module did not run.
was dispatched without a program name	The route has no program name.
could not be started, ESTAE create failed	The server could not set up the error exit for the LINK, so the module did not run.
returned rc n	The module ran and returned a negative code n of its own.
failed with Sxxx ABEND	The module ended with the system abend xxx. Look for the dump (“A Module Abends with S0C4” on page 156).
failed with Unnnn ABEND	The module ended with the user abend nnnn, in decimal. **`U0801`**, with the console line **`@@CRT0 - No storage for C stack`**, is a region that is too small, see below.

Only the abends leave a dump to look for. The other reasons mean that the module did not run or ended by itself. After the module ended, the server frees its storage. If that fails, it writes **HTTPD906W** with the module name and the return code, and the storage stays until the server restarts. The worker lives on in every case and serves the next request.

The abend is U0801, and it comes under load. Every request that runs a module takes one block of storage for its stack: about 256 KB for a module that sets nothing, which is the case for REXX and Lua, and about 64 KB for mvsMF. Up to **MAXTASK** requests run at once. When the region has no free block of that size, the console shows **@@CRTO - No storage for C stack** and **HTTPD908E EXTERNAL PROGRAM name failed with U0801 ABEND**, and the client gets 503. The other requests go on. Raise the **REGION** of the procedure (16M held in the one run that was measured) before you raise **MAXTASK**. If 16M is not enough, lower **MAXTASK**. "The Started Task and the Package" in the *HTTPD Reference* has the sizes and the measurement.

The module is in a library, and the route still fails. The LINK searches the STEPLIB of the started task first, and then the link list. Check that the library is in the STEPLIB of the procedure that started the server, and that the member has the name in the route. If the member comes from a development library, that library has to be in the concatenation, and it has to come before the production library so that its module wins.

A new build of the module has no effect. By default the server loads each module once at the start and holds it until it stops. The requests use that copy. Stop the task before you replace the load library, and start it again. For development you can switch the holding off with **RESIDENT=NO**. A module that was not found at the start (**HTTPD440W**) is not held. It works as soon as it is in the library.

A Module Abends with S0C4

Symptom. The client gets 503 and **External program name failed with S0C4 ABEND**. The same module worked on the test system.

A module that the system fetches from an APF-authorized library or from the link list lands in storage with the protection key 0. Its code runs with the key 8. The first write to the storage of the module abends with S0C4. The library decides, not the attribute AC(1) and not RENT. The module works when its library is not in the APF list: the system fetches it with the key 8, and the write is allowed. The same load module abends on a system where the library is authorized.

This is why **HTTPD002I** matters (see "Authorization Fails" on page 145). The form BY LIBRARY (MODULE KEY 0) says that the modules are fetched with the key 0, and every write to the static data of a module abends. The form BY SVC (MODULE KEY 8) hides the fault. The key in the message is inferred from the way the step was authorized and not measured.

The cure is in the module, not in the authorization. A module must not write to a variable at file scope or to a static variable of a function, and it must not change a string constant in place. Put the state into a local variable, onto the heap, or into the storage that **__wsaget()** returns. Read-only data such as tables is fine. The declaration decides for strings that a function translates in place: **UCHAR text[] = "..."** is a copy on the stack, and **UCHAR *text = "..."** points to the constant itself. A call of **__super()** does not help, because a module can be in key 0 without the right to switch. "Storage Rules" in "The Server Module Interface" in the *HTTPD Reference* has the rules. Chapter 9, "Writing a Server Module" on page 97 shows a module that follows them. Test a module with its library in the APF list before you call it done.

A Response Takes Five Seconds

Symptom. A module answers 200 with the right body, but the client needs about five seconds before it is done.

Look at the headers of such a response. This is a module that has the pitfall:

```
$ curl -sv -o /dev/null http://localhost:8080/hello 2>&1 | grep "^< "
< HTTP/1.1 200 OK
< Date: Tue, 27 Oct 2026 05:56:59 GMT
< Server: HTTPD Server
< X-Powered-By: MVSLOVERS
< Connection: keep-alive
< Content-Type: text/plain
<
```

There is no **`Content-Length`** and no **`Transfer-Encoding`**, and the header **`Connection`** says **`keep-alive`**. The client cannot tell where the body ends. It waits until the server closes the connection, and the server does this only when its keep-alive wait ends. The time of the whole request shows it:

```
$ curl -s -o /dev/null -w "%{http_code} %{time_total}s\n" http://localhost:8080/hello
200 4.632924s
```

The same module, written correctly, takes this:

```
$ curl -s -o /dev/null -w "%{http_code} %{time_total}s\n" http://localhost:8080/hello
200 0.069525s
```

The cause is how the server finds the end of the headers. **`http_printf`** watches for a call whose text is exactly **`"\r\n"`**. When it sees that call, it adds **`Transfer-Encoding: chunked`** for an HTTP/1.1 request that has no **`Content-Length`**, and it turns chunking on. A blank line that is joined to the last header in one call is not found. The module with the pitfall ends its headers like this:

```
http_resp(httpc, 200);
http_printf(httpc, "Content-Type: text/plain\r\n\r\n");
http_printf(httpc, "Hello, %s!\n", name);
```

The right module sends the blank line as a call of its own:

```
http_resp(httpc, 200);
http_printf(httpc, "Content-Type: text/plain\r\n");
http_printf(httpc, "\r\n");
http_printf(httpc, "Hello, %s!\n", name);
```

A module can also send a **`Content-Length`** header and then it is not chunked. The wait in the capture is close to the default of **`KEEPALIVE_TIMEOUT`**, which is 5 seconds. “Chunked Transfer” in “HTTP Behaviour and Limits” in the *HTTPD Reference* has the rules, and Chapter 9, “Writing a Server Module” on page 97 shows the complete module.

A Request Is Slow

Symptom. A client says that one request takes long. You want to know whether the time went to the client, to the server, to a module or to the send. The request trace records the time of each phase of every request, so it answers this. It has to be on before the request is made, because the server reads **`TRACE=`** only at its start. If it is off, switch it on with the steps in “The Request Trace” on page 40 and wait for the problem to come again.

The target **`TRACE`** of HTTPDSRV shows the table. The parameter **`min`** is a number of milliseconds. It leaves out the finished requests that took less. A request that is still running is always in the answer. The first line of the answer holds the totals, and the filter **`tail -n +2`** cuts it. The filter **`cut`** keeps the words of interest: the number of the request, its worker, its number on the connection, whether it still runs, the last phase it reached, its age in milliseconds, the times of the request line and of the end of the headers, the number of waits for the client, and the path.

The first capture was made while a client held a request back. The client had sent the request line and no header line, and the readout is from the third second:

```
$ curl -s -u IBMUSER:****** "http://localhost:8080/.dsrv?target=TRACE&min=1000" | tail -n +2 | cut -d" " -f1,3,7,10,11,13,15,16,27,32
seq=8 w=17FA40 n=0 live=1 last=line1 age=3090 line1=4 req=- polls=7 path=
seq=9 w=17F9C0 n=0 live=1 last=sndl age=33 line1=4 req=10 polls=0 path=/.dsrv
```

The first line is the request of the slow client. It still runs (**live=1**). Its last phase is **line1**: the server read the request line after 4 ms, and since then it has waited for the header lines. The word **req** is **-**, because the end of the headers was not reached. The server has gone back to waiting for the client seven times (**polls=7**), and each wait lasts up to half a second. The time is not the time of the server. The second line is the request that read the trace.

The second capture was made after the client had sent the rest:

```
$ curl -s -u IBMUSER:****** "http://localhost:8080/.dsrv?target=TRACE&min=1000" | tail -n +2 | cut -d" " -f1,3,7,10,11,13,15,16,27,32
seq=8 w=17FA40 n=0 live=0 last=end age=5537 line1=4 req=5517 polls=11 path=/zosmf/info
seq=11 w=17FB40 n=0 live=1 last=sndl age=24 line1=4 req=10 polls=0 path=/.dsrv
```

The request ended with **last=end** after 5,537 ms. The end of the headers came at 5,517 ms, so the server needed 20 ms for everything that follows. The cause of the long request is the client.

When the time is not in the wait for the client, find the largest gap between two phases. The next table says what each gap is. The Reference lists the phases in "The Display Modules" (target **TRACE**).

Table 44. Where the time of a request goes

Gap	Meaning
accept to **run**	The request waited for a worker. All workers were busy ("Requests Wait or Workers Abend" on page 159).
run to **req**	The server waited for the client to send the request line and the headers. **polls** counts the waits of half a second.
req to **parse**	The server parsed the request. For a POST it also read the body, so a client that sends its body slowly shows up here.
parse to **cred**	The server resolved the credential of the request.
cred to **route**	The server checked the policy of the route, with the RACF check of **RES=** if the route has one.
lnk0 to **lnk1**	The module ran. The time includes the load of the module if the server does not hold it in storage ("A Module Does Not Run" on page 155).
snd1 to **sndl**	The server sent the data. **txst** counts the pauses of a tenth of a second in which TCP/IP took no data, for a chunked response and for a static file: the client or the network is slow to receive.

The trace shows what the server did and nothing of the network. A request that is fast in the trace and slow for the client was slow in the client or in the network. On a connection that stays open, the record of the second request starts at the end of the first one, so its **line1** includes the time that the client needed to send it. A finished request that took a second or more stays in the table longer than a fast one, so a slow request is still there when you look.

Static Files Are Not Served

Symptom. Every request for a file gets 404, also for files that are in the image.

HTTPD reads files through UFSD. The message **HTTPD044W UNABLE TO INITIALIZE FILE SYSTEM** says that the server could not create its handle for the file system. It does not say that UFSD is down. A request for a file that HTTPD cannot open gets the 404, whatever the reason. Check in this order:

1. UFSD runs.
2. The image is mounted. The operator command **F UFSD,MOUNT LIST** shows the mounts.
3. The mount point is the value of **DOCROOT**. The default is **/www**.
4. The file is in the image, with the path of the request.
5. The Parmlib member does not set **UFS=0**. The value 0 switches the file system off, and then no file is served.

Chapter 7, "Serving Files" on page 69 shows the document root and the mount.

Requests Wait or Workers Abend

Requests wait. A worker serves one connection until it is closed. A connection with keep-alive holds its worker, also while the connection is idle, for at most **KEEPALIVE_TIMEOUT** seconds between two requests. When all workers up to **MAXTASK** are busy, a further client has to wait for a free worker. Raise **MAXTASK** with **F HTTPD,SET MAXTASK n**, but read about the region in "A Module Does Not Run" on page 155 first, or lower **KEEPALIVE_TIMEOUT** (Chapter 4, "Configuring HTTPD" on page 33). The command **F HTTPD,D T** shows the threads of the server. If one request is stuck in a module, the module is the cause, and more workers do not cure it. At the stop the server waits up to 10 seconds for the socket thread to end. It writes **HTTPD040I** while it waits and **HTTPD041I** when it detaches the task.

A worker abends. An abend in a module does not stop the worker ("A Module Does Not Run" on page 155). An abend in the code of the server gives **HTTPD062E ABEND code IN WORKER(..) CLIENT(..) SOCKET(n)**. The code is the abend word in eight hex digits. The server closes the client, and the worker goes on with the next request. **HTTPD063E** says that the worker could not set up its recovery, and that it dropped the client. A repeated **HTTPD062E** with the same code points to the server and not to a module. Chapter 8, "Operating HTTPD" on page 79 describes the recovery.

Reading the Master Trace Table

Every console message goes into the Master Trace Table, and HTTPD writes to the console only what an operator can act on. So the table is where you look after a failure, also when nobody watched the console. The display module HTTPDMTT shows the table over HTTP when a route for it exists (Chapter 8, "Operating HTTPD" on page 79 shows the route). This request returns the whole table as plain text, and **grep** picks out the messages of HTTPD:

```
curl -s -u IBMUSER:password "http://localhost:8080/.dmtt?d=1" | grep HTTPD
```

With the parameter **d=1** the answer is plain text. Without it, the answer is an HTML page. The module does not search or select. The table holds only the recent messages, so a server that has run for a long time may have lost the messages of its start. Read the table soon after a failure. This is the end of the table after the start of the test server. The capture shows the last lines:

```
$ curl -s -u IBMUSER:****** http://localhost:8080/.dmtt | sed -n "/<pre>/,/<\/pre>/p" | sed -e "s/<[^>]*>//g" | tail -n 4
FFFF  5.33.53 STC   nn  HTTPD061I STARTING WORKER(0FE648)   TCB(9BB328) TASK(1CBFC8) STACKSIZE(65536)
FFFF  5.33.53 STC   nn  HTTPD061I STARTING WORKER(0FE5C8)   TCB(9BAEB8) TASK(1DCFC8) STACKSIZE(65536)
FFFF  5.33.54 STC   nn  HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

The table is the whole console log of the system. Protect the route with a login mode, and with a resource check if you can (Chapter 6, "Authentication and Security" on page 55).

What to Collect

When a problem stays, collect these items before you ask for help:

- the messages of the start, from the banner to **HTTPD001I**, or the lines of the failure, from the console or from the Master Trace Table
- the Parmlib member that the server read
- the output of **F HTTPD,D V** and **F HTTPD,D C**, which show the version and the configuration that the server runs
- the status line and the body of the response, from **curl -i**
- for a module that abended, the name and the abend code from **HTTPD908E** and the dump, if you have it

Remove passwords and the names of your own systems from what you send.

Appendix A. Glossary

This glossary defines the terms of HTTPD, and of HTTP, MVS and the security product as far as this book uses them. A term printed in *italics* in a definition has an entry of its own.

ACEE The accessor environment element. It holds the identity of a user for the security product. HTTPD creates one for each login, and one for the *started task* (Chapter 6, “Authentication and Security” on page 55).

APF The authorized program facility. A program in an APF-authorized library can run with authority. HTTPD needs it, and it obtains it from the library or with *SVC 244* (Chapter 2, “Installing HTTPD” on page 5, Chapter 12, “Troubleshooting” on page 141).

AUTH= The route option that sets how a route asks for a login. The values are **`NONE`**, **`FORM`**, **`BASIC`** and **`TOKEN`**. A route without it is public (Chapter 5, “Routes and Server Modules” on page 47, Chapter 6, “Authentication and Security” on page 55).

Basic The login mode **`AUTH=BASIC`**. The server answers a request without credentials with 401 and a **`WWW-Authenticate`** header that names the *realm*. A request with the header **`X-CSRF-ZOSMF-HEADER`** gets the 401 without that header (Chapter 6, “Authentication and Security” on page 55).

chunked transfer A way to send a body of unknown length. The response has the header **`Transfer-Encoding: chunked`** and no **`Content-Length`**. The server adds the header itself when a module ends the headers with a blank line of its own (Chapter 9, “Writing a Server Module” on page 97, Chapter 12, “Troubleshooting” on page 141).

display module A *server module* of HTTPD that shows the state of the server over HTTP, for example HTTPDMTT for the *master trace table* (Chapter 8, “Operating HTTPD” on page 79).

DOCROOT The Parmlib keyword that names the *UFS* directory with the static files. The default is **`/www`** (Chapter 7, “Serving Files” on page 69).

FMID The id of a function SYSMOD in *SMP*. HTTPD has one for each minor release. The id starts with **`THTP`** and ends with three digits, for example **`THTP440`**. A new release deletes the id of the earlier one, and an id is used once (Chapter 10, “Upgrading HTTPD” on page 111, Chapter 11, “Removing HTTPD” on page 127).

FORM The login mode **`AUTH=FORM`**. A request without a login gets a 302 to the login form (Chapter 6, “Authentication and Security” on page 55).

HTTPC The control block of one client, 4096 bytes. The server creates it when it accepts a connection and frees it when the connection closes. A *server module* works with a pointer to it (Chapter 9, “Writing a Server Module” on page 97).

HTTPD The HTTP server for MVS 3.8j, and the name of its control block. The block holds the state of the whole server (Chapter 1, “Introducing HTTPD” on page 1, Chapter 9, “Writing a Server Module” on page 97).

HTTPX The function vector. A *server module* calls the functions of the server through it. New entries are added at the end, so a module built against an earlier release keeps working (Chapter 9, “Writing a Server Module” on page 97).

keep-alive A connection that stays open after a response, so that the client can send the next request on it. It holds one *worker* until it closes. **KEEPALIVE_TIMEOUT** limits the wait between two requests (Chapter 4, “Configuring HTTPD” on page 33, Chapter 12, “Troubleshooting” on page 141).

link list The list of libraries that the system searches for a load module after the STEPLIB. HTTPD finds its *server modules* in the STEPLIB and then in the link list (Chapter 5, “Routes and Server Modules” on page 47).

listener The socket on which the server waits for connections. The *socket thread* accepts the connections on it. **PORT** sets its port (Chapter 4, “Configuring HTTPD” on page 33).

load module A program in a load library. A *server module* is a load module that HTTPD runs with the LINK service (Chapter 9, “Writing a Server Module” on page 97).

LOC The Parmlib keyword for a route that has no program. It makes a path prefix a place for *static files*, with its own **AUTH=** (Chapter 5, “Routes and Server Modules” on page 47, Chapter 7, “Serving Files” on page 69).

master trace table The table in which the system keeps the most recent console messages. The *display module* HTTPDMTT shows it over HTTP (Chapter 8, “Operating HTTPD” on page 79, Chapter 12, “Troubleshooting” on page 141).

MOD The Parmlib keyword for a route that runs a *server module*. The first route that matches the path wins (Chapter 5, “Routes and Server Modules” on page 47).

Parmlib The data set with the configuration of HTTPD. The DD name is **HTTPPRM**. The member is chosen with **M=** on the start command, and the default is **HTTPPRM0** (Chapter 4, “Configuring HTTPD” on page 33).

RACF The security product of IBM. HTTPD asks the security product, through *SAF*, to check a user id and a password and to check access to a resource.

RAKF A security product for MVS 3.8j that answers the *SAF* calls of HTTPD. A new profile is active after the started task **RAKFPROF** has run (Chapter 6, “Authentication and Security” on page 55, Chapter 12, “Troubleshooting” on page 141).

realm The name that the server gives in the Basic challenge and in the login form. The Parmlib keyword **REALM** sets it (Chapter 6, “Authentication and Security” on page 55).

RES= The route option that names a resource in a class of the security product. A logged-in user needs READ access to it. Without access the answer is 403 (Chapter 6, “Authentication and Security” on page 55).

resident module A *server module* that the server loads once at the start and holds until it ends. After a new build, stop the task and start it again (Chapter 5, “Routes and Server Modules” on page 47, Chapter 12, “Troubleshooting” on page 141).

SAF The system authorization facility. The interface through which a program calls the security product (Chapter 6, “Authentication and Security” on page 55).

server module A load module that serves the requests of a route and runs in the task of HTTPD. mvsMF is one (Chapter 9, “Writing a Server Module” on page 97).

session The state of a login in the server: the user, the *ACEE*, the address of the client and a *token*. It ends by idle time (**SESSION_TIMEOUT**), by age (**SESSION_MAXAGE**),

by logout, or when the server ends (Chapter 6, “Authentication and Security” on page 55, Chapter 4, “Configuring HTTPD” on page 33).

SMF The system management facilities. HTTPD can write SMF records (Chapter 8, “Operating HTTPD” on page 79, Chapter 4, “Configuring HTTPD” on page 33).

SMP The program that installs and removes products on MVS in units called SYSMODs. HTTPD is installed as a function SYSMOD with an *FMID* (Chapter 2, “Installing HTTPD” on page 5, Chapter 11, “Removing HTTPD” on page 127).

socket thread The thread that waits on the *listener* and gives each new connection to a *worker*. It has a stack of 32 KB (Chapter 8, “Operating HTTPD” on page 79).

started task A program that the operator starts with the **`S`** command from a cataloged procedure. HTTPD runs as a started task, and the procedure is **`HTTPD`** (Chapter 8, “Operating HTTPD” on page 79).

static file A file that the server sends as it is, from the *UFS* under *DOCROOT*. (Chapter 7, “Serving Files” on page 69).

STCUSER The PARM keyword of the procedure that names the user id under which the started task runs after its start. The matching keyword for the group is **`STCGROUP`**. The default is the user **`HTTPD`** in the group **`USER`**. Only if the logon fails does the task keep the identity that it inherited (Chapter 6, “Authentication and Security” on page 55, Chapter 12, “Troubleshooting” on page 141).

STEPLIB The DD of the procedure with the load libraries. The system searches it first and then the *link list* (Chapter 5, “Routes and Server Modules” on page 47).

SVC 244 The service call through which a program obtains *APF* authorization at run time. The profile **`SVC244`** in the class **`FACILITY`** gates it (Chapter 12, “Troubleshooting” on page 141).

token A value that stands for a login. The server sets it in the cookie **`Sec-Token`**. A route with **`AUTH=TOKEN`** answers a request without credentials with a bare 401 (Chapter 6, “Authentication and Security” on page 55).

UFS The Unix-like file system with directories and long names that UFSD provides. HTTPD reads its *static files* from it (Chapter 7, “Serving Files” on page 69).

UFSD The product that serves a *UFS* to other tasks. It mounts the *webroot image* at the path that is the value of *DOCROOT* (Chapter 7, “Serving Files” on page 69).

webroot image The file system image with the static files of HTTPD. The package delivers it as **`httpd-webroot.img`**. It is uploaded to a data set and mounted by UFSD (Chapter 2, “Installing HTTPD” on page 5, Chapter 7, “Serving Files” on page 69).

worker A thread that serves the requests of one connection from start to end. The pool has **`MINTASK`** to **`MAXTASK`** workers, and each has a stack of 64 KB (Chapter 4, “Configuring HTTPD” on page 33, Chapter 12, “Troubleshooting” on page 141).

Index

/

/.dm 91
/.dmtt 93, 159
/.dsrv 92
/login 50, 57
/logout 50, 57
/zosmf 50

3

3.x 123
302 151

4

4.0.x 122
401 55, 150
403 55, 65, 152
  no profile 66
404 30, 153
405 153
405 Method Not Allowed 74

5

501 153
501 Not Implemented 74
503 52, 155

_

__wsaget 103

A

abend
  module 107
  worker 94
ACEE 61
allocation 10
APF 8
  troubleshooting 145
APF authorization 51
argv
  module 99
AUTH= 55
  wrong value 143
AUTH=BASIC 55
AUTH=FORM 55
AUTH=NONE 55
AUTH=TOKEN 55
authentication 55
  two stages 55
authorization 8

B

Basic authentication 56
bearer token 58
BIND_TRIES 142

C

C HTTPD 83
cancel 83
CGI= 44
CGI_CONTEXT_POINTERS 52
cgistart 99
chapters
  overview 3
client 28
CLIENT_STATS 38
CLIENT_TIMEOUT 37
CLIENT_TIMEOUT_DUMP 38
CLIENT_TIMEOUT_MSG 38
code page 40
  static files 73
CODEPAGE 40, 73
comment 34
configuration 33
  errors 43
connection refused 149
console
  login messages 67
console commands 84
Content-Type 72
context block 52
CP037 40
credential cache 60, 61, 154
curl 28
  testing a module 106

D

D CONFIG 35
D PORTS 35
D TIME 40
D, procedure symbol 33
DD:HTTPPRM 33
DEBUG 40
default.html 71
DEL SYSMOD 128
deploy
  server module 105
development build 137
directory request 71
DISPLAY CONFIG 85
DISPLAY LOGIN 87
DISPLAY MEMORY 88
display modules 90
  authentication 90
DISPLAY STATS 88
DISPLAY THREADS 87
DOCROOT 39, 69

changing 71
document root 23, 39, 69

E

ESTAE 94, 107
exempt paths 67
extension route 47

F

FACILITY class 65
file extension 72
first match 48
first request 27
FMID 111
form login 57
FTP 9

H

HEAD 74, 153
header terminator 156
HELLO
example module 97
HELP command 85
history 3
http_cgictx_get 52
http_new_session 108
http_printf 100
http_resp 100
HTTPD
level 108
limits 1
overview 1
purpose 1
HTTPD.AHTTPLOD
removal 133
HTTPD.LINKLIB
removal 133
HTTPD.WEBROOT.UFS 69
HTTPD001I 27
HTTPD002I 145
HTTPD003W 154
HTTPD004I 61
HTTPD004W 61, 154
HTTPD007I 67
HTTPD008I 67
HTTPD008W 67
HTTPD009W 67
HTTPD012E 145
HTTPD014E 144
HTTPD020W 82, 146
HTTPD025I 40
HTTPD026E 38, 148
HTTPD026W 43, 147
HTTPD029W 43, 147
HTTPD030E 142
HTTPD030I 142
HTTPD032E 38, 148
HTTPD037E 81, 142
HTTPD042W 62, 67, 154
HTTPD044W 78, 158
HTTPD048E 143
HTTPD054I 35
HTTPD062E 94, 159
HTTPD063E 159
HTTPD065W 149
HTTPD066I 40
HTTPD067W 40
HTTPD070E 40
HTTPD098I 142
HTTPD099I 142
HTTPD101E 89
HTTPD128I 35
HTTPD132I 42
HTTPD133I 35
HTTPD138I 66
HTTPD411E 143
HTTPD412W 53
HTTPD413W 53
HTTPD414W 53
HTTPD419E 143
HTTPD420E 44, 143
HTTPD424W 60
HTTPD425W 53, 65
HTTPD426W 66
HTTPD427E 66, 143
HTTPD430W 42
HTTPD440W 51
HTTPD441W 51
HTTPD442W 51
HTTPD444W 40
HTTPD908E 52, 94, 107, 155
HTTPDBG 40
HTTPDM 90
HTTPDMTT 90, 159
HTTPDSRV 90
HTTPPRB 62, 154
HTTPPRM
DD name 33
member 33
member missing 146
HTTPX
append-only 108

I

IBM1047 40
IDCAMS 133
IEA703I 52
index.html 28, 71
install job 12, 115
installation 5
introduction 1

K

keep-alive 30, 37
wait 156
KEEPALIVE_MAX 37
count 37
KEEPALIVE_TIMEOUT 37
KEYWORD=VALUE 34

L

level 112
limits
security 68

lineage 3
link list 51
LIST 131
LISTCAT 135
LOC 47, 77
  route 47
LOC= 48
LOGIN 44
login
  troubleshooting 154
LtpaToken2 58

M

M, procedure symbol 33
main
  server module 99
master trace table 93, 159
MAXTASK 36
mbt
  server module 103
mbt.toml 103
messages
  reading 141
methods
  static files 74
migration 123
MIME types 72
MINTASK 36
MOD 47
  route 47
MOD= 47
MODIFY command 84
module
  missing 52
  resident 51
  static data 156
  storage 103
  troubleshooting 155
  writing 97
monitoring 95
MOUNT 70
mvsMF 7, 50
MVSMF 50

N

not found 30

O

operation 79
operator 79

P

P HTTPD 82, 128
package
  contents 5
PARMLIB 18
Parmlib member 1, 33
  format 34
  minimal 34
  production 34
password change 61, 154
path
  decoding 74
path traversal 74
pattern
  route 48
percent encoding 74
persistent connection 30
PORT 35
port
  change 35
port in use 142
POST_STRING 99
predecessor 122
prerequisites 2, 7
problem report 160
problems 125
  installation 26
procedure 18
program name 47
public paths 67
public route 56

Q

QUERY_STRING 99
quiesce 107

R

RACF 55
RAKF 8, 55, 65
  users and profiles 20
RAKF0001 62, 154
RAKF0002 62
RAKF0003 62
RAKF0004 62, 154
RAKF0009 62
REALM 60
realm 60, 150
RECLAIM 44
refuse to start 44
removal 127
  procedure 134
  RAKF 134
  webroot 134
request
  flow 2
request trace 40
  slow request 157
request variables 99
requirements 2
RES= 65
  403 152
  profile missing 66
RES_NOPROFILE 66
  wrong value 143
RESIDENT 51, 105
response
  chunked 101
  module 100
response headers 29
restart 25, 84
retired keywords 44
return code 13, 115
  start refused 142

route 1
route errors 53
routes 47
  shipped 25

S

S HTTPD,M= 33
S0C4 156
S222 83
S806 155
sample members 124
scratch 133
SCRIPT_FILENAME 47
Sec-Token 57
Sec-Uri 57, 151
security 55
security system
  accepts any user ID 62
server module 1, 47, 97
  LINK 97
server side includes 74
session 60
  stale 154
SESSION_MAXAGE 38, 60, 154
SESSION_TIMEOUT 38, 60
SET MA 36
SET MAXTASK 88
SET MI 36
SET MINTASK 88
SET STATS 42, 88
shutdown
  module 107
SMF 42
  level 42
  operator 94
SMP
  LIST 131
  removal 128
SMPAPP 128
socket thread 2
SSI 74
stale credential 61
start 24, 79
  console messages 27
  healthy 79
  refused 81, 142
started task 1
  start 79
started task identity 61
static data
  module 103
static files 69
  protect 48
static locations 77
status codes
  troubleshooting 149
status line 29
STCGROUP 61
STCUSER 61, 154
STEPLIB 51
  server module 105
stop 10, 82, 114
  before removal 128
storage key 156
subpool 5 103
SVC 244 145
symptoms 141
SYSPRINT
  not allowed 144

T

text and binary 73
THTP450 136
  development build 137
THTP451 111, 136
time zone 40
time-out
  client 37, 149
  diagnostic 38
TLS 68
token 58
TOKEN
  when to use 60
trace 40
TRACE 40, 157
Transfer-Encoding 101, 156
troubleshooting 141
  first request 31
  static files 78
TZOFFSET 40, 44

U

UCLIN 128
UFS 69
UFS=0 78
UFSD 7, 69
  mount 70
  not running 78
  troubleshooting 158
ufsd-utils 69
uninstall 127
upgrade 111
upload
  XMIT 9

V

verification 17
verify 120
  removal 135
verify configuration 35

W

warnings
  start 145
web root 1
web root image 69
webroot 23
welcome page 28
wildcard 48
worker thread 2
worker threads 36
workers
  busy 159
WWW-Authenticate 56, 150

Z

z/OSMF 50