HITACHI
Inspire the Next

Hitachi Content Platform

Using the Default Namespace

Hitachi Data Systems

MK-95ARC012-19

Contents

Preface ix

Intended audience ix
Product version x
Syntax notation x
Related documents xi
Getting help xiv
Comments xiv

1 Introduction to Hitachi Content Platform 1

About Hitachi Content Platform 2
Object-based storage 2
Namespaces and tenants 3
Object representation 4
Data access 5
Namespace access protocols 5
HCP metadata query API 6
HCP Search Console 7
HCP Data Migrator 8
HCP nodes 9
Replication 9
Default namespace operations 10
Operation restrictions 10
Supported operations 11
Prohibited operations 12

2 HCP file system 15

Root directories 16
Object naming considerations 16
Sample data structure for examples 18

Metadirectories 18
Metadirectories for directories 18
Metadirectories for objects 20
Metafiles 21
Metafiles for directories 21
Metafiles for objects 26
Complete metadata structure 30

3 Object properties 33

Object metadata 34
Ownership and permissions 36
Viewing permissions 37
Octal permission values 38
Ownership and permissions for new objects 38
Changing ownership and permissions for existing objects 39
Retention 39
Retention periods 40
Retention classes 42
Retention settings in retention.txt 43
Changing retention settings 45
Specifying a date and time 48
Specifying an offset 49
atime synchronization with retention 51
Triggering atime synchronization for existing objects 52
Removing the association 54
How atime synchronization works 54
atime synchronization example 56
Shred setting 57
Index setting 58
Custom metadata 60
Replication collisions 62
Object content collisions 62
System metadata collisions 63
Custom metadata collisions 66

4 HTTP 69

URLs for HTTP access to a namespace 70
URL formats 70
URL considerations 72
Access with a cryptographic hash value 73
Transmitting data in compressed format 75
Browsing the namespace with HTTP 76
Working with objects 77

Storing an object and, optionally, custom metadata . . . 77
Checking the existence of an object . . . 86
Retrieving an object and, optionally, custom metadata . . . 89
Deleting an object . . . 102
Working with directories . . . 105
Creating an empty directory . . . 105
Checking the existence of a directory . . . 106
Listing directory contents . . . 108
Deleting a directory . . . 112
Working with system metadata . . . 113
Specifying metadata on object creation . . . 114
Specifying metadata on directory creation . . . 119
Retrieving HCP-specific metadata . . . 121
Retrieving POSIX metadata . . . 123
Modifying HCP-specific metadata . . . 123
Modifying POSIX metadata . . . 125
Working with custom metadata . . . 131
Storing custom metadata . . . 131
Checking the existence of custom metadata . . . 134
Retrieving custom metadata . . . 136
Deleting custom metadata . . . 139
Checking the available storage and software version . . . 141
HTTP usage considerations . . . 143
HTTP permission checking . . . 143
HTTP persistent connections . . . 144
Storing zero-sized files with HTTP . . . 144
Using HTTP with objects open for write . . . 144
Failed HTTP write operations . . . 145
HTTP connection failure handling . . . 145
Data chunking with HTTP write operations . . . 146
Multithreading with HTTP . . . 146

5 WebDAV . . . 147

WebDAV methods . . . 148
URLs for WebDAV access to the default namespace . . . 149
URL formats . . . 149
URL considerations . . . 151
Browsing the namespace with WebDAV . . . 152
WebDAV properties . . . 153
Live and dead properties . . . 154
Storage properties . . . 154
HCP-specific metadata properties for WebDAV . . . 155
Metadata properties for objects . . . 155

Metadata properties for directories . . . 157
PROPPATCH example . . . 158
PROPFIND example . . . 159
Using the custom-metadata.xml file to store dead properties . . . 160
WebDAV usage considerations . . . 161
Basic authentication with WebDAV . . . 161
WebDAV permission checking . . . 161
WebDAV persistent connections . . . 161
WebDAV client timeouts with long-running requests . . . 162
WebDAV object locking . . . 162
Storing zero-sized files with WebDAV . . . 162
Using WebDAV with objects open for write . . . 163
Failed WebDAV write operations . . . 163
Multithreading with WebDAV . . . 163
WebDAV return codes . . . 164

6 CIFS . . . 167

Namespace access with CIFS . . . 168
CIFS examples . . . 168
CIFS example 1: Storing an object . . . 169
CIFS example 2: Changing a retention setting . . . 169
CIFS example 3: Retrieving an object . . . 169
CIFS example 4: Retrieving deletable objects . . . 170
CIFS usage considerations . . . 170
CIFS case sensitivity . . . 170
CIFS permission translations . . . 171
Changing directory permissions when using Active Directory . . . 172
Creating an empty directory with atime synchronization in effect . . . 172
CIFS lazy close . . . 172
Storing zero-sized files with CIFS . . . 172
Out-of-order writes with CIFS . . . 173
Using CIFS with Objects open for write . . . 173
Failed CIFS write operations . . . 174
Temporary files created by Windows clients . . . 174
Multithreading with CIFS . . . 175
CIFS return codes . . . 176

7 NFS . . . 177

Namespace access with NFS . . . 178
NFS examples . . . 179
NFS example 1: Adding a file . . . 179
NFS example 2: Changing a retention setting . . . 179

NFS example 3: Using atime to set retention . . . 180
NFS example 4: Creating a symbolic link in the namespace . . . 180
NFS example 5: Retrieving an object . . . 180
NFS example 6: Retrieving deletable objects . . . 181
NFS usage considerations . . . 181
NFS lazy close . . . 181
Storing zero-sized files with NFS . . . 182
Out-of-order writes with NFS . . . 182
Using NFS with objects open for write . . . 182
Failed NFS write operations . . . 183
NFS reads of large objects . . . 184
Walking large directory trees . . . 184
NFS delete operations . . . 184
NFS mounts on a failed node . . . 184
Multithreading with NFS . . . 185
NFS return codes . . . 186

8 SMTP . . . 187

Storing individual emails . . . 188
Naming conventions for email objects . . . 188

9 General usage considerations . . . 191

Choosing an access protocol . . . 192
Using a hosts file . . . 193
DNS name and IP address considerations . . . 194
Directory structures . . . 195
Non-WORM objects . . . 196
Moving or renaming objects . . . 196
Deleting objects under repair . . . 197
Deleting directories . . . 197
Multithreading . . . 197

A HTTP reference . . . 199

HTTP methods . . . 200
HTTP return codes . . . 204
HCP-specific HTTP response headers . . . 209

B Java classes for examples . . . 213

GZIPCompressedInputStream class . . . 214
WholeIOInputStream class . . . 220
WholeIOOutputStream class . . . 221

Glossary 225

Index 239

Preface

This book is your guide to working with the default namespace for an **Hitachi Content Platform** (**HCP**) system. It introduces HCP concepts and describes how HCP represents namespace content using familiar data structures. It includes instructions for accessing the namespace using the supported namespace access protocols and explains how to store, view, retrieve, and delete objects in the namespace, as well as how to change object metadata such as retention and permissions. It also contains usage considerations to help you work more effectively with the namespace.

Note: Throughout this book, the word *Unix* is used to represent all UNIX®-like operating systems (such as UNIX itself or Linux®).

Intended audience

This book is intended for people who need to know how to store, retrieve, and otherwise manipulate data and metadata in an HCP default namespace. It provides information for users who are writing applications to access the namespace and for users who are accessing the namespace directly through a command-line interface or GUI (such as Windows® Explorer).

If you are writing applications, this book assumes you have programming experience. If you are accessing the default namespace directly, this book assumes you have experience with the tools you use for file manipulation.

Tip: If you are new to HCP, be sure to read the first three chapters of this book before writing HCP applications or accessing the default namespace directly.

Product version

This book applies to release 7.3 of HCP.

Syntax notation

The table below describes the conventions used for the syntax of commands, expressions, URLs, and object names in this book.

Notation	Meaning	Example
boldface	Type exactly as it appears in the syntax (if the context is case insensitive, you can vary the case of the letters you type)	This book shows: **mount** You enter: mount
italics	Replace with a value of the indicated type	This book shows: *hash-algorithm* You enter: SHA-256
|	Vertical bar — Choose one of the elements on either side of the bar, but not both	This book shows: **fcfs_data**|**fcfs_metadata** You enter: fcfs_data or: fcfs_metadata
[ ]	Square brackets — Include none, one, or more of the elements between the brackets	This book shows: **fcfs_data**[*/directory-path*] You enter: fcfs_data or: fcfs_data/images
( )	Parentheses — Include exactly one of the elements between the parentheses	This book shows: (**+**|**-**)*hhmm* You enter: +0500 or: -0500
-object-spec	Replace with the combination of the directory path and name of an object	This book shows: X-DocURI-0: /fcfs_data/*object-spec-1* You see: X-DocURI-0: /fcfs_data/images/wind.jpg
-path	Replace with a directory path with no file or object name	This book shows: **fcfs_data/***directory-path* You enter: fcfs_data/corporate/employees

Related documents

The following documents contain additional information about Hitachi Content Platform:

- *Administering HCP* — This book explains how to use an HCP system to monitor and manage a digital object repository. It discusses the capabilities of the system, as well as its hardware and software components. The book presents both the concepts and instructions you need to configure the system, including creating the tenants that administer access to the repository. It also covers the processes that maintain the integrity and security of the repository contents.

- *Managing a Tenant and Its Namespaces* — This book contains complete information for managing the HCP tenants and namespaces created in an HCP system. It provides instructions for creating namespaces, setting up user accounts, configuring the protocols that allow access to namespaces, managing search and indexing, and downloading installation files for HCP Data Migrator. It also explains how to work with retention classes and the privileged delete functionality.

- *Managing the Default Tenant and Namespace* — This book contains complete information for managing the default tenant and namespace in an HCP system. It provides instructions for changing tenant and namespace settings, configuring the protocols that allow access to the namespace, managing search and indexing, and downloading installation files for HCP Data Migrator. It also explains how to work with retention classes and the privileged delete functionality.

- *Replicating Tenants and Namespaces* — This book covers all aspects of tenant and namespace replication. Replication is the process of keeping selected tenants and namespaces in two or more HCP systems in sync with each other to ensure data availability and enable disaster recovery. The book describes how replication works, contains instructions for working with replication links, and explains how to manage and monitor the replication process.

- *HCP Management API Reference* — This book contains the information you need to use the HCP management API. This RESTful HTTP API enables you to create and manage tenants and namespaces programmatically. The book explains how to use the API to access an HCP system, specify resources, and update and retrieve resource properties.

- *Using a Namespace* — This book describes the properties of objects in HCP namespaces. It provides instructions for accessing namespaces by using the HTTP, WebDAV, CIFS, and NFS protocols for the purpose of storing, retrieving, and deleting objects, as well as changing object metadata such as retention and shred settings. It also explains how to manage namespace content and view namespace information in the Namespace Browser.

- *Using the HCP HS3 API* — This book contains the information you need to use the HCP HS3 API. This S3™-compatible, RESTful, HTTP-based API enables you to work with buckets and objects in HCP. The book introduces the HCP concepts you need to understand in order to use HS3 effectively and contains instructions and examples for each of the bucket and object operations you can perform with HS3.

- *Using the HCP OpenStack Swift API* — This book contains the information you need to use the HCP HSwift API. This OpenStack Swift, RESTful, HTTP-based API enables you to work with containers and objects in HCP. The book introduces the HCP concepts you need to understand in order to use HSwift effectively and contains instructions and examples for each of the container and object operations you can perform with HSwift.

- *HCP Metadata Query API Reference* — This book describes the HCP metadata query API. This RESTful HTTP API enables you to query namespaces for objects that satisfy criteria you specify. The book explains how to construct and perform queries and describes query results. It also contains several examples, which you can use as models for your own queries.

- *Searching Namespaces* — This book describes the HCP Search Console (also called the Metadata Query Engine Console). It explains how to use the Console to search namespaces for objects that satisfy criteria you specify. It also explains how to manage and manipulate queries and search results. The book contains many examples, which you can use as models for your own searches.

- *Using HCP Data Migrator* — This book contains the information you need to install and use HCP Data Migrator (HCP-DM), a utility that works with HCP. This utility enables you to copy data between local file systems, namespaces in HCP, and earlier HCAP archives. It also supports bulk delete operations and bulk operations to change object metadata. Additionally, it supports associating custom metadata and

ACLs with individual objects. The book describes both the interactive window-based interface and the set of command-line tools included in HCP-DM.

- *Installing an HCP System* — This book provides the information you need to install the software for a new HCP system. It explains what you need to know to successfully configure the system and contains step-by-step instructions for the installation procedure.

- *Deploying an HCP-VM System* — This book contains all the information you need to install and configure an HCP-VM system. The book also includes requirements and guidelines for configuring the VMWare® environment in which the system is installed.

- *Third-Party Licenses and Copyrights* — This book contains copyright and license information for third-party software distributed with or embedded in HCP.

- *HCP-DM Third-Party Licenses and Copyrights* — This book contains copyright and license information for third-party software distributed with or embedded in HCP Data Migrator.

- *Installing an HCP SAIN System — Final On-site Setup* — This book contains instructions for deploying an assembled and configured single-rack HCP SAIN system at a customer site. It explains how to make the necessary physical connections and reconfigure the system for the customer computing environment. It also contains instructions for configuring Hi-Track® Monitor to monitor the nodes in an HCP system.

- *Installing an HCP RAIN System — Final On-site Setup* — This book contains instructions for deploying an assembled and configured HCP RAIN system at a customer site. It explains how to make the necessary physical connections and reconfigure the system for the customer computing environment. The book also provides instructions for assembling the components of an HCP RAIN system that was ordered without a rack and for configuring Hi-Track Monitor to monitor the nodes in an HCP system.

Getting help

The Hitachi Data Systems® customer support staff is available 24 hours a day, seven days a week. If you need technical support, call:

- United States: (800) 446-0744
- Outside the United States: (858) 547-4526

Note: If you purchased HCP from a third party, please contact your authorized service provider.

Comments

Please send us your comments on this document:

HCPDocumentationFeedback@hds.com

Include the document title, number, and revision, and refer to specific sections and paragraphs whenever possible. All comments become the property of Hitachi Data Systems.

Thank you!

1

Introduction to Hitachi Content Platform

Hitachi Content Platform (**HCP**) is a distributed storage system designed to support large, growing repositories of fixed-content data. HCP stores objects that include both data and metadata that describes that data and presents the objects as files in a standard directory structure.

HCP provides access to objects through a variety of industry-standard protocols, as well as through various HCP-specific interfaces.

This chapter introduces basic HCP concepts and includes information on what you can do with an HCP repository.

About Hitachi Content Platform

Hitachi Content Platform is a combination of software and hardware that provides an object-based data storage environment. An HCP repository stores all types of data, from simple text files to medical images to multigigabyte database images.

HCP provides easy access to the repository for adding, retrieving, and deleting data. HCP uses write-once, read-many (WORM) storage technology and a variety of policies and internal processes to ensure the integrity of the stored data and the efficient use of storage capacity.

Object-based storage

HCP stores **objects** in the repository. Each object permanently associates data HCP receives (for example, a document, an image, or a movie) with information about that data, called **metadata**.

An object encapsulates:

- **Fixed-content data** — An exact digital reproduction of data as it existed before it was stored in HCP. Once it's in the repository, this fixed-content data cannot be modified.

- **System metadata** — System-managed properties that describe the fixed-content data (for example, its size and creation date). System metadata consists of POSIX metadata as well as HCP-specific settings.

- **Custom metadata** — Optional metadata that a user or application provides to further describe an object. Custom metadata is typically specified in XML format.

  You can use custom metadata to create self-describing objects. Users and applications can use this metadata to understand and repurpose the object content.

HCP also stores directories and symbolic links. These items have POSIX metadata and, in the case of directories, HCP-specific metadata, but no fixed-content data or custom metadata.

HCP supports appendable objects. An **appendable object** is one to which data can be added after it has been successfully stored. Appending data to an object does not modify the original fixed-content data, nor does it create a new version of the object. Once the new data is added to the object, that data also cannot be modified.

Note for WebDAV users: An object is equivalent to a WebDAV resource. A directory is equivalent to a WebDAV collection.

For more information on metadata, see Chapter 3, "Object properties," on page 33.

Namespaces and tenants

An HCP repository is partitioned into namespaces. A **namespace** is a logical grouping of objects such that the objects in one namespace are not visible in any other namespace.

Namespaces provide a mechanism for separating the data stored for different applications, business units, or customers. For example, you could have one namespace for accounts receivable and another for accounts payable.

Namespaces also enable operations to work against selected subsets of repository objects. For example, you could perform a query that targets the accounts receivable and accounts payable namespaces but not the employees namespace.

Default and HCP namespaces

An HCP system can have multiple namespaces, including one special namespace called the **default namespace**. Applications are typically written against namespaces other than the default; these namespaces are called **HCP namespaces**. The default namespace is most often used with applications that existed before release 3.0 of HCP.

The table below outlines the major differences between the default and HCP namespaces.

Feature	HCP Namespaces	Default Namespace
Storage usage quotas	✓	
Object ownership (not related to POSIX UID)	✓	
Access control lists (ACLs) for objects	✓	

(Continued)

Feature	HCP Namespaces	Default Namespace
Object versioning	✓	
Multiple custom metadata annotations	✓	
Namespace ownership by HCP users	✓	
RESTful HTTP/HTTPS API for data access	✓	
Non-RESTful HTTP/HTTPS protocol for data access		✓
Data access authentication with HTTP/HTTPS	✓	
RESTful HS3 API for data access (compatible with Amazon® S3)	✓	
NDMP protocol for backup and restore		✓

Note: This book describes how to use only the default namespace. For information on using HCP namespaces, see *Using a Namespace*.

Tenants

Namespaces are owned and managed by administrative entities called **tenants**. A tenant typically corresponds to an organization such as a company or a division or department within a company.

One tenant, called the **default tenant**, owns the default namespace and only that namespace. Other tenants can each own one or more HCP namespaces.

Object representation

HCP includes a standard POSIX file system called HCP-FS that represents each object in the default namespace as a set of files. One of these files has the same name as the object. This file contains the fixed-content data. When downloaded or opened, this file has the same content as the originally stored item.

The other files that HCP-FS presents contain object metadata. These files, which are either plain text or XML, are called **metafiles**.

HCP-FS presents both data files and metafiles in standard directory structures. Directories that contain metafiles are called **metadirectories**.

This view of stored objects as conventional files and directories enables HCP to support routine file-level calls. Users and applications can thus find fixed-content data and metadata in familiar ways.

For more information on how HCP-FS represents objects, see Chapter 2, "HCP file system," on page 15.

Data access

HCP supports access to namespace content through:

- Several industry-standard protocols
- The HCP metadata query API
- The HCP Search Console
- HCP Data Migrator

Namespace access protocols

HCP supports client access to a namespace through these industry-standard protocols: HTTP, WebDAV, CIFS, and NFS. With these protocols, you can access namespaces programmatically with applications, interactively with a command-line tool, or through a GUI. You can use these protocols to perform actions such as storing objects in the namespace, viewing and retrieving objects, changing object metadata, and deleting objects.

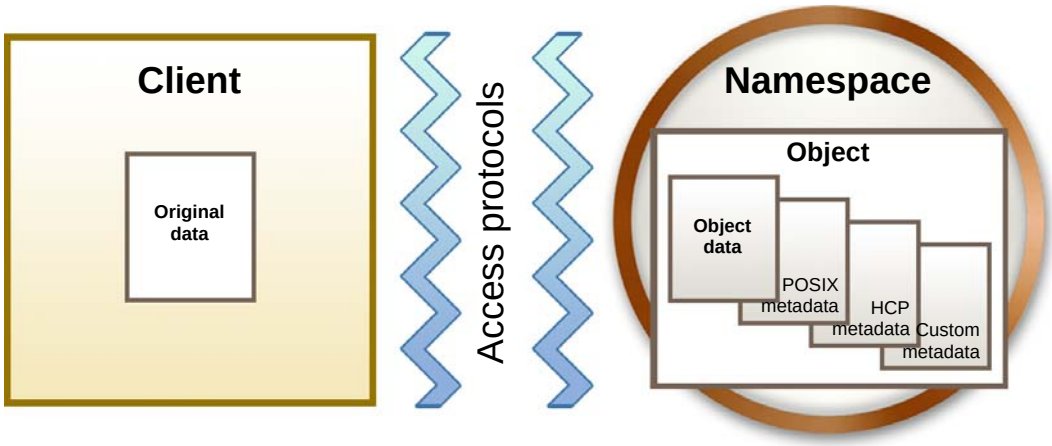
HCP allows special-purpose access to the namespace through two additional protocols: SMTP (for storing email) and NDMP (for backing up and restoring the namespace).

Objects added to the namespace through any protocol are immediately accessible through any other protocol.

Namespace access protocols are enabled individually in the HCP system configuration. If you cannot access the namespace through any given protocol, you can ask your namespace administrator to enable it.

Note: This book does not address the NDMP protocol. For information on that protocol, see your namespace administrator.

The figure below shows the relationship between original data, namespace objects, and the supported namespace access protocols.



For information on how to decide which namespace access protocol is most suitable for your purposes, see "Choosing an access protocol" on page 192.

HCP metadata query API

The **HCP metadata query API** lets you search HCP for objects that meet specified criteria. The API supports two types of queries:

- **Object-based queries** search for objects based on object metadata. This includes both system metadata and the content of custom metadata. The query criteria can also include the object location (that is, the namespace and/or directory that contains the object). These queries use a robust query language that lets you combine search criteria in multiple ways.

  Object-based queries search only for objects that currently exist in the repository.

- **Operation-based queries** search not only for objects currently in the repository but also for information about objects that have been deleted by a user or application or deleted through disposition. Criteria for operation-based queries can include object status (for example, created or deleted), change time, index setting, and location.

The metadata query API returns object metadata only, not object data. The metadata is returned either in XML format, with each object represented by a separate element, or in JSON format, with each object represented by a separate name/value pair. For queries that return large numbers of objects, you can use paged requests.

For information on using the metadata query API, see *HCP Metadata Query API Reference*.

HCP Search Console

The **HCP Search Console** is an easy-to-use web application that lets you search for and manage objects based on specified criteria. For example, you can search for objects that were stored before a certain date or that are larger than a specified size. You can then delete the objects listed in the search results or prevent those objects from being deleted. Similar to the metadata query API, the Search Console returns only object metadata, not object data.

By offering a structured environment for performing searches, the Search Console facilitates e-discovery, namespace analysis, and other activities that require the user to examine the contents of namespaces. From the Search Console, you can:

- Open objects
- Perform bulk operations on objects
- Export search results in standard file formats for use as input to other applications
- Publish feeds to make search results available to web users

The Search Console works with either of these two search facilities:

- The **HCP metadata query engine** — This facility is integrated with HCP and works internally to perform searches and return results to the Search Console. The metadata query engine is also used by the metadata query API.

Note: When working with the metadata query engine, the Search Console is called the **Metadata Query Engine Console**.

- The **Hitachi Data Discovery Suite (DDS) search facility** — This facility interacts with HDDS, which performs searches and returns results to the HCP Search Console. HDDS is a separate product from HCP.

The Search Console can use only one search facility at any given time. The search facility is selected at the HCP system level. If no facility is selected, the HCP system does not support use of the Search Console to search namespaces.

Each search facility maintains its own index of objects in each search-enabled namespace and uses this index for fast retrieval of search results. The search facilities automatically update their indexes to account for new and deleted objects and changes to object metadata.

For information on using the Search Console, see *Searching Namespaces*.

Note: Not all namespaces support search. To find out whether the default namespace is search enabled, see your namespace administrator.

HCP Data Migrator

HCP Data Migrator (**HCP-DM**) is a high-performance, multithreaded, client-side utility for viewing, copying, and deleting data. With HCP-DM, you can:

- Copy objects, files, and directories between the local file system, HCP namespaces, default namespaces, and earlier HCAP archives
- Delete individual objects, files, and directories and perform bulk delete operations
- View the content of objects and files
- Rename files and directories on the local file system
- View object, file, and directory properties
- Change system metadata for multiple objects in a single operation
- Add, replace, or delete custom metadata for objects
- Create empty directories

HCP-DM has both a graphical user interface (GUI) and a command-line interface (CLI).

For information on using HCP-DM, see *Using HCP Data Migrator*.

HCP nodes

The core hardware for an HCP system consists of servers that are networked together. These servers are called **nodes**.

When you access an HCP system, your point of access is an individual node. To identify the system, however, you can use either the DNS name of the system or the IP address of an individual node. When you use the DNS name, HCP selects the access node for you. This helps ensure an even distribution of the processing load.

For information on when to use an IP address instead of the DNS name, see "DNS name and IP address considerations" on page 194.

Replication

Replication is the process of keeping selected HCP tenants and namespaces and selected default-namespace directories in two or more HCP systems in sync with each other. Basically, this entails copying object creations, deletions, and metadata changes between systems. HCP also replicates retention classes. For the default namespace, the HCP system administrator selects the directories to be replicated.

A **replication topology** is a configuration of HCP systems that are related to each other through replication. Typically, the systems in a replication topology are in separate geographic locations and are connected by a high-speed wide area network.

You can read from namespaces on all systems to which those namespaces are replicated. The replication configuration set at the system level determines on which systems you can write to namespaces.

Replication has several purposes, including:

- If one system in a replication topology becomes unavailable (for example, due to network issues), another system in the topology can provide continued data availability.
- If one system in a replication topology suffers irreparable damage, another system in the topology can serve as a source for disaster recovery.
- If multiple HCP systems are widely separated geographically, each system may be able to provide faster data access for some applications than the other systems can, depending on where the applications are running.

- If an object cannot be read from one system in a replication topology (for example, because a node is unavailable), HCP can try to read it from another system in the topology. Whether HCP tries to do this depends on the namespace configuration.

- If a system that participates in a replication topology is unavailable, HTTP requests to that system can be automatically serviced by another system in the topology. Whether HCP tries to do this depends on the namespace configuration.

Note: Not all HCP systems support replication.

Default namespace operations

You use namespace access protocols to perform operations in the default namespace. For more information on using each protocol to perform namespace operations, see the chapters for the individual protocols.

Some operations relate to specific types of metadata. For more information on this metadata, see Chapter 3, “Object properties,” on page 33.

Operation restrictions

The operations that you can perform on a namespace are subject to the following restrictions:

- The namespace access protocol must support the operation.

- You must be allowed access to the target object.

- The namespace access protocol must be configured to allow access to the namespace from your client IP address.

- The namespace configuration must allow the operation. The namespace can be configured to prevent reading, writing, or deleting objects and metadata.

Supported operations

The table below lists the operations HCP supports for the default namespace and indicates which protocols you can use to perform those operations.

Operation	HTTP	WebDAV	CIFS	NFS	SMTP
Write data from files or memory to the namespace to create an object	✓	✓	✓	✓	
Transmit data to and from HCP in gzip-compressed format	✓				
Send email directly to the namespace					✓
View the content of an object	✓	✓	✓	✓	
Copy an object		✓	✓	✓	
Append to existing objects			✓	✓	
Delete an object that's not under retention	✓	✓	✓	✓	
View a metafile	✓	✓	✓	✓	
Override default index, retention, and shred settings when storing an object	✓				
Change the retention setting for an object or directory	✓	✓	✓	✓	
Hold or release an object	✓	✓	✓	✓	
Change the index setting for an object or directory	✓	✓	✓	✓	
Enable shredding for an object and change the shred setting for a directory	✓	✓	✓	✓	
Override default POSIX UID and GID when storing an object	✓				
Override default POSIX permissions when storing an object	✓				
Change the POSIX UID and GID for an object	✓	✓	✓	✓	
Change the POSIX permission settings for an object	✓	✓	✓	✓	
Change the POSIX **atime** or **mtime** value for an object	✓		✓	✓	
Store, replace, or delete custom metadata for an object	✓	✓			
Store or retrieve object data and custom metadata in a single operation	✓				
Read custom metadata	✓	✓	✓	✓	
Create an empty directory in the namespace	✓	✓	✓	✓	
View the namespace directory structure, including both directories and metadirectories	✓	✓	✓	✓	

(Continued)

Operation	HTTP	WebDAV	CIFS	NFS	SMTP
Rename an empty directory (unless **atime** synchronization is enabled)		✓	✓	✓	
Delete an empty directory	✓	✓	✓	✓	
Create a symbolic link			✓	✓	
Read through a symbolic link to an object	✓	✓	✓	✓	
Delete a symbolic link	✓	✓	✓	✓	

Tip: You can use the HCP Search Console to delete, hold, or release multiple objects with a single operation.

These considerations apply to symbolic links:

- If you use CIFS to create a symbolic link, you can read through the link only with CIFS. You cannot use CIFS to read through a symbolic link created with NFS.
- HTTP and WebDAV support for reading through symbolic links is limited to retrieving object data. Other HTTP and WebDAV operations on symbolic links may produce unexpected results.
- HCP doesn't automatically delete a symbolic link when the target object is deleted. Instead, the link remains and points to a nonexistent object. To remove the link, you need to explicitly delete it.

Prohibited operations

In the default namespace, HCP never lets you:

- Rename an object.
- Rename a nonempty directory.
- Overwrite a successfully stored object.
- Modify the fixed-content portion of an object.

- Delete an object that's under retention.

Note: If the namespace is configured to allow it, authorized users of the administrative interface for the namespace can delete objects that are under retention.

- Delete a nonempty directory.
- Shorten the retention period of an object.
- Store a file (other than a file containing custom metadata) or create a directory or symbolic link anywhere in the metadata structure.
- Delete a metafile (other than a metafile containing custom metadata) or metadirectory.
- Create a hard link.

2

HCP file system

The HCP file system (HCP-FS) represents objects in the default namespace using the familiar file and directory structure. For each object, HCP-FS presents a data file and a standard set of metafiles. You can view the object content in the data file. You can view the object metadata in the metafiles. Similarly, HCP-FS presents a standard set of metafiles for each directory. You can view the directory metadata in these metafiles.

HCP-FS maintains separate branches of this structure for files that contain object data and for those that contain object metadata. The structure of the metadata branch parallels that of the data branch.

This chapter describes the files and directories HCP-FS uses to represent namespace objects. It also includes considerations for naming objects.

For information on the metadata HCP maintains for objects, see Chapter 3, "Object properties," on page 33.

Root directories

HCP-FS presents the data files and metafiles for objects in the default namespace under two root directories:

- **fcfs_data** heads the directory structure containing the data files for all objects. You create the structure under `fcfs_data` when you store data and create directories in the namespace. Each data file and directory in this structure has the same name as the object or directory it represents. All object and directory names are user-supplied, with the exception of names for email objects and directories.

- **fcfs_metadata** heads the directory structure containing all the metafiles and metadirectories for objects and directories. This structure parallels the file and directory structure under `fcfs_data`, excluding symbolic links. HCP-FS creates this structure automatically as you store data and create directories in the namespace.

The *fcfs* in `fcfs_data` and `fcfs_metadata` stands for fixed-content file system. You cannot change the name of either of these directories.

Object naming considerations

When naming objects, directories, and symbolic links, keep these considerations in mind:

- The name of each item under `fcfs_data` must conform to POSIX naming conventions. In particular:

  - Object names are case sensitive.

    For concerns about case sensitivity with Windows clients, see "CIFS case sensitivity" on page 170.

  - Names can include nonprinting characters, such as spaces and line breaks.

  - All characters are valid except the NULL character (ASCII 0 (zero)) and the forward slash (ASCII 47 (/)), which is the separator character in directory paths.

  - The client operating system, in conjunction with HCP, ensures that object specifications are converted, as needed, to conform to POSIX requirements (for example, when using CIFS, backslashes (\) are converted to forward slashes (/)).

- .directory-metadata is a reserved name.

- The maximum length for the combined directory path and name of an object, symbolic link, or metafile, starting below `fcfs_data` or `fcfs_metadata` and including separators, is 4,095 bytes.

- For CIFS and NFS, the maximum length of an individual item name is 255 bytes. This applies not only to naming new objects but also to retrieving existing objects. Therefore, an object stored through HTTP or WebDAV with a longer name may not be accessible through CIFS or NFS.

- Some character-set encoding schemes, such as UTF-8, can require more than one byte to encode a single character. As a result, such encoding can invisibly increase the length of an individual object name, causing it to exceed protocol-specific limitations. Or, the encoding can cause a full object specification (directory path and object name) to exceed the HCP limit of 4,095 bytes.

Note: In some cases, an extremely long object name may prevent a CIFS or NFS client from reading the entire directory that contains the object. When this happens, attempts to list the contents of the directory result in an error.

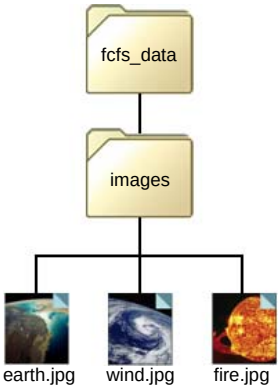
- When searching the namespace, HDDS and HCP rely on UTF-8 encoding conventions to find objects by name. If the name of an object is not UTF-8 encoded, searches for the object by name may return unexpected results.

- When the metadata query engine or HCP search facility indexes an object with a name that includes certain characters that cannot be UTF-8 encoded, it percent-encodes those characters. Searches for such objects by name must explicitly include the percent-encoded characters in the name.

Names for email objects stored through the SMTP protocol are system-generated. For information how HCP names email objects, see "Naming conventions for email objects" on page 188.

Sample data structure for examples

When you view the contents of the default namespace, HCP-FS shows each directory and data file under `fcfs_data`. The figure below shows a sample directory structure headed by `fcfs_data`. It contains one subdirectory, named `images`. `images` contains three data files named `earth.jpg`, `wind.jpg`, and `fire.jpg`. These files represent the content of the objects named `earth.jpg`, `wind.jpg`, and `fire.jpg`, respectively.



All the data files and directories presented by HCP-FS represent objects and directories that users have added to the default namespace, with one exception. The `fcfs_data` directory has a system-generated subdirectory named `.lost+found`, which is where HCP puts broken objects in the unlikely event it finds any. If you see objects in this directory, tell your namespace administrator.

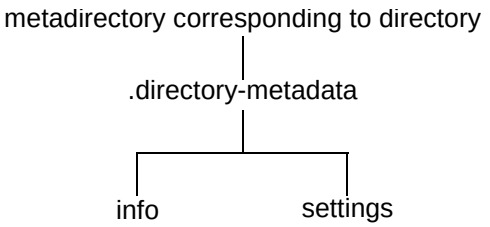
Metadirectories

Objects and directories in the default namespace each have their own set of metafiles organized in a metadirectory structure that parallels the data directory structure. The entire metadirectory structure is under the `fcfs_metadata` metadirectory.

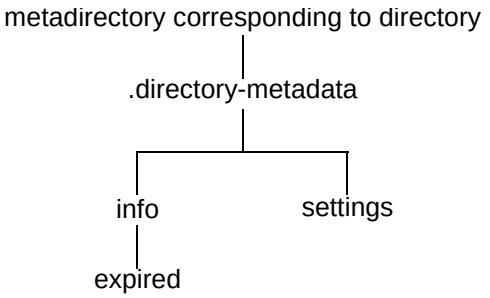
Metadirectories for directories

Each directory in the default namespace has a corresponding metadirectory with the same name (with the exception of `fcfs_data` for which the corresponding metadirectory is `fcfs_metadata`). For example, the `images` directory has a corresponding metadirectory named `images`.

Each of these corresponding metadirectories has a subdirectory named `.directory-metadata`. Each `.directory-metadata` directory has two subdirectories, `info` and `settings`, that contain the metafiles that describe the corresponding directory under `fcfs_data`.



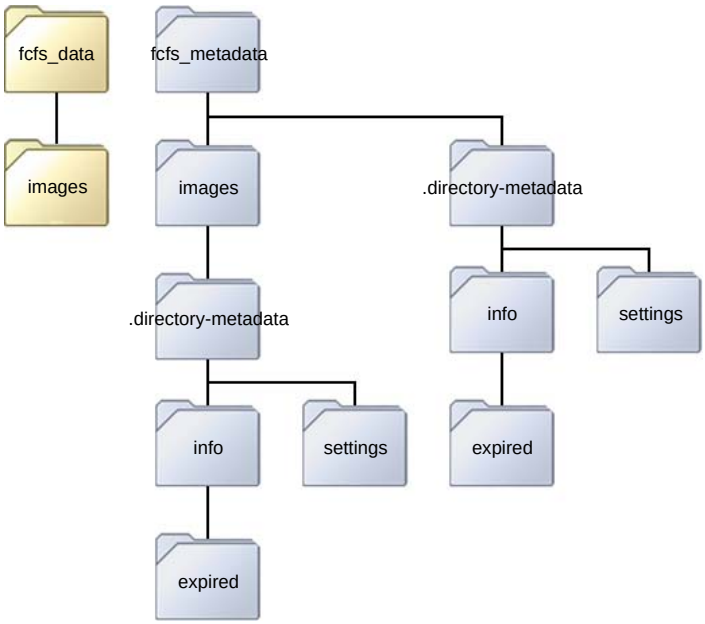
Each `info` directory has a subdirectory named `expired` that contains representations of the currently deletable objects in the corresponding directory under `fcfs_data`.



The object representations in the `expired` directory are metafiles only and have no corresponding data. To see the data for these objects, you need to look in the directory structure under `fcfs_data`. Only the owner of an object can delete that object through the `expired` directory.

The directory tree under `fcfs_metadata` mirrors the tree under `fcfs_data`. So, if the `images` directory has a subdirectory named `planets`, the `images` metadirectory also has a subdirectory named `planets`.

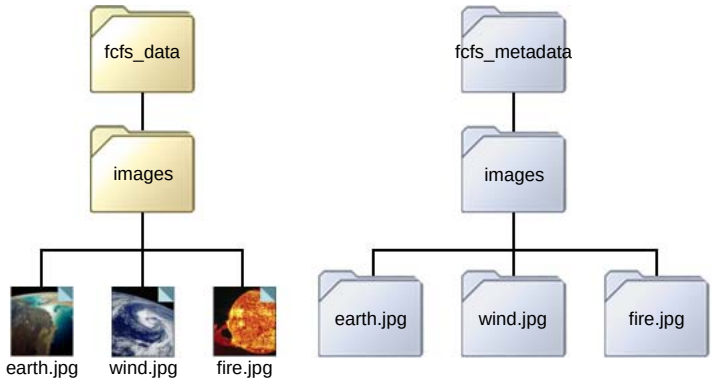
The figure below shows the metadirectory structure that corresponds to the `fcfs_data` and `images` directory structure.



Metadirectories for objects

Each data file for an object has a corresponding metadirectory with the same name. The location of this metadirectory mirrors the location of the data file. For example, the `wind.jpg` data file is in the `images` directory, and the `wind.jpg` metadirectory is in the `images` metadirectory.

The figure below shows the metadirectory structure that corresponds to the data files in the `images` directory.



Metafiles

HCP-FS presents individual metafiles for each piece of HCP-specific metadata for both objects and directories. It doesn't present individual metafiles for POSIX metadata. However, it does present one additional metafile for both objects and directories. This metafile summarizes both the HCP-specific and POSIX metadata for an object.

HCP-FS also presents one special metafile for directories. This metafile, which lists the retention classes defined for the namespace, is the same for each directory. For information on retention classes, see "Retention classes" on page 42.

Metafiles contain either plain text or XML, so you can read them easily. You can view and retrieve metafiles through the HTTP, WebDAV, CIFS, and NFS protocols. You can also use these protocols to overwrite metafiles that contain the HCP-specific metadata you can change. By overwriting a metafile, you change the metadata for the corresponding object.

HCP-FS also shows custom metadata as a metafile. This metafile is present only when you've stored custom metadata for an object. You can store or replace custom metadata only with the HTTP and WebDAV protocols.

HCP-FS doesn't present any metafiles for symbolic links.

Note: You cannot explicitly change the POSIX metadata for metafiles. If you specify an **mtime** value for an object or directory in an HTTP request or WebDAV command, the **mtime** values of the corresponding metadirectories equal the specified value. However, the **mtime** values for the metafiles in these directories reflect the time the request executed. (The **atime** values of the metafiles equal any specified **atime** value.)

Metafiles for directories

HCP-FS presents these metafiles for directories:

- In the `info` directory:

```
created.txt
core-metadata.xml
retention-classes.xml
```

- In the `settings` directory:

  ```
  dpl.txt
  index.txt
  retention.txt
  shred.txt
  ```

For backward compatibility, HCP-FS also presents a metafile named `tpof.txt`. This metafile is superseded by `dpl.txt`.

The table below describes the content of these metafiles. For more information on this metadata, see Chapter 3, "Object properties," on page 33.

Metafile	Description
Metafiles in the info directory	
created.txt	Contains the date the directory was created. This metafile contains two lines: • The first line is the date expressed as the number of seconds since January 1, 1970. • The second line is the date in this ISO 8601 format: yyyy-MM-ddThh:mm:ssZ *Z* represents the offset from UTC and is specified as: (+|-)hhmm For example: 1324287918 2011-12-19T09:45:18-0500 You can view the content of this metafile, but you cannot change it.

(Continued)

Metafile	Description
core-metadata.xml	Contains a summary of the HCP-specific and POSIX metadata for the directory. For example: <core-metadata xsi:schemaLocation="http://www.hds.com core-metadata-7_0.xsd"> <version>3</version> <name>/images</name> <name-bytes>2F696D61676573</name-bytes> <object-type>Directory</object-type> <creation-time>1232376318</creation-time> <update-time>1232376318</update-time> <change-time>1232376318</change-time> <access-time>1232376318</access-time> <uid>0</uid> <gid>0</gid> <mode>40755</mode> <shred>false</shred> <index>true</index> <retention-value>0</retention-value> <retention-string>Deletion Allowed</retention-string> <retention-hold>false</retention-hold> <size>238985</size> <tpof>1</tpof> <dpl>2</dpl> <hash-scheme>SHA-256</hash-scheme> <hash-value>0B86212A66A792A79D58BB18...</hash-value> <replicated>true</replicated> <replicationCollision>false</replicationCollision> <ingestProtocol>CIFS_NFS</ingestProtocol> </core-metadata> The **version** element identifies the version of the `core-metadata.xml` file. You can view the content of this metafile, but you cannot change it. To see the XML schema for this metafile, use this URL: http://default.default.*hcp-domain-name*/static/core-metadata-7_0.xsd

(Continued)

Metafile	Description

retention-classes.xml	Contains a list of the retention classes defined for the namespace. For each retention class, the metafile shows:
	
	• The retention class name
	
	• The type of value defined for the retention class — either Offset or Special Value
	
	• The value of the retention class
	
	• Whether objects in the class are automatically deleted when they expire
	
	• The retention class description
	
	Here's an example of a `retention-classes.xml` metafile that lists two retention classes:
	
	`<retention-classes>`
	`<retention-class>`
	`<name>HlthReg-107</name>`
	`<method>Offset</method>`
	`<value>A+21y</value>`
	`<allow-disposition>true</allow-disposition>`
	`<description>Health reg M-DC006-107</description>`
	`</retention-class>`
	`<retention-class>`
	`<name>SEC-Perm</name>`
	`<method>Special Value</method>`
	`<value>Deletion Prohibited</value>`
	`<allow-disposition>false</allow-disposition>`
	`<description>Permanent record</description>`
	`</retention-class>`
	`</retention-classes>`
	
	You can view the content of this metafile, but you cannot change it.
	
	For more information on retention classes, see "Retention classes" on page 42.

(Continued)

Metafile	Description
Metafiles in the settings directory	
dpl.txt	Contains the data protection level (DPL) for each new object stored in the directory. The DPL is the number of copies of the object HCP must maintain in the namespace to ensure the integrity and availability of the object. For example: 2 You can view the content of this metafile, but you cannot change it.
index.txt	Contains the default index setting for objects and directories added to the directory. You can view and change the content of this metafile. Changing this setting does not affect the index setting for existing objects and directories in the directory. For details on the content of the `index.txt` metafile and how to modify it, see "Index setting" on page 58.
retention.txt	Contains the default retention rule, such as a retention period or class, for objects added to the directory. This rule is also the default rule for new directories created in the directory. You can view and change the content of this metafile. Changing this setting does not affect the retention setting for existing objects and directories in the directory. For details on the content of the `retention.txt` metafile and how to modify it, see "Retention" on page 39.
shred.txt	Contains the default shred setting for objects and directories added to the directory. You can view and change the content of this metafile. Changing this setting does not affect the shred setting for existing objects and directories in the directory. For details on the content of the `shred.txt` metafile and how to modify it, see "Shred setting" on page 57.

Metafiles for objects

HCP-FS presents these metafiles for objects:

```
created.txt
dpl.txt
hash.txt
index.txt
replication.txt
retention.txt
shred.txt
core-metadata.xml
custom-metadata.xml
```

For backward compatibility, HCP-FS also presents a metafile named `tpof.txt`. This metafile is superseded by `dpl.txt`.

The table below describes the metadata in these metafiles. For more information on this metadata, see Chapter 3, "Object properties," on page 33.

Metafile	Description
created.txt	Contains the date and time at which the object was stored in the namespace. This metafile contains two lines: • The first line is the date expressed as the number of seconds since January 1, 1970. • The second line is the date in this ISO 8601 format: *yyyy-MM-dd*T*hh:mm:ssZ* *Z* represents the offset from UTC and is specified as: (+|-)*hhmm* For example: 1324287918 2011-12-19T09:45:18-0500 You can view the content of this metafile, but you cannot change it.

(Continued)

Metafile	Description
dpl.txt	Contains the data protection level (DPL) for the object. The DPL is the number of copies of the object HCP must maintain in the repository to ensure the integrity and availability of the object. For example: 2 You can view the content of this metafile, but you cannot change it.
hash.txt	Contains the name of the cryptographic hash algorithm used to generate the cryptographic hash value for the object, as well as the hash value itself. For example: SHA-256 2BC9AE8640D50145604FB6CFC45A12E5561B40429174CE404A... HCP calculates the hash value for an object from the object data. You can view the content of this metafile, but you cannot change it.
index.txt	Contains the index setting for the object. You can view and change the content of this metafile. For details on the content of the `index.txt` metafile and how to modify it, see "Index setting" on page 58.
replication.txt	Indicates whether the object is replicated, in this format: replicated=true|false The value is true only when the object and all its metadata have been replicated. For example, if you add custom metadata to a replicated object, the content of the `replicated.txt` metafile contents changes to replicated=false. When the metadata is replicated, the value changes back to replicated=true. You can view the content of this metafile, but you cannot change it.

(Continued)

Metafile	Description
retention.txt	Contains the retention setting for the object. You can view and change the content of this metafile. For details on the content of the `retention.txt` metafile and how to modify it, see "Retention" on page 39.
shred.txt	Contains the shred setting for the object. You can view and change the content of this metafile. For details on the content of the `shred.txt` metafile and how to modify it, see "Shred setting" on page 57.

(Continued)

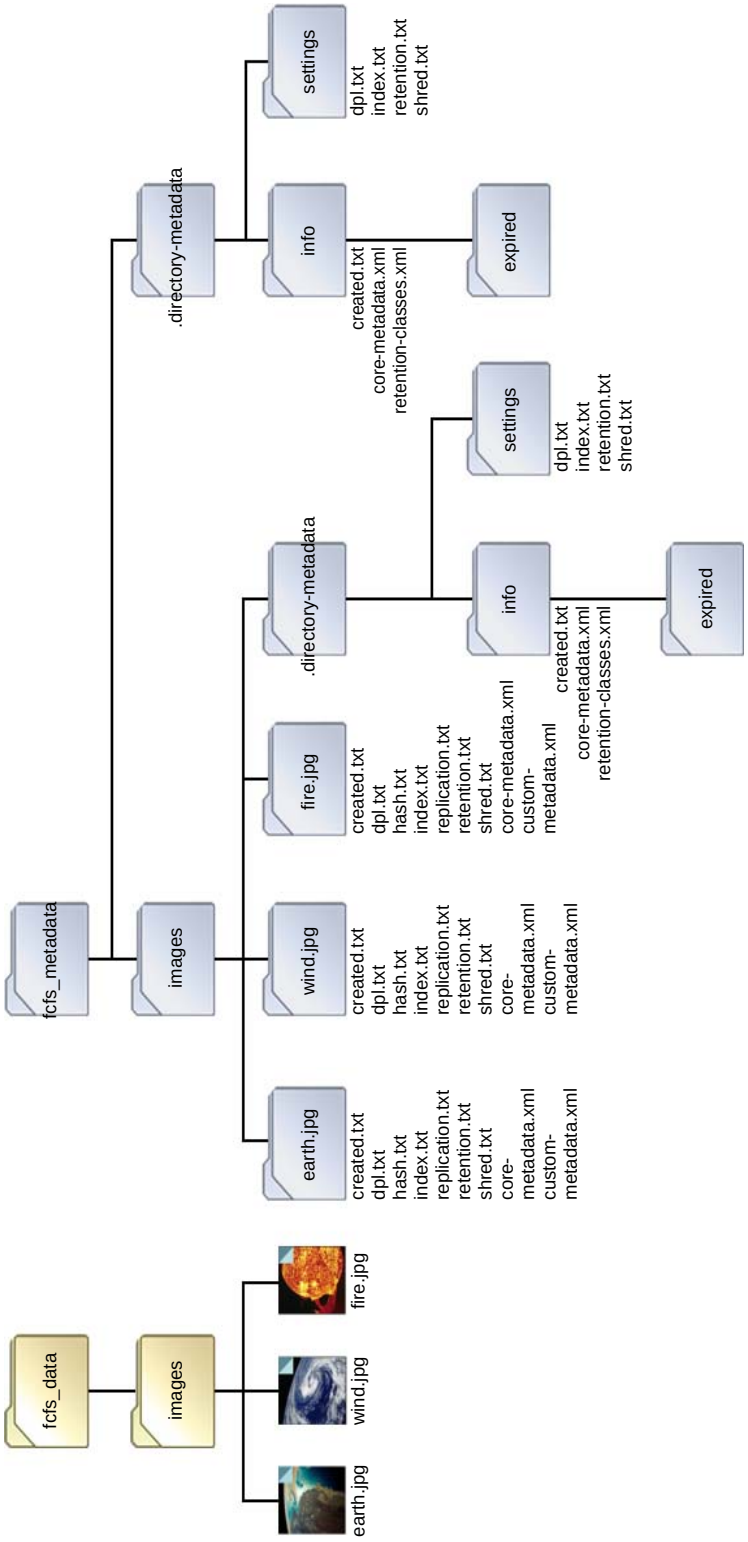
Metafile	Description
core-metadata.xml	Contains a summary of the HCP-specific and POSIX metadata for the object. For example: <core-metadata xsi:schemaLocation="http://www.hds.com core-metadata-5_0.xsd"> <version>3</version> <name>/images/wind.jpg</name> <name-bytes>2F696D616765732F77696E642E6A7067</name-bytes> <object-type>File</object-type> <creation-time>1232376318</creation-time> <update-time>1232376318</update-time> <change-time>1232376318</change-time> <access-time>1232376318</access-time> <uid>0</uid> <gid>0</gid> <mode>100544</mode> <shred>false</shred> <index>true</index> <retention-value>1462979278</retention-value> <retention-string>2016-05-11T11:07:58-0400 (DeptReg223,+7y)</retention-string> <retention-hold>false</retention-hold> <size>238985</size> <tpof>1</tpof> <dpl>2</dpl> <hash-scheme>SHA-256</hash-scheme> <hash-value>0B86212A66A792A79D58BB185EE63A4FADA76...</hash-value> <retention-class>DeptReg223</retention-class> <replicated>true</replicated> <ingestProtocol>HTTP</ingestProtocol> </core-metadata> The **version** element identifies the version of the `core-metadata.xml` file. You can view the content of this metafile, but you cannot change it. To see the XML schema for this metafile, use this URL: http://default.default.*hcp-domain-name*/static/core-metadata-5_0.xsd

(Continued)

Metafile	Description
custom-metadata.xml	Contains the custom metadata for the object. This metafile is present only when the object has custom metadata. You can add, replace, and delete custom metadata for an object only if the namespace is configured to allow it. You can view the custom metadata for an object any time. For more information on custom metadata, see "Custom metadata" on page 60.

Complete metadata structure

The figure on the next page shows the complete metadata structure, including the metafiles, generated for the sample data structure. It assumes you've added custom metadata for each of the objects.


fcfs_data
images
earth.jpg
wind.jpg
fire.jpg
fcfs_metadata
images
earth.jpg
created.txt
dpl.txt
hash.txt
index.txt
replication.txt
retention.txt
shred.txt
core-metadata.xml
custom-metadata.xml
wind.jpg
created.txt
dpl.txt
hash.txt
index.txt
replication.txt
retention.txt
shred.txt
core-metadata.xml
custom-metadata.xml
fire.jpg
created.txt
dpl.txt
hash.txt
index.txt
replication.txt
retention.txt
shred.txt
core-metadata.xml
custom-metadata.xml
.directory-metadata
info
created.txt
core-metadata.xml
retention-classes.xml
settings
dpl.txt
index.txt
retention.txt
shred.txt
expired
.directory-metadata
info
created.txt
core-metadata.xml
retention-classes.xml
settings
dpl.txt
index.txt
retention.txt
shred.txt
expired

3

Object properties

Objects in the default namespace have a number of properties, such as a retention period and an index setting. These values are defined by the object metadata. HCP maintains both HCP-specific and POSIX metadata for objects and directories. For symbolic links, it maintains only POSIX metadata. Objects can also have custom metadata, which is user supplied.

You can view all the metadata for objects and modify some of it. The way you view and modify metadata depends on what the metadata is and on which namespace access protocol you're using.

This chapter begins with an overview of the types of metadata HCP maintains for objects. It then provides detailed information about metadata you can change, including custom metadata.

Object metadata

HCP supports three types of metadata: HCP-specific, POSIX, and custom. All the metadata for an object is viewable; only some of it can be changed.

Only the owner of an object or a user with an ID of 0 (zero), otherwise known as the **root** user, can modify the HCP-specific and POSIX metadata for that object. Only a user with write permission for an object or the root user can add, replace, or delete custom metadata for that object.

For more information on object ownership and permissions, "Ownership and permissions" on page 36.

Note: Only the root user can change the ownership of an object.

HCP-specific metadata

The namespace contains HCP-specific metadata for objects and directories. HCP-specific metadata consists of:

- The date and time the object was added to the namespace. You can view this metadata in the `created.txt` metafile, but you cannot change it.

- The date and time the object was last changed. This value is returned in directory listings and you can view it in the `core-metadata.xml` metafile. The change time is the time of the most recent of these events:

  - The object was closed after being added to the namespace.

  - Any metadata, including custom metadata, was changed.

  - The object was recovered from a replica.

  - An attempt by the HCP search facility to index the object failed. When this happens, the change time for the object is set to two weeks in the future, at which time the HCP search facility tries again to index it.

  - Data was added to an appendable object.

  If an object has not changed since ingestion, this value and the value of the `created.txt` metafile may not be identical. This is because the ingest time is set when HCP opens the object for write and the change time is set when HCP closes the object after ingestion is complete

The change time is the same as the POSIX **ctime** attribute value.

- For objects, the data protection level (DPL); for directories, the DPL for objects added to the directory. The DPL specifies the number of copies of the object HCP must maintain in the repository to ensure the integrity and availability of the object. Regardless of the DPL, you see each object as a single entity.

  The DPL is set at the namespace level. You can view this metadata in the `dpl.txt` metafile, but you cannot change it. However, namespace configuration changes can cause it to change.

- For objects only, an indication of whether the object has been replicated. You can view this metadata in the `replication.txt` metafile, but you cannot change it.

- For objects only, the cryptographic hash value of the object, along with the name of the cryptographic hash algorithm used to generate that value. You can view this metadata in the `hash.txt` metafile, but you cannot change it.

- The index setting for the object. You can view and change this setting in the `index.txt` metafile. For more information on index settings, see "Index setting" on page 58.

- The retention setting for the object. You can view and change this setting in the `retention.txt` metafile. For more information on retention settings, see "Retention" on page 39.

- The shred setting for the object. You can view and change this setting in the `shred.txt` metafile. For more information on shred settings, see "Shred setting" on page 57.

For more information on metafiles, see "Metafiles" on page 21.

POSIX metadata

HCP maintains this POSIX metadata for all objects, directories, and symbolic links:

- The user ID of the user that owns the item and the group ID of the owning group. For more information on object ownership, see "Ownership and permissions" below.

- A POSIX permissions value. For more information on POSIX permissions, see "Ownership and permissions" below.

- The **atime**, **ctime**, and **mtime** attributes for the item:

  - **atime** (access time) is initially the time the item was added to the namespace. You can change the value of this attribute. For objects, changing the value of this attribute has no effect unless the **atime** attribute is synchronized with HCP retention settings.

    HCP does not automatically update the value of this attribute except when **atime** synchronization is in effect. For information on **atime** synchronization, see "atime synchronization with retention" on page 51.

  - **ctime** (change time) is the time of the last change to the item metadata. The initial value of this attribute is the time the item was added to the namespace. HCP automatically updates the value each time the item metadata changes.

    You cannot change the value of this attribute.

  - **mtime** (modify time) is initially the time the item was added to the namespace. You can change the value of this attribute. However, this has no effect on the item.

    HCP does not automatically update the value of this attribute.

Note: When the **atime** or **mtime** value of a subdirectory changes, HCP does not update the entry for the subdirectory in the parent directory listing. However, HCP does update the self entry for the subdirectory (that is, the . entry) in the subdirectory listing.

Ownership and permissions

All objects have owners, owning groups, and permissions that follow POSIX standards:

- Each object is associated with one owner, represented by a user ID, and one owning group, represented by a group ID.

  User IDs and group IDs are integers greater than or equal to zero.

- Each object is associated with a POSIX permissions value, which is made up of three sets of POSIX permissions — one for the user identified by the POSIX user ID, one for the group identified by the POSIX group ID, and one for all others. A set of permissions is any combination of read, write, or execute, including none.

POSIX permissions determine the actions users can perform on an item when accessing it on a CIFS or NFS client:

- For an object:
  - Read permission lets users view and retrieve the object content.
  - Write permission has no effect.

Note: Even if an object has write permission, its data is secure because WORM semantics prevent it from being modified.

  - Execute permission, which applies only to objects created for executable files, lets users execute the object.
- For a directory:
  - Read permission lets users see which objects are in the directory.
  - Write permission lets users add and delete objects in the directory or rename empty subdirectories.
  - Execute permission lets users traverse the directory to get to known objects in it, but it does not let users read the directory.

Viewing permissions

With the HTTP, WebDAV, or NFS protocol, permissions are represented by three 3-character strings — one for the owner, one for the owning group, and one for all others. From left to right, the three character positions in each string represent read (r), write (w), and execute (x). Each position has either the character that represents the applicable permission or a hyphen (-), meaning that the permission is denied.

For example, the string below means that the owner has all permissions for the object, the owning group has read and execute permissions, and others have only read permission:

```
-rwxr-xr--
```

The initial hyphen (-) indicates that this is an object. For a directory, the hyphen is replaced by the letter d. For a symbolic link, it is replaced by the letter l (lower case L).

Windows displays permissions in the **Security** tab in the **Properties** window for an object. These permissions don't map exactly to the POSIX permissions used in the default namespace. For information on how Windows displays the POSIX permissions associated with objects, see "CIFS permission translations" on page 171.

Octal permission values

Each permission for owner, owning group, and other has a unique octal value, as shown in the table below.

	Read	Write	Execute
Owner	400	200	100
Group	040	020	010
Other	004	002	001

You can represent permissions numerically by combining these values. For example, the octal value 755 represents these permissions:

Owner has read, write, and execute permissions (700).
Group has read and execute permissions (050).
Other has read and execute permissions (005).

You need to use these values to specify permissions when using the HTTP and WebDAV protocols. You can also use them to specify permissions when using NFS.

Ownership and permissions for new objects

HCP sets the initial owner (UID), owning group (GID), and permissions for an object when the object is added to the namespace. The values for these properties depend on the access protocol you're using. The table below describes how HCP obtains these values.

Protocol	UID and GID	Permissions
HTTP	Determined by the HTTP protocol configuration (can be overridden in the request to store the object)	Determined by the HTTP protocol configuration (can be overridden in the request to store the object)
WebDAV	Determined by the WebDAV protocol configuration	Determined by the WebDAV protocol configuration
CIFS with Active Directory® authentication	UID and GID of the logged-in user	Determined by the client

(Continued)

Protocol	UID and GID	Permissions
CIFS with anonymous access	Determined by the CIFS protocol configuration	Determined by the client
NFS	Determined by the client	Determined by the client
SMTP	Determined by the SMTP protocol configuration	Determined by the SMTP protocol configuration

Changing ownership and permissions for existing objects

If you're the root user, you can change the owner of an existing object. If you're either the owner or the root user, you can change the object permissions. You can make these changes through the HTTP (if HCP is configured to allow it), WebDAV, CIFS, and NFS protocols.

Note: The namespace can be configured to disallow ownership and permission changes for objects that are under retention.

To change ownership and permissions:

- Through HTTP, you use the HCP-specific **CHOWN** and **CHMOD** methods. For information on these methods, see "Modifying POSIX metadata" on page 125.
- Through WebDAV, you use the WebDAV **PROPPATCH** method with the HCP-specific **uid**, **gid**, and **mode** properties. For information on these properties, see "HCP-specific metadata properties for WebDAV" on page 155.
- Through NFS or CIFS, you use the standard technique for that protocol.

  For information on changing directory permissions through the CIFS protocol when using Active Directory, see "Changing directory permissions when using Active Directory" on page 172.

Retention

Both objects and directories have a retention property. For objects, this property determines how long the object must remain in the namespace before it can be deleted. This can range from allowing the object to be deleted any time to preventing the object from ever being deleted. While an object cannot be deleted due to its retention property, it is said to be **under retention**.

For a directory, the retention property determines the default retention period for new objects added to that directory.

If an object is immediately placed under retention when it's stored, it's stored with no write permissions. When an existing object is placed under retention, its write permissions are removed. For more information on permissions, see "Ownership and permissions" on page 36.

Retention periods

The **retention period** for an object is the length of time the object must remain in the repository. A retention period can be a specific length of time, infinite time, or no time, in which case the object can be deleted at any time. If you try to delete an object that's under retention, HCP prevents you from doing so.

When the retention period for an object expires, the object becomes deletable, and an entry for it appears in the `expired` metadirectory that corresponds to the directory in which the object is stored. For more information on the `expired` directory, see "Metadirectories for directories" on page 18.

Note: The namespace can be configured to allow administrative users to delete objects under retention. This is called **privileged delete**.

Special retention settings

HCP supports three special named retention settings that do not specify explicit retention periods. You can specify each setting by numeric value or name.

Value	Name	Meaning
0	Deletion Allowed	Allows the object to be deleted at any time
-1	Deletion Prohibited	Prevents the object from being deleted and its retention setting from being changed
-2	Initial Unspecified	Specifies that the object does not yet have a retention setting

Default retention settings

Each object and directory in the default namespace has a retention setting. The default retention setting for a new object is determined by the retention setting of its parent directory:

- When you add an object to the namespace, its retention setting is calculated from the retention setting of its parent directory.

- When you use SMTP to add email to the namespace, it inherits the retention setting from the namespace configuration.

- If the default retention setting is in the past, objects that would otherwise get the default setting are added with a setting of Deletion Allowed (0).

- When you add a directory to the namespace, it inherits the retention setting of its parent directory.

You can view the retention setting for an object or directory in its `retention.txt` metafile. For information on what this setting looks like, see "Retention settings in retention.txt" on page 43.

Overriding default retention settings

When you use HTTP to add an object to the namespace, you can override the default retention setting for that object. For more information on overriding default retention settings, see "Specifying metadata on object creation" on page 114.

Automatic deletion

The namespace can be configured to automatically delete objects after their retention periods expire. For an object to be deleted automatically:

- An explicit retention period must expire. Objects with a retention setting of Deletion Allowed (0) or Initial Unspecified (-2) are not automatically deleted.

- If the object is in a retention class, the class must have automatic deletion enabled.

Holding objects

You can place an object on **hold** to prevent it from being deleted. An object that is on hold cannot be deleted by any means. Holding objects is particularly useful when the objects are needed for legal discovery. Objects on hold do not appear in the `expired` metadirectory, even if their retention periods have expired.

While an object is on hold, you cannot change its retention setting. You can, however, change its shred setting. If the namespace is configured to allow changes to custom metadata for objects under retention, you can also change its custom metadata.

You can also release an object from hold. When an object is released, its previous retention setting is again in effect.

Tip: You can use the HCP Search Console to place multiple objects on hold or release multiple objects at the same time.

Retention classes

A **retention class** is a named retention value that, when used as the retention setting for an object, specifies how long the object must remain in the repository. This value can be:

- A duration after object creation. For example, a retention class named HlthReg-107 could have a duration of 21 years. All objects that have that class as their retention setting could not be deleted for 21 years after they're created.
- One of these special values:
  - Deletion Allowed (0)
  - Deletion Prohibited (-1)
  - Initial Unspecified (-2)

Retention class duration values use this format:

A+*years*y+*months*M+*days*d

In this format, A represents the time at which the object was created. For example, this value specifies a retention period of one year, two months, and three days:

A+1y+2M+3d

The duration specification can omit portions with zero values. For example, this value specifies a six-month retention period:

A+6M

You can use retention classes to consistently manage data that must conform to a specific retention rule. For example, if local law requires that medical records be kept for a specific number of years, you can use a retention class to enforce that requirement.

Namespace administrators create retention classes. When creating a class, the administrator specifies the class name, the value, and whether to automatically delete objects in the class when their retention periods expire. The administrator can also specify a description of the class.

Note: Automatic deletion must be enabled for the namespace for objects in retention classes to be automatically deleted. For more information on automatic deletion, see "Automatic deletion" on page 41.

These rules apply to retention class values:

- Administrators can increase the duration of retention classes.
- The namespace can be configured to allow administrators to decrease retention class durations or delete retention classes.
- Any change to a retention class duration changes the retention periods of objects in the class.
- If a retention class is deleted, the objects in that class have a retention setting of Deletion Prohibited (-1) and cannot be deleted.

  If a new retention class is created with the same name as a deleted retention class, existing objects in the deleted class get the retention setting of the new class.

You can assign a retention class to an object either when you create the object or at a later time. For information on the when you can replace the retention class assigned to an object, see "Changing retention settings" on page 45.

Tip: For a list of the retention classes defined for the namespace, look at the `retention-classes.xml` metafile for any directory. For information on this metafile, see "Metafiles for directories" on page 21.

Retention settings in retention.txt

The `retention.txt` metafile for an object shows you the current retention setting for that object. These settings are different for objects and directories.

retention.txt settings for an object

The table below shows the possible retention settings in `retention.txt` for an object.

0 Deletion Allowed	0 Deletion Allowed Hold
0 Deletion Allowed (*retention-class-name*, 0)	0 Deletion Allowed (*retention-class-name*, 0) Hold
-1 Deletion Prohibited	-1 Deletion Prohibited Hold
-1 Deletion Prohibited (*retention-class-name*, -1)	-1 Deletion Prohibited (*retention-class-name*, -1) Hold
-2 Initial Unspecified	-2 Initial Unspecified Hold
-2 Initial Unspecified (*retention-class-name*, -2)	-2 Initial Unspecified (*retention-class-name*, -2) Hold
retention-period-end-seconds-past-1970-1-1 *retention-period-end-datetime*	*retention-period-end-seconds-past-1970-1-1* *retention-period-end-datetime* Hold
retention-period-end-seconds-past-1970-1-1 *retention-period-end-datetime* (*retention-class-name*, *retention-class-duration*)	*retention-period-end-seconds-past-1970-1-1* *retention-period-end-datetime* (*retention-class-name*, *retention-class-duration*) Hold

retention.txt settings for a directory

The table below shows the possible retention settings in `retention.txt` for a directory.

0 Deletion Allowed
0 Deletion Allowed (*retention-class-name*, 0)
-1 Deletion Prohibited
-1 Deletion Prohibited (*retention-class-name*, -1)
-2 Initial Unspecified

(Continued)

-2 Initial Unspecified (*retention-class-name*, -2)
retention-offset
retention-offset (*retention-class-name*, *retention-class-value*)
retention-period-end-seconds-past-1970-1-1
retention-period-end-datetime

retention.txt settings for deleted retention classes

If the retention class assigned to an object or directory is deleted, the `retention.txt` metafile for the object or directory then contains:

- A retention setting of Deletion Prohibited (-1)
- The name of the deleted retention class
- A retention class value of **undefined**

For example, suppose you assign an object to the HlthReg-107 retention class, and then the class is deleted. The `retention.txt` metafile for the object then contains:

```
-1
Deletion Prohibited (HlthReg-107, undefined)
```

If a new retention class named HlthReg-107 is created, existing objects and directories assigned to the HlthReg-107 retention class get the retention setting of the new class.

Changing retention settings

If you're either the owner of an object or directory or the root user, you can change its retention setting:

- For an object:
  - If the object is under retention, you can change its retention setting to lengthen the retention period but not to shorten it.
  - If the object is not under retention, you can change its retention setting to any time — past or present. If you change it to a time in the past, the object is immediately deletable.

- For a directory, you can change the setting to any valid value (see the table below). Changing the retention setting for a directory affects only new objects and directories added to the directory. It does not affect any existing objects or directories.

- For an object that's in a retention class, you can replace the class with another class with an equal or greater retention period, but you cannot replace the class with an explicit retention setting, such as -1 (Deletion Prohibited) or a specific date and time.

To change the retention setting for an object, you overwrite its `retention.txt` metafile. In the new file, you specify a single value that tells HCP what change to make. This value must be on a single line. To ensure that HCP processes the value correctly, end the line with a carriage return.

Tip: With Windows and Unix, you can also use the **echo** command to insert the new value into the `retention.txt` metafile.

The table below shows the values you can use to change the retention setting for an object. These values are not case sensitive.

Value	Effect
0 (zero) *or* Deletion Allowed	Allows the object to be deleted at any time. You can assign this value to an object only when you add it to the namespace or when its retention setting is -2. You can assign it to a directory at any time. The value -0 is equivalent to 0 (zero).
-1 *or* Deletion Prohibited	Prevents the object from being deleted and its retention setting from being changed. The object is stored permanently. You can assign this value to an object or directory at any time. If an object is assigned to a retention class and that class is then deleted, the retention setting for that object changes to -1.
-2 *or* Initial Unspecified	Specifies that the object does not yet have a retention setting. You can assign this value to a directory at any time. You can assign it directly to an object when you add the object to the namespace with HTTP (see "Specifying metadata on object creation" on page 114). You can also directly change the retention setting for an object from 0 to -2. While an object has a retention setting of -2, you cannot delete it. You can change -2 to any other retention setting for both objects and directories.

(Continued)

Value	Effect
datetime	Prevents the object from being deleted until the specified date and time. You can assign this value to an object if the specified date and time is later than the current retention setting for the object. You cannot assign it to an object for which the current retention setting is -1. You can assign this value to a directory at any time, as long as the specified date and time are later than the current date and time. For a description of the *datetime* format, see "Specifying a date and time" below.  **Note:** If the retention setting for a directory becomes earlier than the current time (due to the passage of time), objects added to that directory are immediately expired and, therefore, deletable.
offset	Prevents the object from being deleted until the date and time derived from the specified offset. You can assign this value to an object at any time, except when its current retention setting is -1. You can assign this value to a directory at any time. For an object, an offset is used to calculate a new retention setting. As a result, when you next look in `retention.txt`, you see the calculated value, not the specified offset. For a directory, the specified offset becomes the retention setting. As a result, when you next look in `retention.txt`, you see the offset specification. For a description of the *offset* format, see "Specifying an offset" on page 49.

(Continued)

Value	Effect
C+*retention-class-name*	Prevents the object from being deleted until the period of time specified by the retention class has elapsed. You can assign this value to an object if any one of these is true: • The current retention period for the object has expired. • The current retention period for the object has not expired, and the retention class results in a retention period that is longer than the current retention period. • The current retention setting for the object is 0 or -2. • The current retention setting for the object is -1 and the class has a value of -1. • The object is in a retention class with a value of 0 or -2 and the new class has a value of 0 or -2. • The object is in a retention class and the new class either doesn't change or increases the retention period for the object. For purposes of comparison, a class with a retention value of -1 has the longest possible retention period and a class with a retention value of 0 has the shortest possible retention period. You can assign this value to a directory at any time. The retention class you assign must already be defined for the namespace. Retention class names are not case sensitive.
Hold	Prevents the object from being deleted until it is released. You can assign this value to an object at any time. You cannot assign this value to a directory.
Unhold	Releases an object that's on hold. When an object is released, its previous retention setting is again in effect.

Specifying a date and time

You can set retention by specifying a date and time in either of these formats:

- Time in seconds since January 1, 1970, at 00:00:00. For example:

  1450137600

  The calendar date that corresponds to 1450137600 is Tuesday, December 15, 2015, at 00:00:00 EST.

- Date and time in this ISO 8601 format:

  yyyy-*MM*-*dd*T*hh*:*mm*:*ss*Z

  Z represents the offset from UTC and is specified as:

  (+|-)*hhmm*

  For example, 2015-11-16T14:27:20-0500 represents the start of the 20th second into 2:27 PM, November 16, 2015, EST.

If you specify certain forms of invalid dates, HCP automatically adjusts the retention setting to make a real date. For example, if you specify 2015-11-33, which is three days past the end of November, HCP changes it to 2015-12-03.

Specifying an offset

You can set retention by specifying an offset from:

- The time at which the object was added to the namespace
- The current retention setting for the object
- The current time

Because you can only extend a retention period, the offset must be a positive value.

Offset syntax

To use an offset as a retention setting, specify a standard expression that conforms to this syntax:

^([**RAN**])?([**+-**]\d+**y**)?([**+-**]\d+**M**)?([**+-**]\d+**w**)?([**+-**]\d+**d**)?([**+-**]\d+**h**)?([**+-**]\d+**m**)?([**+-**]\d+**s**)?

The table below explains this syntax.

Character	Description
^	Start of the expression
( )	Sequence of terms treated as a single term
?	Indicator that the preceding term is optional
[ ]	Group of alternatives, exactly one of which must be used
+	Plus

(Continued)

Character	Description
-	Minus
R*	The current retention setting for the object. **R** is meaningful only when changing the retention setting for an object
A*	The time at which the object was added to the repository
N*	The current time
\d+	An integer in the range 0 (zero) through 9,999
y	Years
M	Months
w	Weeks
d	Days
h	Hours
m	Minutes
s	Seconds
*__R__, __A__, and __N__ are mutually exclusive. If you don't include any of them, the default is __R__.	

In an expression that specifies an offset:

- The time measurements must go from the largest unit to the smallest (that is, in the order in which they appear in the syntax).
- **R**, **A**, **N**, and the characters that represent time measurements are case sensitive.
- You can omit a sequence of terms in which \d+ is zero.

Tip: When you add an object to the namespace or change the retention setting for a directory, **R**, **A**, and **N** are equivalent; that is, they all represent the current date and time. Because **A** and **N** are more intuitively meaningful, you should use either one of them instead of **R** for these purposes.

Offset examples

Here are some examples of using an offset to extend a retention period; these examples use the NFS protocol:

- This command changes the retention setting for the `images` directory to 100 years past the time objects are added to that directory:

  ```
  echo "A+100y"  >  /metadatamount/images/.directory-metadata/settings/retention.txt
  ```

- This command sets the end of the retention period for the wind.jpg object to 20 days minus five hours past the current date and time:

  ```
  echo "N+20d-5h"  >  /metadatamount/images/wind.jpg/retention.txt
  ```

- This command extends the current retention period for wind.jpg by two years and one day:

  ```
  echo "R+2y+1d"  >  /metadatamount/images/wind.jpg/retention.txt
  ```

atime synchronization with retention

Some file systems support the use of the POSIX **atime** attribute to set retention. To take advantage of this existing mechanism, HCP gives you the option of synchronizing **atime** values with HCP retention settings. When these properties are synchronized, changing one for an object causes an equivalent change in the other.

Your namespace administrator enables or disables **atime** synchronization for the namespace. While **atime** synchronization is enabled, **atime** values are automatically synchronized with retention settings for objects *subsequently added* to the namespace except in these cases:

- The object is added through NFS with an initial retention setting of Deletion Allowed.
- The object is added through any protocol with an initial retention setting that is either Initial Unspecified or a retention class.
- The namespace supports appendable objects.

In these cases, the **atime** value of an object is set to the time the object is stored.

For any given object, if **atime** synchronization was not enabled automatically, you can enable it manually. For more information on doing this, see "Triggering atime synchronization for existing objects" on page 52.

While **atime** synchronization is enabled for the namespace, the rules for changing retention settings also apply to changing **atime** values. You cannot use **atime** to shorten a retention period, nor can you use it to specify a retention period if the current setting is **Deletion Prohibited**. Additionally, you cannot change the **atime** value if the object is on hold.

Note: If both **atime** synchronization is enabled and appendable objects are supported, do not use `retention.txt` to change object retention settings. Use only the **atime** attribute.

atime synchronization does not work with objects in retention classes. When you assign an object to a retention class, the **atime** value for the object does not change, even if the **atime** value had previously been synchronized with the retention setting. Triggering **atime** synchronization for an object in a retention class has no effect.

By default, **atime** synchronization is disabled when the namespace is created. Ask your namespace administrator whether it has been enabled.

Note: With **atime** synchronization enabled, you cannot rename empty directories. This includes any directories you create using CIFS, which, by default, are named `New Folder`.

Triggering atime synchronization for existing objects

While **atime** synchronization is enabled, you can use either the **atime** attribute or the `retention.txt` metafile to trigger synchronization for an individual existing object (for which synchronization is not currently in effect):

- To use the **atime** attribute to trigger synchronization for an object with retention currently set to a specific date and time in the future, use HTTP, CIFS, or NFS to make a valid change to the value of the **atime** attribute.

- To use the **atime** attribute to trigger synchronization for an object with retention currently set to either **Deletion Allowed** or **Initial Unspecified**:

  1. Check that the **atime** value is the retention setting you want. If it isn't, be sure to change it before performing step 2.

2. Do one of these:

   – If the object has any write permissions for the owner, owning group, or other, use HTTP, CIFS, or NFS to remove them.

   – If the object has no write permissions for the owner, owning group, or other, use HTTP, CIFS, or NFS to add at least one and then remove all you added.

   Changing permissions through WebDAV does not trigger **atime** synchronization.

Note: Read and execute permissions have no effect on this process.

Important:

- Some commands you can use to add objects to the namespace, such as the Unix **cp** command, have options for preserving the original permissions and timestamps. These commands may modify permissions and **atime** values in a way that causes the stored object to become read-only, thereby triggering retention. If **atime** synchronization is enabled, you should review the use of these commands to ensure they do not result in unexpected retention settings.

- Existing applications ported to HCP may remove write permissions from objects without regard to their **atime** values. If **atime** synchronization is enabled for the namespace, this can have the unintended effect of giving some objects infinite retention. Be sure to review ported applications for this behavior before running them.

- To use the `retention.txt` metafile to trigger **atime** synchronization for any object, regardless of its current retention setting, change the retention setting for the object to any valid value except a retention class.

Triggering **atime** synchronization for an object creates an association between its **atime** value and retention setting. Subsequent changes to POSIX permissions do not remove this association.

Removing the association

The association between the **atime** value and retention setting for an object remains in effect until one of these happens:

- The retention period for the object expires.
- You assign the object to a retention class.
- Your namespace administrator disables **atime** synchronization for the namespace. Any changes to **atime** attributes or retention settings made after synchronization is disabled are independent of each other. If your namespace administrator subsequently reenables **atime** synchronization for the namespace, these properties remain independent until you trigger synchronization again for the individual object, as described in "Triggering atime synchronization for existing objects" above.

How atime synchronization works

atime values and retention settings have these general correspondences:

- **atime** values in the future correspond to specific retention settings in the future.
- **atime** values more than 24 hours in the past correspond to retention settings that prevent deletion.
- **atime** values 24 hours or less in the past correspond to retention settings that permit immediate deletion.

The table below shows the effects of valid **atime** changes on retention settings for an object with **atime** synchronization in effect.

Changing the atime value to	When the current retention setting is	Changes the retention setting to
A time later than the current date and time	**Initial Unspecified** (-2) or **Deletion Allowed** (0)	The new time of the **atime** attribute
A time later than the current retention setting	A specific date and time	The new time of the **atime** attribute
A time more than 24 hours before the current date and time*	**Initial Unspecified** (-2), **Deletion Allowed** (0), or a specific date and time	**Deletion Prohibited**
A time 24 hours or less before the current date and time*	**Initial Unspecified** (-2) or **Deletion Allowed** (0)	The new time of the **atime** attribute, which immediately makes the object expired and deletable
*Twenty-four hours is the default setting for this threshold. If you want it changed, please contact your namespace administrator.		

The table below shows the effects of valid changes to retention settings on **atime** values for an object with **atime** synchronization in effect.

Changing the retention setting to	When the current retention setting is	Changes the atime value to
A time later than the current date and time	**Initial Unspecified** (-2) or **Deletion Allowed** (0)	The same time as the new retention setting
A time later than the current retention setting	A specific date and time	The same time as the new retention setting
A time before the current date and time	**Initial Unspecified** (-2) or **Deletion Allowed** (0)	The same time as the new retention setting
Deletion Allowed	**Initial Unspecified** (-2)	January 1, 1970 00:00:00 GMT
Deletion Prohibited	**Initial Unspecified** (-2), **Deletion Allowed** (0), or a specific date and time	The current date and time minus the threshold beyond which **atime** sets retention to **Deletion Prohibited***
*Twenty-four hours is the default setting for this threshold. If you want it changed, please contact your namespace administrator.		

If **atime** synchronization has already been triggered for an object and the object is under retention, you cannot use **atime** to change its retention setting while HCP is configured to disallow permission changes for objects under retention. However, you can modify the setting in `retention.txt`, and, when you do so, the **atime** value is synchronized with the new retention setting.

atime synchronization example

The following example shows how to use the **atime** attribute to trigger retention for the existing `wind.jpg` object after **atime** synchronization has been enabled for the namespace; the example uses the NFS protocol:

1. Optionally, check the current retention setting for the `wind.jpg` object:

   ```
   cat /metadatamount/images/wind.jpg/retention.txt
   0
   Deletion Allowed
   ```

2. Optionally, check the current permissions for the wind.jpg object:

   ```
   ls -l /datamount/images/wind.jpg
   -r--r--r--  1 root root 23221 Nov 19 09:45 /datamount/images/wind.jpg
   ```

   Notice that the object has no write permissions.

3. Set the **atime** attribute for the `wind.jpg` object:

   ```
   touch -a -t 201512310000 /datamount/images/wind.jpg
   ```

Note: To set the value of the **atime** attribute, you can use the HTTP **TOUCH** method, Windows **SetFileTime** library call, the Unix **utime** library call, or the Unix **touch** command.

4. Optionally, verify step 4:

   ```
   stat /datamount/images/wind.jpg
     File: "/datamount/images/wind.jpg"
     Size: 23221           Blocks: 112        IO Block: 32768  regular file
   Device: 15h/21d Inode: 18          Links: 1
   Access: (0444/-r--r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
   Access: 2015-12-31 00:00:00.000000000 -0500
   Modify: 2011-11-19 09:45:18.000000000 -0500
   Change: 2011-11-23 13:10:17.000000000 -0500
   ```

5. Add write permissions to the `wind.jpg` object:

   ```
   chmod a+w /datamount/images/wind.jpg
   ```

6. Remove all write permissions from the `wind.jpg` object:

   ```
   chmod a-w /datamount/images/wind.jpg
   ```

7. Optionally, verify that the retention setting has changed to match the **atime** value:

   ```
   cat /metadatamount/images/wind.jpg/retention.txt
   1451520000
   2015-12-31T00:00:00-0500
   ```

Shred setting

Shredding, also called **secure deletion**, is the process of deleting an object and overwriting the places where its copies were stored in such a way that none of its data or metadata, including custom metadata, can be reconstructed.

About shred settings

Every object has a shred setting that determines whether it will be shredded when it's deleted. Every directory has a shred setting. This setting is the default shred setting for each object added to the directory. However, email stored using SMTP gets its shred setting from the namespace configuration.

When you use HTTP to add an object to the namespace, you can override the default shred setting for that object as well as for any new directories in the object path. For more information on overriding default shred settings, see "Specifying metadata on object creation" on page 114.

You can view the shred setting for an object in its `shred.txt` metafile. In this metafile:

- A value of **0** (zero) means don't shred.
- A value of **1** (one) means shred.

By default, the shred setting for the `fcfs_data` directory is zero.

Tip: As a general rule, multiple objects with the same content should all have the same shred setting.

Changing shred settings

If you're either the owner of an object or the root user, you can change its shred setting:

- For an object, you can change the shred setting from **0** (zero) to **1** (one) but not from **1** (one) to **0** (zero).
- For a directory, you can change the shred setting either way.

Changing the shred setting for a directory affects only new objects added to the directory. It does not affect any existing objects. If you want any of the existing objects to be shredded, you need to change their shred settings individually.

To change the shred setting for an object, you overwrite its `shred.txt` metafile. In the new file, you specify only the new value.

Tip: With Windows and Unix, you can also use the **echo** command to insert the new value into the `shred.txt` metafile.

Index setting

Each object has an index setting that is either **true** or **false**. The setting is present regardless of whether the namespace supports search operations.

The metadata query engine uses the index setting to determine whether to index custom metadata for an object:

- For objects with an index setting **true**, the metadata query engine indexes custom metadata.
- For objects with an index setting **false**, the metadata query engine does not index custom metadata.

The HCP search facility uses the index setting to determine whether to index an object at all:

- The HCP search facility indexes objects with an index setting **true**.
- The HCP search facility does not index objects with an index setting **false**.

Metadata query API requests can use the index setting as a search criterion. Additionally, third-party applications can use this setting for their own purposes.

You can view the index setting for an object or directory in its `index.txt` metafile. In this metafile:

- A value of **1** (one) means true.
- A value of **0** (zero) means false.

Default index settings

Every directory has an index setting. This setting is the default index setting for each object added to the directory. However, email stored using SMTP gets its index setting from the namespace configuration.

When you use HTTP to add an object to the namespace, you can override the default index setting for that object as well as for any new directories in the object path. For more information on overriding default index settings, see "Specifying metadata on object creation" on page 114.

By default, the index setting for the `fcfs_data` directory is 1 (one).

Changing index settings

If you're either the owner of an object or directory or the root user, you can change its index setting.

Changing the index setting for a directory affects only new objects added to the directory. It does not affect any existing objects. To change the index setting of an existing object, change the setting for that object.

Changing the index setting for an object causes these changes to the indexes maintained by the metadata query engine and the HCP search facility:

- If you change the index setting of an object from **true** to **false**:
  - The metadata query engine removes the custom metadata for the object from its index.
  - If the HCP search facility is enabled, it removes the object from its index.
- If you change the index setting of an object from **false** to **true**:
  - The metadata query engine indexes the custom metadata for the object.
  - If the HCP search facility is enabled, it indexes the object.

To change the index setting for an object or directory, you overwrite its `index.txt` metafile. In the new file, you specify only the new value.

Tip: With Windows and Unix, you can also use the **echo** command to insert the new value into the `index.txt` metafile.

Custom metadata

Custom metadata is user-supplied descriptive information about an object. You store this metadata in `custom-metadata.xml` in the metadirectory that corresponds to the object. For example, the custom metadata file for the `wind.jpg` object in the `images` directory is:

```
fcfs_metadata/images/wind.jpg/custom-metadata.xml
```

Custom metadata is stored as a unit. You can add, replace, or delete it in its entirety. You cannot modify it in place.

Custom metadata files

Custom metadata is typically specified using XML, but this is not required. The namespace configuration determines whether HCP checks that custom metadata is well-formed XML. While checking is enabled, if you try to store custom metadata that is not well-formed XML, HCP rejects it.

Here's an example of custom metadata that is well-formed XML:

```
<?xml version="1.0" ?>
<weather>
    <location>Massachusetts</location>
    <date>20110130</date>
    <duration_secs>180</duration_secs>
    <temp_F>
        <temp_high>31</temp_high>
        <temp_low>31</temp_low>
    </temp_F>
    <velocity_mph>
        <velocity_high>22</velocity_high>
        <velocity_low>17</velocity_low>
    </velocity_mph>
</weather>
```

The metadata query engine and the HCP search facility can index the content of custom metadata. In this case, users can search for objects based on their custom metadata. If such a search results in a match, HCP returns the object for which the custom metadata was stored, not the `custom-metadata.xml` file.

Working with custom metadata

To add, replace, and delete custom metadata, you need to use the HTTP or WebDAV protocol. HCP does not support custom metadata operations through other protocols. For an example of storing custom metadata with HTTP, see "Storing custom metadata" on page 131.

With the HTTP protocol, you can use a single request to store or retrieve both object data and custom metadata. With WebDAV, you need to store or retrieve the custom metadata separately from the object data.

With WebDAV, you can use the `custom-metadata.xml` metafile to store dead properties. For more information on storing dead properties as custom metadata, see "Using the custom-metadata.xml file to store dead properties" on page 160.

The namespace configuration determines what you can do with custom metadata for objects that are under retention. The namespace can be set to:

- Allow all custom metadata operations for objects under retention
- Allow only the addition of new custom metadata for objects under retention and disallow replacement or deletion of existing custom metadata
- Disallow adding, replacing, or deleting custom metadata for objects under retention

Errors when saving custom metadata

An attempt to save custom metadata may fail with a 400 (Bad Request) error in any of these cases:

- The namespace has custom metadata XML checking enabled, and the custom metadata is not well-formed XML.
- The custom metadata XML has a large number of different elements and attributes.

  In this case, try restructuring the XML to have fewer different elements and attributes. For example, try concatenating multiple element values, such as the different parts of an address, to create a new value for a single element.

  If you cannot restructure the XML to prevent failures, ask your namespace administrator about reconfiguring the namespace to prevent HCP from checking whether custom metadata XML is well formed.

- A number of clients try to store custom metadata for multiple objects at the same time.

  In this case, limit the number of concurrent requests from clients to the namespace.

Replication collisions

If users can write to multiple systems in a replication topology, collisions can occur when different changes are made to the same objects on different systems and those changes are then replicated. The way HCP handles collisions that occur due to replication depends on the type of collision. However, the general rule is that more recent changes have priority over conflicting less recent changes.

For an introduction to replication, see "Replication" on page 9.

Object content collisions

An object content collision occurs when these events occur in the order shown:

1. An object is created with the same name in the same directory on two systems in a replication topology, but the object has different content on the two systems.

2. The object on one of the systems is replicated to the other system.

When an object content collision occurs, the more recently created object keeps its name and location. The other object is either moved to the `.lost+found` directory or renamed, depending on the namespace configuration.

When HCP moves an object to the `.lost+found` directory, the full object path becomes `.lost+found/replication/`*`system-generated-directory/old-object-path`*.

When renaming an object due to a content collision, HCP changes the object name to *`object-name`*`.collision`. If the new name is already in use, HCP changes the object name to *`object-name`*`.1.collision`. If that name is already in use, HCP successively increments the middle integer by one until a unique name is formed.

Objects that have been relocated or renamed due to content collisions are flagged as replication collisions in their system metadata. The **replicationCollision** element in the `core-metadata.xml` file indicates whether an object is flagged as a replication collision.

You can use the metadata query API to search for objects that are flagged as replication collisions. For information the metadata query API, see *HCP Metadata Query API Reference*.

If an object that's flagged as a replication collision changes (for example, if its retention period is extended), its collision flag is removed. If you create a copy of a flagged object with a new name, the collision flag is not set on the copy.

Depending on the namespace configuration, objects flagged as replication collisions may be automatically deleted after a set number of days. The days are counted from the time the collision flag is set. If the collision flag is removed from an object, the object is no longer eligible for automatic deletion.

System metadata collisions

A system metadata collision occurs when these events occur in the order shown:

1. Different changes are made to the system metadata for a given object on each of two systems in a replication topology.

2. The changed system metadata on one of the systems is replicated to the other system.

For example, suppose a user on one system changes the shred setting for an object while a user on the other system changes the index setting for the same object. When the object on either system is replicated to the other system, a system metadata collision occurs.

If a collision occurs when changed system metadata for a given object is replicated from one system (system A) in a replication topology to another system (system B) in the topology:

- For changed system metadata other than the retention setting and hold status:

    - If the last change made on system A is more recent than the last change made on system B, HCP changes the system metadata on system B to match the system metadata on system A.

  - If the last change on system B is more recent than the last change on system A, HCP does not change the system metadata on system B.

- For a changed retention setting:

  - If the retention setting on system A specifies a longer retention period than does the retention setting on system B, HCP changes the retention setting on system B to match the retention setting on system A.

  - If the retention setting on system B specifies a longer retention period than does the retention setting on system A, HCP does not change the retention setting on system B.

- For a changed hold status:

  - If the object is on hold on system A but not on system B, HCP places the object on hold on system B.

  - If the object is on hold on system B but not on system A, HCP leaves the object on hold on system B.

Here are some examples of how HCP handles collisions when changed system metadata for a given object is replicated from one system (system A) in a replication topology to another system (system B) in the topology.

Example 1

The object starts out on both system A and system B with these system metadata settings:

Shred: false
Index: false

The table below shows a sequence of events in which the system metadata for the object is changed and the changes are then replicated.

Sequence	Event
1	On system A, a client changes the shred setting to true.
2	On system B, a client changes the index setting to true.
3	The changes on system A are replicated to system B. The resulting settings for the object on system B are: Shred: false Index: true

Example 2

The object starts out on both system A and system B with these system metadata settings:

Retention: Initial Unspecified
Shred: false
Index: false

The table below shows a sequence of events in which the system metadata for the object is changed and the changes are then replicated.

Sequence	Event
1	On system A, a client changes the retention setting to Deletion Prohibited.
2	On system B, a client changes the retention setting to Deletion Allowed.
3	On system B, a client changes the index setting to true.
4	On system A, a client changes the shred setting to true.
5	The changes on system A are replicated to system B. The resulting settings for the object on system B are: Retention: Deletion Prohibited Shred: true Index: false

Example 3

The object starts out on both system A and system B with these system metadata settings:

Retention: Initial Unspecified
Hold: true
Shred: false
Index: false

The table below shows a sequence of events in which the system metadata for the object is changed and the changes are then replicated.

Sequence	Event
1	On system A, a client changes the retention setting to Deletion Allowed.
2	On system B, a client changes the retention setting to Deletion Prohibited.
3	On system B, a client changes the index setting to true.
4	On system A, a client changes the shred setting to true.

(Continued)

Sequence	Event
5	On system A, a client releases the object from hold.
6	The changes on system A are replicated to system B. The resulting settings for the object on system B are: Retention: Deletion Prohibited Hold: true Shred: true Index: false
7	The changes on system B are replicated to system A. The resulting settings for the object on system A are: Retention: Deletion Prohibited Hold: true Shred: true Index: false

Custom metadata collisions

A custom metadata collision occurs when these events occur in the order shown:

1. One of these changes occurs:
   - Custom metadata is added to a given object on each of two systems in a replication topology, but the added custom metadata is different on the two systems.

     The addition of custom metadata to an object on only one of the systems does not result in a custom metadata collision. Instead, the new custom metadata is replicated from that system to the other system without conflict.
   - The custom metadata for a given object is replaced on each of two systems in a replication topology, but the replacement custom metadata is different on the two systems.
   - The custom metadata for a given object is replaced on one system in a replication topology, and the same custom metadata is deleted on another system in the topology.
2. The change made on one of the systems is replicated to the other system.

Custom metadata is treated as a single unit. If a collision occurs when a custom metadata change for a given object is replicated from one system (system A) in a replication topology to another system (system B) in the topology:

- If the last change on system A is more recent than the last change on system B, HCP applies the change from system A to the custom metadata on system B

- If the last change on system B is more recent than the last change on system A, HCP does not change the custom metadata on system B

For example, suppose a given object starts out with the same custom metadata on system A and system B. The table below shows a sequence of events in which the custom metadata for the object is changed and the change is then replicated.

Sequence	Event
1	On system B, a client replaces the custom metadata for the object with new custom metadata.
2	On system A, a client replaces the custom metadata for the object with different custom metadata from the custom metadata used on system B.
3	The change on system A is replicated to system B. The resulting custom metadata for the object on system B is the new custom metadata from system A.

4

HTTP

HTTP is one of the industry-standard protocols HCP supports for namespace access. To access the namespace through HTTP, you can write applications that use any standard HTTP client library, or you can use a command-line tool, such as cURL, that supports HTTP. You can also use HTTP to access the namespace directly from a web browser.

Using the HTTP protocol, you can store, view, retrieve, and delete objects. You can override certain metadata when you store new objects. You can also add, replace, and delete custom metadata, as well as change certain system metadata for existing objects.

HCP is compliant with HTTP/1.1, as specified by RFC 2616.

For you to access the namespace through HTTP, this protocol must be enabled in default namespace configuration. If you cannot access the namespace in this way, see your namespace administrator.

This chapter explains how to use HTTP for namespace access. It does not cover the HCP metadata query API, which uses the HTTP POST method to retrieve metadata for objects that match specified query criteria. For information on the metadata query API, see *HCP Metadata Query API Reference*.

The examples in this chapter use cURL and Python with PycURL, a Python interface that uses the libcurl library. cURL and PycURL are both freely available open-source software. You can download them from http://curl.haxx.se.

Note: In version 7.12.1 of PycURL, the **PUT** method was deprecated and replaced with **UPLOAD**. The Python examples in this book show **UPLOAD** but work equally well with **PUT**.

For a condensed reference of the HTTP methods you use and responses you get when accessing a namespace, see Appendix A, “HTTP reference,” on page 199.

URLs for HTTP access to a namespace

Depending on the method you’re using and what you want to do, the URL you use for namespace access can identify any of:

- The namespace as a whole
- A directory
- An object
- A symbolic link
- A metadirectory
- A metafile for an object or directory

URL formats

The following sections show the URL formats you can use for default namespace access. In these formats, you can identify the HCP system by either DNS name or IP address. For information on the relative advantages of DNS names and IP addresses, see “DNS name and IP address considerations” on page 194.

If the HCP system does not support DNS, you can use the client `hosts` files to enable access to the default namespace by hostname. For information on configuring hostnames on the client system, see “Using a hosts file” on page 193.

Notes:

- The URL formats and examples that follow show *http*. Your namespace administrator can configure the namespace to require SSL security for the HTTP protocol. In this case, you need to specify *https* instead of *http* in your URLs.
- If you use HTTPS, check with your namespace administrator as to whether you need to disable SSL certificate verification. For example, with cURL you may need to use the **-k** or **--insecure** option.

URL for the namespace as a whole

A URL that identifies the namespace as a whole has one of these formats:

http://default.default.*hcp-domain-name*

http://*node-ip-address*

For example:

http://default.default.hcp.example.com

URLs for objects, directories, and symbolic links

To access an object, directory, or symbolic link in the default namespace, you use a URL that includes the `fcfs_data` directory. The format for this is one of:

http://default.default.*hcp-domain-name***/fcfs_data**[*/directory-path* [*/object-name*]]

http://*node-ip-address***/fcfs_data**[*/directory-path*[*/object-name*]]

Here's a sample URL that identifies a directory:

http://default.default.hcp.example.com/fcfs_data/Corporate/Employees

Here's a sample URL that identifies an object:

http://192.168.210.16/fcfs_data/Corporate/Employees/23_Jon_Doe

You cannot tell from a URL whether it represents an object, directory, or symbolic link.

URLs for metafiles and metadirectories

To access a metafile or metadirectory, you use a URL that includes the `fcfs_metadata` metadirectory. The format for this is one of:

http://default.default.*hcp-domain-name***/fcfs_metadata/** *metadirectory-path*[*/metafile-name*]

http://*node-ip-address***/fcfs_metadata/***metadirectory-path* [*/metafile-name*]

Here's a sample URL that identifies a metadirectory:

http://192.168.210.16/fcfs_metadata/Corporate/Employees/2193_John_Doe

Here's a sample URL that identifies a metafile:

http://default.default.hcp.example.com/fcfs_metadata/Corporate/Employees/2193_John_Doe/shred.txt

URL considerations

The following considerations apply to specifying URLs in HTTP requests against the default namespace. For considerations that apply specifically to naming new objects, see "Object naming considerations" on page 16.

URL length

For all HTTP methods, the portion of a URL after `fcfs_data` or `fcfs_metadata`, excluding any appended query parameters, is limited to 4,095 bytes. If an HTTP request includes a URL that violates that limit, HCP returns a status code of 414.

URL character case

All elements of a URL except the hostname are case sensitive. This includes the `fcfs_data` and `fcfs_metadata` elements.

Names with non-ASCII, nonprintable characters

When you store an object or directory with non-ASCII, nonprintable characters in its name, those characters are percent encoded in the name displayed back to you. In the `core-metadata.xml` file for an object, those characters are also percent encoded, but the percent signs (%) are not displayed.

Regardless of how the name is displayed, the object or directory is stored with its original name, and you can access it either by its original name or by the name with the percent-encoded characters.

Percent-encoding for special characters

Some characters have special meaning when used in a URL and may be interpreted incorrectly when used for other purposes. To avoid ambiguity, percent-encode the special characters listed in the table below.

Character	Percent-encoded value
Space	%20
Tab	%09
New line	%0A
Carriage return	%0D
+	%2B

(Continued)

Character	Percent-encoded value
%	%25
#	%23
?	%3F
&	%26

Percent-encoded values are not case sensitive.

Note: Do not percent-encode query parameters appended to URLs. For information on these parameters, see "Specifying metadata on object creation" on page 114.

Quotation marks with URLs in command lines

When using a command-line tool to access the namespace through HTTP, you work in a Unix, Mac OS® X, or Windows shell. Some characters in the commands you enter may have special meaning to the shell. For example, the ampersand (&) used in URLs to join multiple query parameters also often indicates that a process should be put in the background.

To avoid the possibility of the Windows, Unix, or Mac OS X shell misinterpreting special characters in a URL, always enclose the entire URL in double quotation marks.

Access with a cryptographic hash value

If an object is indexed by the search facility selected for use with the Search Console, you can use the cryptographic hash value for the object to identify it in a URL instead of using the object name. The format for this is:

http://default.default.*hcp-domain-name/hash-algorithm/hash-value*

In this format:

- *hash-algorithm* is the name of the cryptographic hash algorithm used to calculate the hash value.
- *hash-value* is the cryptographic hash value for the object.

The hash algorithm name and the hash value are not case sensitive.

For example:

http://default.default.hcp.example.com/SHA-256/E3B0C44298FC1C149AFBF4C899...

Note: If the Search Console is using the HDDS search facility, the ability to access objects by their cryptographic hash values depends on how that facility is configured.

How to get the cryptographic hash value

To get the cryptographic hash value of an object, you can:

- Compute the hash value on the original file using a publicly available tool such as SlavaSoft **FSUM**. Be sure to use the same cryptographic hash algorithm as HCP uses.
- Look at the value in the `hash.txt` metafile for the object.
- Find the hash value in the X-ArcHash response header returned for the HTTP **PUT** request used to store the object. For information on this response header, see "Request-specific response headers" on page 81.

Response headers for multiple matching objects

Although unlikely, the namespace can contain multiple objects with the same cryptographic hash value. As a result, the hash value you specify in the URL in an HTTP request may identify more than one object. If it does, HCP returns a status code of 300, and the response headers include:

X-DocCount: *n*
X-DocURI-0: /fcfs_data/*object-spec-1*
X-DocURI-1: /fcfs_data/*object-spec-2*
.
.
.
X-DocURI-*n-1*: /fcfs_data/*object-spec-n*

n is the number of objects with the specified hash value.

Example

Here's an example of response headers for multiple matching objects:

```
HTTP/1.1 300 Multiple Choices
X-ArcServicedBySystem: hcp.example.com
X-DocCount: 2
X-DocURI: /fcfs_data/Corporate/Employees/2193_John_Doe
X-DocURI: /fcfs_data/HR/Benefits/Plans-2012/Medical.pdf
X-RequestId: 71547E802A1CCE9E
X-ArcClusterTime: 1333828702
Content-Length: 0
```

Transmitting data in compressed format

To save bandwidth, you can compress object data or custom metadata in gzip format before sending it to HCP. In the **PUT** request, you tell HCP that data is compressed so that HCP knows to decompress the data before storing it.

Similarly, in a **GET** request, you can tell HCP to return object data or custom metadata in compressed format. In this case, you need to decompress the returned data yourself.

HCP supports only the gzip algorithm for compressed data transmission.

You tell HCP that the request body is compressed by including a Content-Encoding header with the value gzip in the HTTP **PUT** request. In this case, HCP uses the gzip algorithm to decompress the received data.

You tell HCP to send a compressed response by specifying an Accept-Encoding header in the HTTP **GET** request. If the header specifies gzip, a list of compression algorithms that includes gzip, or *****, HCP uses the gzip algorithm to compress the data before sending it.

For examples of sending and receiving objects in compressed format, see "Example 2: Sending object data in compressed format (Unix)" on page 82 and "Example 4: Retrieving object data in compressed format (command line)" on page 98.

Note: HCP normally compresses object data and custom metadata that it stores, so you do not need to explicitly compress objects for storage. However, if you do need to store gzip-compressed objects or custom metadata, do not use a Content-Encoding header. To retrieve stored gzip-compressed data, do not use an Accept-Encoding header.

Browsing the namespace with HTTP

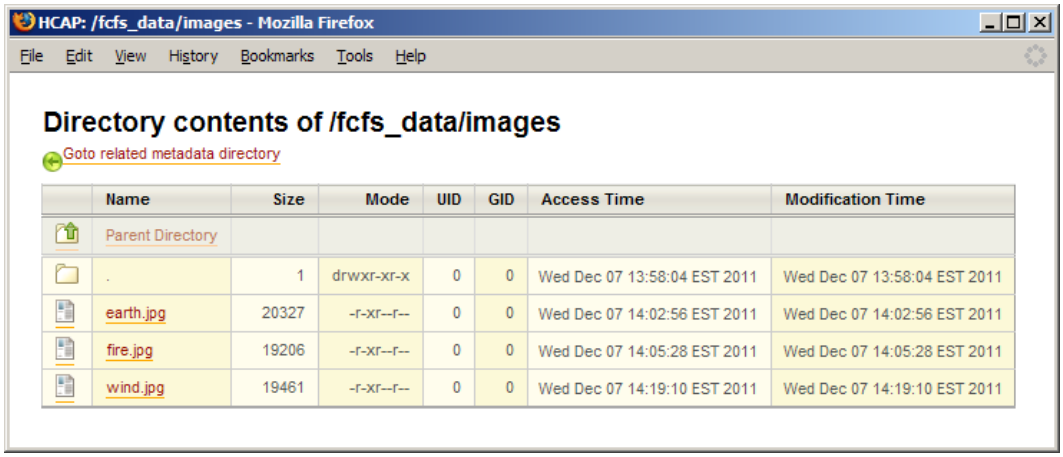
To view the namespace content in a web browser with HTTP, enter an HTTP access namespace URL in the browser address field:

- If you enter the URL for the entire namespace, the browser lists the two root directories, `fcfs_data` and `fcfs_metadata`.
- If you enter the URL for a directory or metadirectory, the browser lists the contents of that directory.

Note: Some browsers may not be able to successfully render pages for directories that contain a very large number of objects, directories, or symbolic links.

- If you enter the URL for an object, the browser downloads the object data and either opens it in the default application for the content type or prompts to open or save it.
- If you enter the URL for a metafile, the browser downloads and displays the contents of that metafile.

For the first two cases, HCP provides an XML stylesheet that determines the appearance of the browser display. The sample browser window below shows what this looks like for the `images` directory.



Tip: You can use the view-source option in the web browser to see the XML that HCP returns.

Working with objects

You can use HTTP to perform these operations on objects:

- Add an object (with or without custom metadata) to the namespace
- Check whether an object exists
- Retrieve all or part of an object
- Delete an object

You can also manage the metadata and custom metadata for an object. For more information on this, see "Working with system metadata" on page 113 and "Working with custom metadata" on page 131.

Storing an object and, optionally, custom metadata

You use the HTTP **PUT** method to store an object in the namespace. Optionally, you can use the same request to store custom metadata for the object.

By default, when you store an object, it inherits several metadata values from its parent directory. You can override some of this metadata when you store the object. For more information on this, see "Specifying metadata on object creation" on page 114.

Request contents — storing object data only

The **PUT** request must include these HTTP elements:

- A URL specifying the location in which to store the object
- A body containing the data to be stored in the namespace

Request contents — sending data in compressed format

You can send object data in compressed format and have HCP decompress it before storing it. To do this, in addition to specifying the request elements listed above:

- Use gzip to compress the content before sending it.
- Include a Content-Encoding request header with a value of gzip.
- Use a chunked transfer encoding.

Request contents — storing object data and custom metadata together

If you're storing object data and custom metadata in a single operation, the **PUT** request must specify these HTTP elements:

- An X-ArcSize header specifying the size, in bytes, of the object data
- A URL specifying the location in which to store the object
- A **type** URL query parameter with a value of **whole-object**
- A body containing the fixed-content data to be stored, followed by the custom metadata, with no delimiter between them

When you store object data and custom metadata in a single operation, the object data must always precede the custom metadata. This differs from the behavior when you retrieve the object data together with the custom metadata, where you can tell HCP to return the results in either order.

You can send the body containing the object data and custom metadata in gzip-compressed format and have HCP decompress both parts before storing them. To do this, follow the instructions in "Request contents — sending data in compressed format" on page 77.

Request-specific return codes

The table below describes the HTTP return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
201	Created	HCP successfully stored the object. If necessary, HCP created new directories in the object path.

400	Bad Request	One of:
		• The URL in the request is not well-formed.
		• The request has a Content-Encoding header that specifies gzip, but the data is not in gzip-compressed format.
		• The request has a **type=whole-object** query parameter, and either:
		  ○ The request does not have an X-ArcSize header.
		  ○ The X-ArcSize header value is greater than the content length.
		• The namespace has custom metadata XML checking enabled, and the request includes custom metadata that is not well-formed XML.
		If the request that causes this error contains both object data and custom metadata, HCP creates an empty object before it returns the error. To resolve this issue, you can either:
		  ○ Fix the custom metadata and retry the request.
		  ○ Add the object again without any custom metadata, thereby replacing the empty object. You can then fix the custom metadata at a later time and add it in a separate request.
		• The request contains an unsupported query parameter or an invalid value for a query parameter.
		If more information about the error is available, the HTTP response headers include the HCP-specific X-ArcErrorMessage header.

(Continued)

Code	Meaning	Description
403	Forbidden	One of: • The namespace does not exist. • The access method (HTTP or HTTPS) is disabled. • You do not have permission to write to the target directory. If more information about the error is available, the HTTP response headers include the HCP-specific X-ArcErrorMessage header.
409	Conflict	HCP could not add the object to the namespace because the object already exists.
413	File Too Large	One of: • Not enough space is available to store the object. Try the request again after objects are deleted from the namespace or the system storage capacity is increased. • The request is trying to store an object that is larger than two TB. HCP cannot store objects larger than two TB. • The request is trying to store custom metadata that is larger than one GB. HCP cannot store custom metadata larger than one GB.
415	Unsupported Media Type	The request has a Content-Encoding header with a value other than gzip.

Request-specific response headers

The table below describes the request-specific response headers returned by a successful request. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Header	Description
X-ArcCustomMetadata Hash	*Returned only if the request contains both data and custom metadata.* The cryptographic hash algorithm HCP uses and the cryptographic hash value of the stored custom metadata, in this format: X-ArcCustomMetadataHash: *hash-algorithm hash-value* You can use the returned hash value to verify that the stored custom metadata is the same as the metadata you sent. To do so, compare this value with a hash value that you generate from the original custom metadata.
X-ArcHash	The cryptographic hash algorithm HCP uses and the cryptographic hash value of the stored object, in this format: X-ArcHash: *hash-algorithm hash-value* You can use the returned hash value to verify that the stored data is the same as the data you sent. To do so, compare this value with a hash value that you generate from the original data.

Example 1: Storing an object

Here's a sample HTTP **PUT** request that stores an object named `wind.jpg` in the `images` directory.

Request with curl command line

```
curl -iT wind.jpg "http://default.default.hcp.example.com/fcfs_data/images/
    wind.jpg"
```

Request in Python using PycURL

```
import pycurl
import os
filehandle = open("wind.jpg", 'rb')
curl = pycurl.Curl()
```

```
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images/wind.jpg")
curl.setopt(pycurl.UPLOAD, 1)
curl.setopt(pycurl.INFILESIZE, os.path.getsize("wind.jpg"))
curl.setopt(pycurl.READFUNCTION, filehandle.read)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
filehandle.close()
```

Request headers

```
PUT /fcfs_data/images/wind.jpg HTTP/1.1
Host: default.default.hcp.example.com
Content-Length: 19461
```

Response headers

```
HTTP/1.1 201 Created
X-ArcServicedBySystem: hcp.example.com
Location: /fcfs_data/images/wind.jpg
X-ArcHash: SHA-256 E6803D3096172298880D60A270940EF4BB2FA2E146CC01BFB...
X-ArcClusterTime: 1333828702
Content-Length: 0
```

Example 2: Sending object data in compressed format (Unix)

Here's a Unix command line that uses the gzip utility to compress the `wind.jpg` file and then pipes the compressed output to a **curl** command. The **curl** command makes an HTTP **PUT** request that sends the data and tells HCP that the data is compressed.

Request with gzip and curl commands

```
gzip -c wind.jpg |
    curl -iT - "https://default.default.hcp.example.com/fcfs_data/images/wind.jpg"
    -H "Content-Encoding: gzip"
```

Request headers

```
PUT /fcfs_data/images/wind.jpg HTTP/1.1
Host: /default.default.hcp.example.com
Content-Length: 124863
Transfer-Encoding: chunked
Content-Encoding: gzip
Expect: 100-continue
```

Response headers

```
HTTP/1.1 100 Continue
HTTP/1.1 201 Created
X-ArcServicedBySystem: hcp.example.com
Location: /fcfs_data/images/wind.jpg
X-ArcHash: SHA-256 E830B86212A66A792A79D58BB185EE63A4FADA76BB8A1...
X-ArcClusterTime: 1333828702
Content-Length: 0
```

Example 3: Sending object data in compressed format (Java®)

Here's the partial implementation of a Java class named HTTPCompression. The implementation shows the WriteToHCP method, which stores an object in the default namespace. The method compresses the data before sending it and uses the Content-Encoding header to tell HCP that the data is compressed.

The WriteToHCP method uses the GZIPCompressedInputStream helper class. For an implementation of this class, see "GZIPCompressedInputStream class" on page 214.

```
import org.apache.http.client.methods.HttpPut;
import org.apache.http.HttpResponse;
import org.apache.http.util.EntityUtils;
import com.hds.hcp.examples.GZIPCompressedInputStream;

class HTTPCompression {
  .
  .
  .
void WriteToHCP() throws Exception {

  /*
   * Set up the PUT request.
   *
   * This method assumes that the HTTP client has already been
   * initialized.
   */
  HttpPut httpRequest = new HttpPut(sHCPFilePath);

  // Indicate that the content encoding is gzip.
  httpRequest.setHeader("Content-Encoding", "gzip");

  // Open an input stream to the file that will be sent to HCP.
  // This file will be processed by the GZIPCompressedInputStream to
  // produce gzip-compressed content when read by the Apache HTTP client.
  GZIPCompressedInputStream compressedInputFile
    = new GZIPCompressedInputStream(new FileInputStream(
                                    sBaseFileName + ".toHCP"));
```

```
  // Point the HttpRequest to the input stream.
  httpRequest.setEntity(new InputStreamEntity(compressedInputFile, -1));

  /*
   * Now execute the PUT request.
   */
  HttpResponse httpResponse = mHttpClient.execute(httpRequest);

  /*
   * Process the HTTP response.
   */
  // If the return code is anything but in the 200 range indicating
  // success, throw an exception.
  if (2 != (int)(httpResponse.getStatusLine().getStatusCode() / 100))
  {
    // Clean up after ourselves and release the HTTP connection to the
    // connection manager.
    EntityUtils.consume(httpResponse.getEntity());

    throw new Exception("Unexpected HTTP status code: " +
      httpResponse.getStatusLine().getStatusCode() + " (" +
      httpResponse.getStatusLine().getReasonPhrase() + ")");
  }

  // Clean up after ourselves and release the HTTP connection to the
  // connection manager.
  Entity Utils.consume(httpResponse.getEntity());
}
  .
  .
  .
}
```

Example 4: Storing object data with custom metadata (Unix)

Here's a Unix command line that uses an HTTP **PUT** request to store the object data and custom metadata for a file named `wind.jpg`. The request stores the object in the `images` directory.

The **cat** command appends the contents of the `wind-custom-metadata.xml` file to the contents of the `wind.jpg` file. The result is piped to a **curl** command that sends the data to HCP.

Unix command line

```
cat wind.jpg wind-custom-metadata.xml | curl -iT -
    -H "X-ArcSize: `stat -c %s wind.jpg`"
    "https://default.default.hcp.example.com/fcfs_data/images/wind.jpg?
    type=whole-object"
```

Request headers

```
PUT /fcfs_data/images/wind2.jpg HTTP/1.1
Host: /default.default.hcp.example.com
X-ArcSize: 237423
Content-Length: 238985
```

Response headers

```
HTTP/1.1 201 Created
X-ArcServicedBySystem: hcp.example.com
Location: /fcfs_data/images/wind2.jpg
X-ArcHash: SHA-256 E830B86212A66A792A79D58BB185EE63A4FADA76BB8A1...
X-ArcCustomMetadataHash: SHA-256 86212A6692A79D5B185EE63A4DA76BBC...
X-ArcTime: 1323449152
Content-Length: 0
```

Example 5: Storing object data with custom metadata (Java)

Here's the partial implementation of a Java class named WholeIO. The implementation shows the WholeWriteToHCP method, which uses a single HTTP **PUT** request to store data and custom metadata for an object.

The WholeWriteToHCP method uses the WholeIOInputStream helper class. For an implementation of this class, see "WholeIOInputStream class" on page 220.

```
import org.apache.http.client.methods.HttpPut;
import org.apache.http.HttpResponse;
import org.apache.http.util.EntityUtils;
import com.hds.hcp.examples.WholeIOInputStream;

class WholeIO {
  .
  .
  .
void WholeWriteToHCP() throws Exception {

  /*
   * Set up the PUT request to store both object data and custom
   * metadata.
   *
   * This method assumes that the HTTP client has already been
   * initialized.
   */
  HttpPut httpRequest = new HttpPut(sHCPFilePath +
    "?type=whole-object");

  FileInputStream dataFile = new FileInputStream(sBaseFileName);
```

```
  // Put the size of the object data into the X-ArcSize header.
  httpRequest.setHeader("X-ArcSize",
    String.valueOf(dataFile.available()));

  // Point the HttpRequest to the input stream with the object data
  // followed by the custom metadata.
  httpRequest.setEntity(
    new InputStreamEntity(
      new WholeIOInputStream(
        new FileInputStream(sBaseFileName),
        new FileInputStream(sBaseFileName + ".cm")),
      -1));

  /*
   * Now execute the PUT request.
   */
  HttpResponse httpResponse = mHttpClient.execute(httpRequest);

  // If the return code is anything but in the 200 range indicating
  // success, throw an exception.
  if (2 != (int)(httpResponse.getStatusLine().getStatusCode() / 100))
  {
    // Clean up after ourselves and release the HTTP connection to the
    // connection manager.
    EntityUtils.consume(httpResponse.getEntity());

    throw new Exception("Unexpected HTTP status code: " +
      httpResponse.getStatusLine().getStatusCode() + " (" +
      httpResponse.getStatusLine().getReasonPhrase() + ")");
  }

  // Clean up after ourselves and release the HTTP connection to the
  // connection manager.
  EntityUtils.consume(httpResponse.getEntity());
}
 .
 .
 .
}
```

Checking the existence of an object

You use the HTTP **HEAD** method to check whether an object exists in the namespace. A 200 (OK) return code indicates that the requested object exists. A 404 (Not Found) return code indicates that the specified URL does not identify an existing object.

Using **HEAD** with a symbolic link checks the existence of the object that's the target of the link.

You can also use the **HEAD** method to retrieve POSIX metadata for an object without retrieving the object data. The POSIX metadata is returned in HTTP response headers.

Request contents

The **HEAD** request must specify the object URL in one of these formats:

- The object path
- If the object is indexed by the search facility selected for use with the Search Console, the cryptographic hash value for the object

  For information on using a hash value to identify an object, see "Access with a cryptographic hash value" on page 73.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
200	OK	HCP found the object.
300	Multiple Choices	For a request by cryptographic hash value, HCP found two or more objects with the specified hash value.
404	Not Found	HCP could not find the specified object. For a request by a cryptographic hash value, this return code can indicate that the object has not been indexed by the search facility selected for use with the Search Console. If the HDDS search facility is selected for use with the Search Console and the request specifies a cryptographic hash value, this return code can indicate that the value was found in HDDS but the object could not be retrieved from HCP.
503	Service Unavailable	For a request by cryptographic hash value, one of: • The cryptographic hash algorithm specified in the request is not the one that the namespace is using. • HCP cannot process the hash value because no search facility is selected for use with the Search Console. • The request URL specifies a namespace other than the default namespace.

Request-specific response headers

The table below describes request-specific response headers returned if HCP finds the object specified by the request. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Header	Description
X-ArcPermissionsUidGid	The POSIX permissions (mode), owner ID, and group ID for the object, in this format: X-ArcPermissionsUidGid: mode=*posix-mode*; uid=*uid*; gid=*gid*
X-ArcSize	The size of the object, in bytes.
X-ArcTimes	The POSIX **ctime**, **mtime**, and **atime** values for the object, in this format: X-ArcTimes: ctime=*ctime*; mtime=*mtime*; atime=*atime*

Example: Checking the existence of an object

Here's a sample HTTP **HEAD** request that checks the existence of an object named `wind.jpg` in the `images` directory.

Request with curl command line

```
curl -I "http://default.default.hcp.example.com/fcfs_data/images/wind.jpg"
```

Request in Python using PycURL

```
import pycurl
import StringIO
cin = StringIO.StringIO()
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images/wind.jpg")
curl.setopt(pycurl.HEADER, 1)
curl.setopt(pycurl.NOBODY, 1)
curl.setopt(pycurl.WRITEFUNCTION, cin.write)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
print cin.getvalue()
curl.close()
```

Request headers

```
HEAD /fcfs_data/images/wind.jpg HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcClusterTime: 1333828702
Content-Type: image/jpeg
Content-Length: 19461
X-ArcPermissionsUidGid: mode=0100775; uid=32; gid=86
X-ArcServicedBySystem: hcp.example.com
X-ArcTimes: ctime=1323449152; mtime=1323449152; atime=1323449152
X-ArcSize: 28463
```

Retrieving an object and, optionally, custom metadata

You use the HTTP **GET** method to retrieve an object from the namespace. When you retrieve an object, you can:

- Tell HCP to return the data in gzip-compressed format
- Get all or part of the object data
- Use a single request to retrieve the object data and custom metadata together

You cannot retrieve part of the object data together with the custom metadata in a single request.

Using **GET** with a symbolic link returns the object that's the target of the link.

To request only a part of the object data, you specify the range of bytes you want in the HTTP **GET** request URL. By specifying a byte range, you can limit the amount of data returned, even when you don't know the size of the object.

Request contents

The **GET** request must specify the object URL in one of these formats:

- The object path
- If the object is indexed by the search facility selected for use with the Search Console, the cryptographic hash value for the object

  For information on using a hash value to identify an object, see "Access with a cryptographic hash value" on page 73.

Request contents — requesting data in compressed format

To request that HCP return the object data in gzip-compressed format, use an Accept-Encoding header containing the value gzip or *. The header can specify additional compression algorithms, but HCP uses only gzip.

Request contents — choosing not to wait for delayed retrievals

HCP may detect that a **GET** request will take a significant amount of time to return an object. You can choose to have the request fail in this situation instead of waiting for HCP to return the object. To do this, in addition to specifying the request elements listed in "Request contents" above, use the **nowait** query parameter.

When a **GET** request fails because the request would take a significant amount of time to return an object and the **nowait** parameter is specified, HCP returns an HTTP 503 (Service Unavailable) error code.

Tip: If the request specifies **nowait** and HCP returns an HTTP 503 error code, retry the request a few times, waiting about thirty seconds in between retries.

Request contents — retrieving object data and custom metadata together

To retrieve object data and custom metadata with a single request, in addition to the elements described above, specify these elements:

- A **type** URL query parameter with a value of **whole-object**
- Optionally, an X-ArcCustomMetadataFirst header specifying the order of the parts, as follows:
  - **true** — The custom metadata should precede the object data.
  - **false** — The object data should precede the custom metadata.

  The default is **false**.

You can also retrieve the object data and custom metadata in gzip-compressed format by specifying an Accept-Encoding header containing the value gzip or *. The header can specify additional compression algorithms, but HCP uses only gzip.

Request contents — requesting a partial object

To retrieve only part of the object data, in addition to the elements described in "Request contents" and, optionally, "Request contents — requesting data in compressed format", specify an HTTP Range request header with the range of bytes to retrieve. Bytes are counted in the object data only. The first byte of the data is in position 0 (zero), so a range of 1-5 specifies the second through sixth bytes of the object, not the first through fifth.

The Range header has this format:

Range: bytes=*range*

These rules apply to the Range header:

- If you omit the Range header, HCP returns the complete object data.
- If you specify a valid range, HCP returns the requested amount of data with a status code of 206.
- If you specify an invalid range, HCP ignores it and returns the complete object data with a status code of 416.
- You cannot request partial object data and custom metadata in the same request. If the request includes a Range header and a **type=whole-object** query parameter, the request fails, and HCP returns a status code of 400.

The table below shows the ways in which you can specify a byte range.

Range Specification	Description	Example
start-position–*end-position*	Bytes in *start-position* through *end-position*, inclusive. If *end-position* is greater than the size of the data, HCP returns the bytes from *start-position* through the end of the data.	Five hundred bytes beginning with the two-hundred-first: 200-699
start-position–	Bytes in *start-position* through the end of the object data.	All the bytes beginning with the seventy-sixth and continuing through the end of the object: 75-

(Continued)

Range Specification	Description	Example
–offset-from-end	Bytes in the *offset-from-end* position, counted back from the last position in the object data, through the end of the object data.	The last 25 bytes of the object: -25

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
200	OK	HCP successfully retrieved the object. This code is also returned if the URL specified a valid directory path and HCP returned a directory listing.
206	Partial content	HCP successfully retrieved the data in the byte range specified in the request.
300	Multiple Choice	For a request by cryptographic hash value, HCP found two or more objects with the specified hash value.
400	Bad Request	The request was not valid. These are some, but not all, of the possible reasons: • The request has both a **type=whole-object** query parameter and a Range request header. • The URL in the request is not well-formed. • The request contains an unsupported query parameter or an invalid value for a query parameter. If more information about the error is available, the HTTP response headers include the HCP-specific X-ArcErrorMessage header.

(Continued)

Code	Meaning	Description
404	Not Found	HCP could not find the specified object. For a request by cryptographic hash value, this return code can indicate that the object has not been indexed by the search facility selected for use with the Search Console. If the HDDS search facility is selected for use with the Search Console and the request specifies a cryptographic hash value, this return code can indicate that the value was found in HDDS but the object could not be retrieved from HCP.
406	Not Acceptable	The request has an Accept-Encoding header that does not include gzip or specify *.
416	Requested range not satisfiable	One of: • The specified start position is greater than the size of the requested data. • The size of the specified range is 0 (zero).
503	Service Unavailable	One of: • For a request by cryptographic hash value, the cryptographic hash algorithm specified in the request is not the one the namespace is using. • For a request by cryptographic hash value, HCP cannot process the hash value because no search facility is selected for use with the Search Console. • For a request by cryptographic hash value, the request URL specifies a namespace other than the default namespace. • The request specifies the **nowait** query parameter, and HCP determined that the request would have taken a significant amount of time to return the object.

Request-specific response headers

The table below describes request-specific response headers returned by a successful request. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Header	Description
Content-Encoding	*Returned only if HCP compressed the response before returning it.* Always gzip.
Content-Length	The length, in bytes, of the returned data. This header has these characteristics: • If you requested that the response be compressed, this is the compressed size of the returned data. • If you requested uncompressed object data without custom metadata, the value of this header is the same as the value of the X-ArcSize header. • If you requested uncompressed partial content, the value is the size of the returned part. This value is equal to the difference between the *start-position* and *end-position* values in the Content-Range header plus one byte. • If you requested uncompressed object data and custom metadata, the value is the sum of the size of the object data (the X-ArcSize header) and the size of the custom metadata. If the returned data is large, HCP may send a chunked response, which does not include this header.
Content-Range	*Returned only when getting partial content.* The byte range of the returned object data, in this format: *start-position–end-position / object-size* *object-size* is the total size of the object data and is the same as the value of the X-ArcSize header.

(Continued)

Header	Description
Content-Type	The type of content: • If you requested all or part of the object data only, this is the Internet media type of the object data, such as text/plain or image/jpg. • If you requested the object data and custom metadata together, this value is always application/octet-stream.
X-ArcContentLength	*Returned only if HCP compressed the response before returning it.* The uncompressed length of the returned data. If the returned data includes both the object data and custom metadata, this is the length of both together.
X-ArcCustomMetadata ContentType	*Returned only if the request asked for the object data and custom metadata.* Always text/xml.
X-ArcCustomMetadata First	*Returned only if the request asked for the object data and custom metadata.* One of: • **true** — The custom metadata precedes the object data. • **false** — The object data precedes the custom metadata.
X-ArcDataContentType	*Returned only if the request asked for the object data and custom metadata.* The Internet media type of the object, such as text/plain or image/jpg.
X-ArcPermissionsUidGid	The POSIX permissions (mode), owner ID, and group ID for the object, in this format: X-ArcPermissionsUidGid: mode=*posix-mode*; uid=*uid*; gid=*gid*
X-ArcSize	The size of the object, in bytes.
X-ArcTimes	The POSIX **ctime**, **mtime**, and **atime** values for the object, in this format: X-ArcTimes: ctime=*ctime*; mtime=*mtime*; atime=*atime*

Response body

The body of the HTTP response contains the requested object data or object data and custom metadata.

Example 1: Retrieving an object by name

Here's a sample HTTP **GET** request that retrieves the object named `wind.jpg` and stores it using the same name on the client system.

Request with curl command line

```
curl  "http://default.default.hcp.example.com/fcfs_data/images/wind.jpg"  >
    wind.jpg
```

Tip: If a **GET** request unexpectedly returns a zero-length file, use the **`-i`** parameter with **curl** to return the response headers in the target file. These headers may provide helpful information for diagnosing the problem.

Request in Python using PycURL

```
import pycurl
filehandle = open("wind2.jpg", 'wb')
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images/wind.jpg")
curl.setopt(pycurl.WRITEFUNCTION, filehandle.write)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
filehandle.close()
```

Request headers

```
GET /fcfs_data/images/wind.jpg HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcClusterTime: 1333828702
Content-Length: 19461
Content-Type: image/jpeg
X-ArcPermissionsUidGid: mode=0100775; uid=10; gid=43
X-ArcServicedBySystem: hcp.example.com
X-ArcTimes: ctime=1323449152; mtime=1323449152; atime=1323449152
X-ArcSize: 19461
```

Example 2: Retrieving an object by its cryptographic hash value

Here's a sample HTTP **GET** request that retrieves an object by its cryptographic hash value (an option available only when a search facility is selected for use with the Search Console) and stores it as `earth.jpg` on the client system.

Request with curl command line

```
curl "http://default.default.hcp.example.com/SHA-256/E3B0C44298FC1C149AF..." >
    earth.jpg
```

Request in Python using PycURL

```
import pycurl
filehandle = open("earth2.jpg", 'wb')
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  SHA-256/E3B0C44298FC1C149AFBF4C8...")
curl.setopt(pycurl.WRITEFUNCTION, filehandle.write)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
filehandle.close()
```

Request headers

```
GET /SHA-256/E3B0C44298FC1C149AFBF4C8E6803D3096172298880D60... HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcClusterTime: 1333828702
Content-Length: 20327
Content-Type: image/jpeg
X-ArcPermissionsUidGid: mode=0100764; uid=10; gid=43
X-ArcServicedBySystem: hcp.example.com
X-ArcTimes: ctime=1323449152; mtime=1323449152; atime=1323449152
X-ArcSize: 20327
```

Example 3: Retrieving part of an object

Here's a sample HTTP **GET** request that retrieves the first 500 bytes of an object named `Recruiters.txt` and stores the returned data as `RecruitersTop.txt` on the client system.

Request with curl command line

```
curl -i -r 0-499 "http://default.default.hcp.example.com/fcfs_data/HR/
    Recruiters.txt" > RecruitersTop.txt
```

Request in Python using PycURL

```
import pycurl
filehandle = open("RecruitersTop.txt", 'wb')
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/HR/Recruiters.txt")
curl.setopt(pycurl.WRITEFUNCTION, filehandle.write)
curl.setopt(pycurl.RANGE, "0-499")
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
filehandle.close()
```

Request headers

```
GET /fcfs_data/HR/Recruiters.txt HTTP/1.1
Range: bytes=0-499
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 206 Partial Content
X-ArcClusterTime: 1333828702
Content-Type: text/plain
Content-Range: bytes 0-499/238985
Content-Length: 500
X-ArcPermissionsUidGid: mode=0100740; uid=45; gid=76
X-ArcServicedBySystem: hcp.example.com
X-ArcTimes: ctime=1323449152; mtime=1323449152; atime=1323449152
X-ArcSize: 238985
```

Example 4: Retrieving object data in compressed format (command line)

Here's a sample **curl** command that tells HCP to compress the `wind.jpg` object before sending it to the client and then decompresses the returned content.

Request with curl command line

```
curl --compressed
    "http://default.default.hcp.example.com/fcfs_data/images/wind.jpg"  >  wind.jpg
```

Request in Python using PycURL

```
import pycurl
filehandle = open("wind.jpg", 'wb')
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "https://default.default.hcp.example.com \
  /fcfs_data/images/wind.jpg")
```

```
curl.setopt(pycurl.ENCODING, 'gzip')
curl.setopt(pycurl.WRITEFUNCTION, filehandle.write)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
filehandle.close()
```

Request headers

```
GET /fcfs_data/images/wind.jpg HTTP/1.1
Host: default.default.hcp.example.com
Accept-Encoding: deflate, gzip
```

Response headers

```
HTTP/1.1 200 OK
X-ArcClusterTime: 1333828702
Content-Encoding: gzip
Content-Length: 93452
Content-Type: image/jpeg
X-ArcContent-Length: 129461
X-ArcPermissionsUidGid: mode=0100775; uid=10; gid=43
X-ArcServicedBySystem: hcp.example.com
X-ArcTimes: ctime=1323449152; mtime=1323449152; atime=1323449152
X-ArcSize: 129461
```

Response body

The contents of the wind.jpg object in gzip-compressed format.

Example 5: Retrieving object data in compressed format (Java)

Here's the partial implementation of a Java class named HTTPCompression. The implementation shows the ReadFromHCP method, which retrieves an object from the default namespace. It uses the Accept-Encoding header to tell HCP to compress the object before returning it and then decompresses the results.

```
import org.apache.http.client.methods.HttpGet;
import org.apache.http.HttpResponse;
import org.apache.http.util.EntityUtils;
import java.util.zip.GZIPInputStream;

class HTTPCompression {
  .
  .
  .
void ReadFromHCP() throws Exception {

  /*
   * Set up the GET request.
   *
```

```
     * This method assumes that the HTTP client has already been
     * initialized.
     */
    HttpGet httpRequest = new HttpGet(sHCPFilePath);

    // Indicate that you want HCP to compress the returned data with gzip.
    httpRequest.setHeader("Accept-Encoding", "gzip");

    /*
     * Now execute the GET request.
     */
    HttpResponse httpResponse = mHttpClient.execute(httpRequest);

    /*
     * Process the HTTP response.
     */

    // If the return code is anything but in the 200 range indicating
    // success, throw an exception.
    if (2 != (int)(httpResponse.getStatusLine().getStatusCode() / 100))
    {
      // Clean up after ourselves and release the HTTP connection to the
      // connection manager.
      EntityUtils.consume(httpResponse.getEntity());

      throw new Exception("Unexpected HTTP status code: " +
        httpResponse.getStatusLine().getStatusCode() + " (" +
        httpResponse.getStatusLine().getReasonPhrase() + ")");
    }

    /*
     * Write the decompressed file to disk.
     */
    FileOutputStream outputFile = new FileOutputStream(
      sBaseFileName + ".fromHCP");

    // Build the string that contains the response body for return to the
    // caller.
    GZIPInputStream bodyISR = new
      GZIPInputStream(httpResponse.getEntity().getContent());
    byte partialRead[] = new byte[1024];
    int readSize = 0;
    while (-1 != (readSize = bodyISR.read(partialRead))) {
      outputFile.write(partialRead, 0, readSize);
    }

    // Clean up after ourselves and release the HTTP connection to the
    // connection manager.
    EntityUtils.consume(httpResponse.getEntity());
  }
    .
    .
    .
  }
```

Example 6: Retrieving object data and custom metadata together (Java)

Here's the partial implementation of a Java class named WholeIO. The implementation shows the WholeReadFromHCP method, which retrieves object data and custom metadata in a single data stream, splits the object data from the custom metadata, and stores each in a separate file.

The WholeReadFromHCP method uses the WholeIOOutputStream helper class. For an implementation of this class, see "WholeIOOutputStream class" on page 221.

```
import org.apache.http.client.methods.HttpGet;
import org.apache.http.HttpResponse;
import org.apache.http.util.EntityUtils;
import com.hds.hcp.examples.WholeIOOutputStream;

class WholeIO {
  .
  .
  .
void WholeReadFromHCP() throws Exception {

  /*
   * Set up the GET request, specifying whole-object I/O.
   *
   * This method assumes that the HTTP client has already been
   * initialized.
   */
  HttpGet httpRequest = new HttpGet(sHCPFilePath +
                                    "?type=whole-object");

  // Request the custom metadata before the object data.
  // This can be useful if the application examines the custom metadata
  // to set the context for the data that will follow.
  httpRequest.setHeader("X-ArcCustomMetadataFirst", "true");

  /*
   * Now execute the GET request.
   */
  HttpResponse httpResponse = mHttpClient.execute(httpRequest);

  // If the return code is anything but in the 200 range indicating
  // success, throw an exception.
  if (2 != (int)(httpResponse.getStatusLine().getStatusCode() / 100))
  {
    // Clean up after ourselves and release the HTTP connection to the
    // connection manager.
    EntityUtils.consume(httpResponse.getEntity());

    throw new Exception("Unexpected HTTP status code: " +
      httpResponse.getStatusLine().getStatusCode() + " (" +
      httpResponse.getStatusLine().getReasonPhrase() + ")");
  }
```

```
  /*
   * Determine whether the object data or custom metadata is first.
   */
  Boolean cmFirst = new Boolean(
    httpResponse.getFirstHeader("X-ArcCustomMetadataFirst").getValue());

  /*
   * Determine the size of the first part based on whether the object
   * data or custom metadata is first.
   */

  // Assume object data is first.
  int firstPartSize = Integer.valueOf(
    httpResponse.getFirstHeader("X-ArcSize").getValue());

  // If custom metadata is first, do the math.
  if (cmFirst) {
    // Subtract the data size from the content length returned.
    firstPartSize = Integer.valueOf(
      httpResponse.getFirstHeader("Content-Length").getValue())
      - firstPartSize;
  }

  /*
   * Split and write the files to disk.
   */
  WholeIOOutputStream outputCreator= new WholeIOOutputStream(
    new FileOutputStream(sBaseFileName + ".fromHCP"),
    new FileOutputStream(sBaseFileName + ".fromHCP.cm"),
    cmFirst);

  outputCreator.copy(httpResponse.getEntity().getContent(),
                     firstPartSize);

  outputCreator.close();  // Files should be created.

  // Clean up after ourselves and release the HTTP connection to the
  // connection manager.
  EntityUtils.consume(httpResponse.getEntity());
}
  .
  .
  .
}
```

Deleting an object

You use the HTTP **DELETE** method to delete an object from the namespace.

Note: Using the **DELETE** method with a symbolic link deletes the link, not the target object.

Request contents

The **DELETE** request must specify the object URL in one of these formats:

- The object path
- If the object is indexed by the search facility selected for use with the Search Console, the cryptographic hash value for the object

  For information on using a hash value to identify an object, see "Access with a cryptographic hash value" on page 73.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
200	OK	HCP successfully deleted the object.
300	Multiple Choice	For a request by cryptographic hash value, HCP found two or more objects with the specified hash value.
403	Forbidden	One of: • The namespace does not exist. • The access method (HTTP or HTTPS) is disabled. • The object is under retention. • You do not have permission to perform the requested operation. If more information about the error is available, the HTTP response headers include the HCP-specific X-ArcErrorMessage header.
404	Not Found	HCP could not find the specified object. For a request by cryptographic hash value, this return code can indicate that the object has not been indexed by the search facility selected for use with the Search Console. If the HDDS search facility is selected for use with the Search Console and the request specifies a cryptographic hash value, this return code can indicate that the value was found in HDDS but the object could not be retrieved from HCP.

(Continued)

Code	Meaning	Description
409	Conflict	HCP could not delete the specified object because the object is currently being written to the namespace.
503	Service Unavailable	For a request by cryptographic hash value, one of: • The cryptographic hash algorithm specified in the request is not the one the namespace is using. • HCP cannot process the hash value because no search facility is selected for use with the Search Console. • The request URL specifies a namespace other than the default namespace.

Request-specific response headers

This request does not have any request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Example: Deleting an object

Here's a sample HTTP **DELETE** request that deletes the object named `wind.jpg` from the `images` directory in the default namespace.

Request with curl command line

```
curl -iX DELETE "http://default.default.hcp.example.com/fcfs_data/images/
    wind.jpg"
```

Request in Python using PycURL

```
import pycurl
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images/wind.jpg")
curl.setopt(pycurl.CUSTOMREQUEST, "DELETE")
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
```

Request headers

```
DELETE /fcfs_data/images/wind.jpg HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcServicedBySystem: hcp.example.com
X-ArcClusterTime: 1333828702
Content-Length: 0
```

Working with directories

You can perform these operations on directories

- Create an empty directory
- Check the existence of a directory
- List directory contents
- Delete an empty directory

You can also manage the metadata for a directory. For more information on this, see "Working with system metadata" on page 113.

Creating an empty directory

You use the HCP-specific HTTP **MKDIR** method to create empty directories in the namespace. If any other directories in the path you specify do not already exist, HCP creates them as well.

Request contents

The **MKDIR** request must specify a URL with the path of the new directory.

You can use URL query parameters to override the default POSIX metadata for the directory. For a description of these parameters, see "Specifying metadata on directory creation" on page 119.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
201	Created	HCP successfully created the directory.
409	Conflict	HCP could not create the directory in the namespace because the directory already exists.

Request-specific response headers

The directory creation operation does not have any request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Example: Adding a directory

Here's a sample HTTP **MKDIR** request that creates a directory named `images` under `fcfs_data`.

Request with curl command line

```
curl -iX MKDIR "http://default.default.hcp.example.com/fcfs_data/images"
```

Request in Python using PycURL

```
import pycurl
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images")
curl.setopt(pycurl.CUSTOMREQUEST, "MKDIR")
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
```

Request headers

```
MKDIR /fcfs_data/images HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 201 Created
X-ArcServicedBySystem: hcp.example.com
Location: /fcfs_data/images
Content-Length: 0
```

Checking the existence of a directory

You use the HTTP **HEAD** method to check whether a directory exists in the namespace.

Request contents

The **HEAD** request must specify the URL of the directory.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
200	OK	HCP found the directory.
404	Not Found	HCP could not find the specified directory.

Request-specific response headers

The table below describes the request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Header	Description
X-ArcPermissionsUidGid	The POSIX permissions (mode), owner ID, and group ID for the directory, in this format: X-ArcPermissionsUidGid: mode=*posix-mode*; uid=*uid*; gid=*gid*
X-ArcSize	For directories, always **-1**.
X-ArcTimes	The POSIX **ctime**, **mtime**, and **atime** values for the directory, in this format: X-ArcTimes: ctime=*ctime*; mtime=*mtime*; atime=*atime*

Example: Checking the existence of a directory

Here's a sample HTTP **HEAD** request that checks the existence of the `images` directory.

Request with curl command line

```
curl -iI "http://default.default.hcp.example.com/fcfs_data/images"
```

Request in Python using PycURL

```
import pycurl
import StringIO
cin = StringIO.StringIO()
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images")
curl.setopt(pycurl.HEADER, 1)
curl.setopt(pycurl.NOBODY, 1)
curl.setopt(pycurl.WRITEFUNCTION, cin.write)
```

```
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
print cin.getvalue()
curl.close()
```

Request headers

```
HEAD /fcfs_data/images HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcClusterTime: 1333828702
Content-Type: text/xml
X-ArcPermissionsUidGid: mode=040700; uid=0; gid=0
X-ArcServicedBySystem: hcp.example.com
X-ArcTimes: ctime=1323449152; mtime=1323449152; atime=1323449152
X-ArcSize: -1
Content-Length: 0
```

Listing directory contents

You use the HTTP **GET** method to list the contents of a directory in the namespace.

Request contents

The **GET** request must specify the URL of the directory.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
200	OK	HCP successfully retrieved the directory listing.
404	Not Found	HCP could not find the specified directory.

Request-specific response headers

The table below describes the request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Header	Description
X-ArcObjectType	For directories, always **directory**.
X-ArcPermissionsUidGid	The POSIX permissions (mode), owner ID, and group ID for the directory, in this format: X-ArcPermissionsUidGid: mode=*posix-mode*; uid=*uid*; gid=*gid*
X-ArcSize	For directories, always **-1**.
X-ArcTimes	The POSIX **ctime**, **mtime**, and **atime** values for the directory, in this format: X-ArcTimes: ctime=*ctime*; mtime=*mtime*; atime=*atime*

Response body

The body of the HTTP response consists of XML that lists the contents of the requested directory. It lists only the immediate directory contents, not the contents of any subdirectories.

The XML for the list has this format:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="/static/directory.xsl"?>
<directory xsi:noNamespaceSchemaLocation="/static/directory.xsd"
    path="directory-path"
    parentDir="parent-directory-path"
     xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance">

<!--Entry format -->
    <entry name="object-directory-or-symbolic-link-name"
        utf8Name="object-directory-or-symbolic-link-name"
        fileType="directory|file|symlink"
        mode="posix-access-mode-as-decimal-number"
        modeString="posix-access-mode-as-mask"
        uid="posix-uid"
        gid="posix-gid"
        size="bytes"
        accessTime="seconds-since-1/1/1970"
        accessTimeString="datetime-value"
        modTime="seconds-since-1/1/1970"
        modTimeString="datetime-value"
```

```
/>

</directory>
```

Notes:

- For directories, the value of the size attribute is always **1**.
- For symbolic links, the value of the size attribute is always **0**.
- The entry for the requested directory has a name value of **.** (period).

Example: Listing directory contents

Here's a sample HTTP **GET** request that retrieves the contents of the `images` directory and saves the results in the `imagesdir.xml` file.

Request with curl command line

```
curl -i "http://default.default.hcp.example.com/fcfs_data/images" > imagesdir.xml
```

Request in Python using PycURL

```
import pycurl
filehandle = open("imagesdir.xml", 'wb')
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images")
curl.setopt(pycurl.WRITEFUNCTION, filehandle.write)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
filehandle.close()
```

Request headers

```
GET /fcfs_data/images HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcClusterTime: 1324554500
Content-Type: text/xml
X-ArcPermissionsUidGid: mode=040777; uid=0; gid=0
X-ArcServicedBySystem: hcp.example.com
X-ArcTimes: ctime=1324553000; mtime=1324553000; atime=1324553000
X-ArcSize: -1
Content-Length: 1473
```

Response body

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="/static/directory.xsl"?>
<directory xsi:noNamespaceSchemaLocation="/static/directory.xsd"
        path="/fcfs_data/images"
        parentDir="/fcfs_data"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <entry name="."
        utf8Name="."
        fileType="directory"
        mode="16895"
        modeString="drwxrwxrwx"
        uid="0"
        gid="0"
        size="1"
        accessTime="1321269812"
        accessTimeString="Mon Nov 14 11:23:32 EST 2011"
        modTime="1321269812"
        modTimeString="Mon Nov 14 11:23:32 EST 2011"/>
    <entry name="wind.jpg"
        utf8Name="wind.jpg"
        fileType="file"
        mode="33268"
        modeString="-rwxrw-r--"
        uid="10"
        gid="43"
        size="19461"
        accessTime="1323773156"
        accessTimeString="Tue Dec 13 10:45:56 EST 2011"
        modTime="1323773156"
        modTimeString="Tue Dec 13 10:45:56 EST 2011"/>
    <entry name="fire.jpg"
        utf8Name="fire.jpg"
        fileType="file"
        mode="33268"
        modeString="-rwxrw-r--"
        uid="10"
        gid="43"
        size="19206"
        accessTime="1321350662"
        accessTimeString="Tue Nov 15 9:51:02 EDT 2011"
        modTime="1321350662"
        modTimeString="Tue Nov 15 9:51:02 EDT 2011"/>
    <entry name="earth.jpg"
        utf8Name="earth.jpg"
        fileType="file"
        mode="33268"
        modeString="-rwxrw-r--"
```

```
        uid="10"
        gid="43"
        size="20327"
        accessTime="1321366428"
        accessTimeString="Tue Nov 15 14:13:48 EDT 2011"
        modTime="1321366428"
        modTimeString="Tue Nov 15 14:13:48 EDT 2011"/>
</directory>
```

Deleting a directory

You use the HTTP **DELETE** method to delete an empty directory from the namespace. You cannot delete a directory that contains any objects, subdirectories, or symbolic links.

Request contents

The **DELETE** request must specify the URL of the directory.

Request-specific return codes

The table below describes the HTTP return codes that have specific meanings for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
200	OK	HCP successfully deleted the directory.
403	Forbidden	One of: • The namespace does not exist. • The access method (HTTP or HTTPS) is disabled. • The directory is not empty. • You do not have permission to delete the directory. If more information about the error is available, the HTTP response headers include the HCP-specific X-ArcErrorMessage header.
404	Not Found	HCP could not find the specified directory.
409	Conflict	HCP could not delete the specified directory because the directory is currently being written to the namespace.

Request-specific response headers

This request does not have any request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Example: Deleting a directory

Here's a sample HTTP **DELETE** request that deletes the directory named `obsolete` from the `images` directory in the default namespace.

Request with curl command line

```
curl -iX DELETE "http://default.default.hcp.example.com/fcfs_data/images/
    obsolete"
```

Request in Python using PycURL

```
import pycurl
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images/obsolete")
curl.setopt(pycurl.CUSTOMREQUEST, "DELETE")
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
```

Request headers

```
DELETE /fcfs_data/images/obsolete HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcServicedBySystem: hcp.example.com
X-ArcClusterTime: 1333828702
Content-Length: 0
```

Working with system metadata

You can perform these operations on the system metadata associated with objects and directories:

- Specify metadata when creating an object
- Specify POSIX metadata when creating a directory
- Retrieve the HCP-specific metadata for objects and directories
- Retrieve the POSIX metadata for objects and directories
- Change HCP-specific metadata for an existing object or directory

- Change the POSIX metadata for an existing object or directory

Note: For detailed information on metadata values, see Chapter 3, "Object properties," on page 33.

Specifying metadata on object creation

When you store an object in the namespace, you can override the defaults for some of its metadata. If the **PUT** request to store the object creates any new directories, the metadata you specify also overrides the defaults for those directories.

You can use query parameters to override the default values for these object properties:

- POSIX owner ID and group ID
- POSIX permissions
- POSIX **atime** and **mtime**
- Retention setting
- Shred setting
- Index setting

For more information on object metadata, see Chapter 3, "Object properties," on page 33.

Request contents

The **PUT** request must include these HTTP elements:

- A URL specifying the location in which to store the object.
- A body containing the data to be stored.
- One or more query parameters to set the metadata values. These parameters are case sensitive.

The table below describes the query parameters you can use to override object metadata defaults.

Parameter	Description
uid	The user ID of the object owner. Valid values are integers greater than or equal to zero.
gid	The ID of the owning group for the object. Valid values are integers greater than or equal to zero.
file_permissions	The POSIX permissions for the object, specified as a three-digit octal value. For more information on permission values, see "Octal permission values" on page 38.
directory_permissions	The POSIX permissions for any new directories in the object path, specified as a three-digit octal value.
atime	The POSIX **atime** value for the object, specified as seconds since January 1, 1970, at 00:00:00 UTC.
mtime	The POSIX **mtime** value for the object, specified as seconds since January 1, 1970, at 00:00:00 UTC.
retention	The retention setting for the object, as described in "Changing retention settings" on page 45. You cannot specify **Hold** or **Unhold** as the value for a metadata override.
shred	The shred setting for the object, specified as a numeric value (0 or 1), as described in "Shred setting" on page 57.
index	The index setting for the object, specified as a numeric value (0 or 1), as described in "Index setting" on page 58.

Request-specific return codes

The table below describes the return codes that have specific meaning for This request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
201	Created	HCP successfully stored the object.
400	Bad Request	One of: • The URL in the request is not well-formed. • The request has a Content-Encoding header that specifies gzip, but the data is not in gzip-compressed format. • The request contains an unsupported query parameter or an invalid value for a query parameter. If more information about the error is available, the HTTP response headers include the HCP-specific X-ArcErrorMessage header.
403	Forbidden	One of: • The namespace does not exist. • The access method (HTTP or HTTPS) is disabled. • The namespace is configured not to allow owner, group, and permission overrides on object creation. • You do not have permission to write to the target directory. If more information about the error is available, the HTTP response headers include the HCP-specific X-ArcErrorMessage header.
409	Conflict	HCP could not store the object in the namespace because the object already exists.
413	File Too Large	One of: • Not enough space is available to store the object. Try the request again after objects are deleted from the namespace or the system storage capacity is increased. • The request is trying to store an object that is larger than two TB. HCP cannot store objects larger than two TB.

Request-specific response header

The table below describes the request-specific response header returned by a successful request. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Header	Description
X-ArcHash	The cryptographic hash algorithm HCP uses and the cryptographic hash value of the stored object, in this format: X-ArcHash: *hash-algorithm hash-value* You can use the returned hash value to verify that the stored data is the same as the data you sent. To do so, compare this value with a hash value that you generate from the original data.

For more information on metadata values, see Chapter 3, "Object properties," on page 33.

Example 1: Overriding owner, group, and permissions

Here's a sample HTTP **PUT** request that stores an object named `wind.jpg` in the default namespace, overriding the default owner, group, and permissions for the object in the process.

Request with curl command line

```
curl -iT wind.jpg "http://default.default.hcp.example.com/fcfs_data/images/
    wind.jpg?uid=10&gid=43&file_permissions=764"
```

Request in Python using PycURL

```
import pycurl
filehandle = open("wind.jpg", 'rb')
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images/wind.jpg?uid=10&gid=43&file_permissions=764")
curl.setopt(pycurl.UPLOAD, 1)
curl.setopt(pycurl.INFILESIZE, os.path.getsize("wind.jpg"))
curl.setopt(pycurl.READFUNCTION, filehandle.read)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
filehandle.close()
```

Request headers

```
PUT /fcfs_data/images/wind.jpg?uid=10&gid=43&file_permissions=764 HTTP/1.1
Host: default.default.hcp.example.com
Content-Length: 5421780
```

Response headers

```
HTTP/1.1 201 Created
X-ArcServicedBySystem: hcp.example.com
Location: /fcfs_data/images/wind.jpg
X-ArcHash: SHA-256 E830B86212A66A792A79D58BB185EE63A4FADA76BB8A1C25...
X-ArcClusterTime: 1333828702
Content-Length: 0
```

Example 2: Overriding the default retention setting

Here's a sample HTTP **PUT** request that stores an object named `fire.jpg` in the default namespace and, in the process, overrides the default retention setting for the object by assigning the object to a retention class.

Request with curl command line

```
curl -iT wind.jpg "http://default.default.hcp.example.com/fcfs_data/images/
    fire.jpg?retention=C+Reg-107"
```

Request in Python using PycURL

```
import pycurl
filehandle = open("fire.jpg", 'rb')
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images/wind.jpg?retention=C+Reg-107")
curl.setopt(pycurl.UPLOAD, 1)
curl.setopt(pycurl.INFILESIZE, os.path.getsize("wind.jpg"))
curl.setopt(pycurl.READFUNCTION, filehandle.read)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
filehandle.close()
```

Request headers

```
PUT /fcfs_data/images/wind.jpg?retention=C+Reg-107 HTTP/1.1
Host: default.default.hcp.example.com
Content-Length: 5421780
```

Response headers

```
HTTP/1.1 201 Created
X-ArcServicedBySystem: hcp.example.com
Location: /fcfs_data/images/wind.jpg
X-ArcHash: SHA-256 38CF3DC9001F01588937A53DF0C9D0ADF4C7C7D147B1A107...
X-ArcClusterTime: 1333828702
Content-Length: 0
```

Specifying metadata on directory creation

When you use the HTTP **MKDIR** method to create one or more directories in the default namespace, you can override the defaults for this POSIX metadata:

- Owner ID and group ID
- Permissions
- **atime** and **mtime**

You use query parameters to override the metadata defaults.

Request contents

The **MKDIR** request must specify these HTTP elements:

- A URL with the path of the new directory.
- One or more query parameters to set the metadata values. These parameters are case sensitive.

The table below describes the query parameters you can use to override directory metadata defaults.

Parameter	Description
uid	The user ID of the directory owner for all new directories in the directory path. Valid values are integers greater than or equal to zero.
gid	The ID of the owning group for all new directories in the directory path. Valid values are integers greater than or equal to zero.
directory_permissions	The POSIX permissions for all new directories in the directory path, specified as a three-digit octal value. For more information on permission values, see "Octal permission values" on page 38.

(Continued)

Parameter	Description
atime	The POSIX **atime** value for all new directories in the directory path, specified as seconds since January 1, 1970, at 00:00:00 UTC.
mtime	The POSIX **mtime** value for all new directories in the directory path, specified as seconds since January 1, 1970, at 00:00:00 UTC.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
201	Created	HCP successfully created the directory.
403	Forbidden	One of: • The namespace does not exist. • The access method (HTTP or HTTPS) is disabled. • You do not have permission to create a directory in the specified location. • The request contains an unsupported query parameter or an invalid value for a query parameter.
409	Conflict	HCP could not create the directory in the namespace because the directory already exists.

Request-specific response headers

This request does not have any request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Example: Overriding owner, group, and permissions

Here's a sample HTTP **MKDIR** request that creates a directory named `images` in the default namespace, overriding the default owner, group, and permissions for the directory in the process.

Request with curl command line

```
curl -iX MKDIR "http://default.default.hcp.example.com/fcfs_data/images?uid=10
    &gid=43&directory_permissions=764"
```

Request in Python using PycURL

```
import pycurl
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images?uid=10&gid=43&directory_permissions=764")
curl.setopt(pycurl.CUSTOMREQUEST, "MKDIR")
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
```

Request headers

```
MKDIR /fcfs_data/images?uid=10&gid=43&directory_permissions=764 HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 201 Created
X-ArcServicedBySystem: hcp.example.com
Location: /fcfs_data/images
Content-Length: 0
```

Retrieving HCP-specific metadata

You use the HTTP **GET** method to retrieve HCP-specific metadata for an object or directory. You do this by retrieving the contents of the metafile that contains the information. You can retrieve the contents of any metafile, including `core-metadata.xml`. For detailed information on the metafiles for objects and directories, see "Metafiles" on page 21.

Request contents

The **GET** request must specify the URL of the metafile that contains the metadata you want.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
200	OK	HCP successfully retrieved the metafile.
404	Not Found	HCP could not find the specified metafile.

Request-specific response headers

The table below describes the request-specific response headers returned by a successful request. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Header	Description
X-ArcPermissionsUidGid	The POSIX permissions (mode), owner ID, and group ID for the metafile, in this format: X-ArcPermissionsUidGid: mode=*posix-mode*; uid=*uid*; gid=*gid*
X-ArcSize	The size of the metafile, in bytes.
X-ArcTimes	The POSIX **ctime**, **mtime**, and **atime** values for the metafile, in this format: X-ArcTimes: ctime=*ctime*; mtime=*mtime*; atime=*atime*

Response body

The body of the HTTP response contains the contents of the requested metafile.

Example: Getting the retention setting for an object

Here's a sample HTTP **GET** request that retrieves the contents of the `retention.txt` metafile for the `images/wind.jpg` object.

Request with curl command line

```
curl -i "http://default.default.hcp.example.com/fcfs_metadata/images/wind.jpg/
    retention.txt"
```

Request in Python using PycURL

```
import pycurl
import StringIO
cin = StringIO.StringIO()
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_metadata/images/wind.jpg/retention.txt")
curl.setopt(pycurl.WRITEFUNCTION, cin.write)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
print cin.getvalue()
curl.close()
```

Request headers

```
GET /fcfs_metadata/images/wind.jpg HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcClusterTime: 1333828702
Content-Type: text/plain
Content-Length: 128
X-ArcPermissionsUidGid: mode=0100644; uid=0; gid=0
X-ArcServicedBySystem: hcp.example.com
X-ArcTimes: ctime=1323449971; mtime=1323449971; atime=1323449971
X-ArcSize: 128
```

Response body

```
1922272200
2030-11-30T8:30:00-0400 (Reg-107, A+21y)
```

Retrieving POSIX metadata

To retrieve the POSIX metadata (UID, GID, permissions, **atime**, **mtime**, and **ctime**) for an object or directory, check object or directory existence as described in "Checking the existence of an object" on page 86 and "Checking the existence of a directory" on page 106.

The POSIX metadata is also returned when you retrieve the object or list the contents of the directory.

Modifying HCP-specific metadata

You use the **PUT** method to change this HCP-specific metadata for existing objects and directories:

- Retention setting
- Shred setting
- Index setting

You change these settings by overwriting the applicable metafile, such as `retention.txt`.

Request contents

The **PUT** request must include these HTTP elements:

- The URL of the metafile you are overwriting
- The new metadata value as the request body

The body content depends on the metafile being overwritten. The table below lists the metafiles and the values you can specify.

Metafile	Valid values
index.txt	**0** (zero) or **1** (one) For the metadata query engine, zero means don't index custom metadata. One means index custom metadata. For the HCP search facility, zero means don't index the object. One means index the object.
retention.txt	Any valid retention setting. For more information, see "Changing retention settings" on page 45.
shred.txt	**0** or **1** Zero means don't shred. One means shred.

Request-specific return codes

The table below describes the HTTP return codes that have specific meanings for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
201	Created	HCP successfully stored the metafile.
400	Bad Request	The request is trying to store a metafile for a nonexistent object.

Request-specific response headers

This request does not have any request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Example: Changing the retention setting for an existing object

Here's a sample HTTP **PUT** request that assigns the `images/wind.jpg` file to the Reg-107 retention class.

Request with curl command line

```
echo C+Reg-107 | curl -iT - "http://default.default.hcp.example.com/
    fcfs_metadata/images/wind.jpg/retention.txt"
```

Request in Python using PycURL

```
import pycurl
import StringIO
data = "C+Reg-107"
cin = StringIO.StringIO(data)
curl = pycurl.Curl()
curl.setopt(pycurl.READFUNCTION, cin.read)
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_metadata/images/wind.jpg/retention.txt")
curl.setopt(pycurl.UPLOAD, 1)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
```

Request headers

```
PUT /fcfs_metadata/images/wind.jpg/retention.txt HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 201 Created
X-ArcServicedBySystem: hcp.example.com
X-ArcClusterTime: 1333828702
Content-Length:  0
```

Modifying POSIX metadata

You use these HCP-specific methods to change the POSIX metadata for existing objects and directories:

- **CHMOD** changes the permissions
- **CHOWN** changes the POSIX user ID and group ID
- **TOUCH** changes **atime** and **mtime**

Note: The namespace can be configured to disallow permission, owner, and group changes through the HTTP protocol. It can also be configured to disallow these changes through any protocol for objects that are under retention. To find out how your namespace is configured, see your namespace administrator.

You cannot use the **CHMOD** or **CHOWN** method on a symbolic link. Using **TOUCH** on a symbolic link changes the applicable value for the object that's the target of the link.

You cannot use HTTP **TOUCH** to create new objects the way you can with the Unix **touch** command.

Request contents

The request must include these HTTP elements:

- The method: **CHOWN**, **CHMOD**, or **TOUCH**
- The object URL in one of these formats:
  - The object path
  - If the object is indexed by the search facility selected for use with the Search Console, the cryptographic hash value for the object

    For information on using a hash value to identify an object, see "Access with a cryptographic hash value" on page 73.
- One or more query parameters specifying the new metadata

The table below lists the query parameters for each method.

Method	Parameters	Description
CHMOD	**permissions=***octal-permission-value*	The **permissions** parameter specifies the new POSIX permissions for the object or directory as an octal value. For more information on permission values, see "Octal permission values" on page 38.
CHOWN	Both: **uid=***user-id* **gid=***group-id*	The **uid** parameter specifies the POSIX user ID of the object or directory owner. The **gid** parameter specifies the POSIX group ID of the owning group for the object or directory. Valid values for both parameters are integers greater than or equal to zero. To change only one value, specify the new value for the changed ID and the current value for the unchanged ID.

(Continued)

Method	Parameters	Description
TOUCH	Either or both: **atime=***value* **mtime=***value*	The **atime** parameter specifies the new POSIX atime value for the object or directory. The **mtime** parameter specifies the new POSIX mtime value for the object or directory. Valid values for both parameters are seconds since January 1, 1970, at 00:00:00 UTC or, for the current time, **now**.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
200	OK	HCP successfully changed the object or directory metadata.
300	Multiple Choice	For a request by cryptographic hash value, HCP found two or more objects with the specified hash value.

400	Bad request	One of:
		
		• The URL in the request is not well-formed.
		
		• The request specifies a cryptographic hash value that's not valid for the specified cryptographic hash algorithm.
		
		• The request does not contain a required query parameter:
		
		  ○ For **CHMOD**, the **permissions** parameter is missing.
		
		  ○ For **CHOWN**, one or both of the **uid** and **gid** parameters are missing.
		
		  ○ For **TOUCH**, both of the **atime** and **mtime** parameters are missing.
		
		• The request contains an unsupported query parameter or an invalid value for a query parameter.
		
		If more information about the error is available, the HTTP response headers include the HCP-specific X-ArcErrorMessage header.

(Continued)

Code	Meaning	Description
403	Forbidden	One of: • The namespace does not exist. • The access method (HTTP or HTTPS) is disabled. • You do not have permission to change the specified metadata for the object or directory. • For the **CHOWN** or **CHMOD** method, the URL specifies a symbolic link.
404	Not Found	HCP could not find the specified object or directory. If the request specifies a cryptographic hash value, this return code can indicate that the object has not been indexed by the search facility selected for use with the Search Console. If the HDDS search facility is selected for use with the Search Console and the request specifies a cryptographic hash value, this return code can indicate that the value was found in HDDS but the object could not be retrieved from HCP.
503	Service Unavailable	For a request by cryptographic hash value, one of: • The cryptographic hash algorithm specified in the request is not the one the namespace is using. • HCP cannot process the hash value because no search facility is selected for use with the Search Console. • The request URL specifies a namespace other than the default namespace.

Request-specific response headers

This request does not have any request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Example 1: Changing the permissions for an existing object

Here's a sample HTTP **CHMOD** request that changes the permissions for the object named `wind.jpg` to 755.

Request with curl command line

```
curl -iX CHMOD "http://default.default.hcp.example.com/fcfs_data/images/
    wind.jpg?permissions=755"
```

Request in Python using PycURL

```
import pycurl
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images/wind.jpg?permissions=755")
curl.setopt(pycurl.CUSTOMREQUEST, "CHMOD")
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
```

Request headers

```
CHMOD /fcfs_data/images/wind.jpg?permissions=755 HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcServicedBySystem: hcp.example.com
Content-Length: 0
```

Example 2: Changing the user and group IDs for an existing object

Here's a sample HTTP **CHOWN** request that changes the owner ID and group ID for the object named `wind.jpg` to 22 and 17, respectively.

Request with curl command line

```
curl -iX CHOWN "http://default.default.hcp.example.com/fcfs_data/images/
    wind.jpg?uid=22&gid=17"
```

Request in Python using PycURL

```
import pycurl
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images/wind.jpg?uid=22&gid=17")
curl.setopt(pycurl.CUSTOMREQUEST, "CHOWN")
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
```

Request headers

```
CHOWN /fcfs_data/images/wind.jpg?uid=22&gid=17 HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcServicedBySystem: hcp.example.com
Content-Length: 0
```

Example 3: Changing the atime value for an existing object

Here's a sample HTTP **TOUCH** request that changes the **atime** value for the object named `wind.jpg` to September 9, 2015, at 4:00 p.m. UTC.

Request with curl command line

```
curl -iX TOUCH "http://default.default.hcp.example.com/fcfs_data/images/
    wind.jpg?atime=1441814400"
```

Request in Python using PycURL

```
import pycurl
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_data/images/wind.jpg?atime=1441814400")
curl.setopt(pycurl.CUSTOMREQUEST, "TOUCH")
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
```

Request headers

```
TOUCH /fcfs_data/images/wind.jpg?atime=1441814400 HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcServicedBySystem: hcp.example.com
Content-Length: 0
```

Working with custom metadata

You can perform these operations on custom metadata for an object:

- Store or replace the custom metadata
- Check the existence of custom metadata
- Retrieve the custom metadata
- Delete the custom metadata

Note: For detailed information on custom metadata, including possible limitations on operations on custom metadata for objects under retention, see "Custom metadata" on page 60.

Storing custom metadata

You use an HTTP **PUT** request to store or replace custom metadata for an existing object. The namespace can be configured to require custom metadata to be well-formed XML. In this case, HCP rejects custom metadata that is not well-formed XML and returns a 400 error code.

Custom metadata is stored as a single unit. You can add it or replace it in its entirety, but you cannot add to or change existing custom metadata. If you store custom metadata for an object that already has custom metadata, the new metadata replaces the existing metadata.

In addition to storing custom metadata for an existing object, you can use a single request to store object data and custom metadata at the same time. For more information on this, see "Storing an object and, optionally, custom metadata" on page 77.

Request contents

The **PUT** request must include these HTTP elements:

- A URL specifying the path to the `custom-metadata.xml` file for the object. The file name must be `custom-metadata.xml`, even if the custom metadata is not in XML format.
- A body consisting of the custom metadata.

Request contents — sending data in compressed format

You can send custom metadata in compressed format and have HCP decompress the data before storing it. To do this, in addition to specifying the request elements listed above:

- Use gzip to compress the custom metadata before sending it.
- Include a Content-Encoding request header with a value of gzip.
- Use a chunked transfer encoding.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
201	Created	HCP successfully stored the custom metadata.
400	Bad Request	One of: • The URL in the request is not well-formed. • The namespace has custom metadata XML checking enabled, and the request includes custom metadata that is not well-formed XML. • The request is trying to store custom metadata for a directory or symbolic link. • The request has a Content-Encoding header that specifies gzip, but the custom metadata is not in gzip-compressed format. If more information about the error is available, the HTTP response headers include the HCP-specific X-ArcErrorMessage header.
404	Not Found	HCP could not find the object for which the custom metadata is being stored.
409	Conflict	The object for which the custom metadata is being stored was ingested using CIFS or NFS and the lazy close period for the object has not expired.

(Continued)

Code	Meaning	Description
413	File too Large	One of: • Not enough space is available to store the custom metadata. Try the request again after objects are deleted from the namespace or the system storage capacity is increased. • The request is trying to store custom metadata that is larger than one GB. HCP cannot store custom metadata larger than one GB.
415	Unsupported Media Type	The request has a Content-Encoding header with a value other than gzip.

Request-specific response headers

The table below describes the request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Header	Description
X-ArcHash	The cryptographic hash algorithm HCP uses and the cryptographic hash value of the stored XML, in this format: X-ArcHash: *hash-algorithm hash-value* You can use the returned hash value to verify that the stored data is the same as the data you sent. To do so, compare this value with a hash value that you generate from the original data.

Example: Storing custom metadata for an object

Here's a sample HTTP **PUT** request that stores the custom metadata specified in the `wind.custom-metadata.xml` file for an existing object named `wind.jpg`.

Request with curl command line

```
curl -iT wind.custom-metadata.xml "http://default.default.hcp.example.com/
    fcfs_metadata/images/wind.jpg/custom-metadata.xml"
```

Request in Python using PycURL

```
import pycurl
filehandle = open("wind.custom-metadata.xml", 'rb')
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_metadata/images/wind.jpg/custom-metadata.xml")
curl.setopt(pycurl.UPLOAD, 1)
curl.setopt(pycurl.INFILESIZE,
  os.path.getsize("wind.custom-metadata.xml"))
curl.setopt(pycurl.READFUNCTION, filehandle.read)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
filehandle.close()
```

Request headers

```
PUT /fcfs_metadata/images/wind.jpg/custom-metadata.xml HTTP/1.1
Host: default.default.hcp.example.com
Content-Length: 317
```

Response headers

```
HTTP/1.1 201 Created
X-ArcServicedBySystem: hcp.example.com
X-ArcHash: SHA-256
  20BA1FDC958D8519D11A4CC2D6D65EC64DD12466E456A32DB800D9FC329A02B9
Location: /fcfs_metadata/images/wind.jpg/custom-metadata.xml
X-ArcClusterTime: 1333828702
Content-Length: 0
```

For more information on `custom-metadata.xml` files, see "Custom metadata" on page 60.

Checking the existence of custom metadata

You use the HTTP **HEAD** method to check whether an object has custom metadata.

Request contents

The **HEAD** request must specify the URL of the `custom-metadata.xml` file.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
200	OK	HCP found the custom metadata.
204	No Content	The specified object does not have custom metadata.
404	Not Found	HCP could not find the specified object.

Request-specific response headers

The table below describes the request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Header	Description
X-ArcPermissionsUidGid	The POSIX permissions (mode), owner ID, and group ID for the custom metadata metafile. These values are the same as those for the object. The header has this format: X-ArcPermissionsUidGid: mode=*posix-mode*; uid=*uid*; gid=*gid*
X-ArcSize	The size of the `custom-metadata.xml` file, in bytes.
X-ArcTimes	The POSIX **ctime**, **mtime**, and **atime** values for the `custom-metadata.xml` file. These values are the same as those for the object. The header has this format: X-ArcTimes: ctime=*ctime*; mtime=*mtime*; atime=*atime*

Example: Checking the existence of custom metadata

Here's a sample HTTP **HEAD** request that checks the existence of custom metadata for the `images/wind.jpg` object.

Request with curl command line

```
curl -iI "http://default.default.hcp.example.com/fcfs_metadata/images/wind.jpg/
    custom-metadata.xml"
```

Request in Python using PycURL

```
import pycurl
import StringIO
cin = StringIO.StringIO()
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_metadata/images/wind.jpg/custom-metadata.xml")
curl.setopt(pycurl.HEADER, 1)
curl.setopt(pycurl.NOBODY, 1)
curl.setopt(pycurl.WRITEFUNCTION, cin.write)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
print cin.getvalue()
curl.close()
```

Request headers

```
HEAD /fcfs_data/images/wind.jpg/custom-metadata.xml HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcClusterTime: 1333828702
Content-Type: text/xml
Content-Length: 317
X-ArcPermissionsUidGid: mode=0100644; uid=0; gid=0
X-ArcServicedBySystem: hcp.example.com
X-ArcTimes: ctime=1323449971; mtime=1323449971; atime=1323449971
X-ArcSize: 317
```

Retrieving custom metadata

You use the HTTP **GET** method to retrieve the custom metadata for an object.

You can also use a single request to retrieve object data and custom metadata at the same time. For more information, see "Retrieving an object and, optionally, custom metadata" on page 89.

Request contents

The **GET** request must specify the URL of the `custom-metadata.xml` file.

Request contents – retrieving data in compressed format

To request that HCP return the custom metadata in gzip-compressed format, use an Accept-Encoding header containing the value gzip or *. The header can specify additional compression algorithms but HCP uses only gzip.

Request contents — choosing not to wait for delayed retrievals

HCP may detect that a **GET** request will take a significant amount of time to return custom metadata. You can choose to have the request fail in this situation instead of waiting for HCP to return the custom metadata. To do this, in addition to specifying the request elements listed in "Request contents" above, use the **nowait** query parameter.

When a **GET** request fails because the request would take a significant amount of time to return an object and the **nowait** parameter is specified, HCP returns an HTTP 503 (Service Unavailable) error code.

Tip: If the request specifies **nowait** and HCP returns an HTTP 503 error code, retry the request a few times, waiting about thirty seconds in between retries.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
200	OK	HCP successfully retrieved the custom metadata.
204	No Content	The object does not have custom metadata.
404	Not Found	HCP could not find the specified object.
406	Not Acceptable	The request has an Accept-Encoding header that does not include gzip or specify *.
503	Service Unavailable	The request specifies the **nowait** query parameter, and HCP determined that the request would have taken a significant amount of time to return the custom metadata.

Request-specific response headers

The table below describes request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Header	Description
Content-Encoding	*Returned only if HCP compressed the custom metadata before returning it.* Always gzip.
X-ArcContentLength	*Returned only if HCP compressed the custom metadata before returning it.* The length of the stored custom metadata before compression.
X-ArcPermissionsUidGid	The POSIX permissions (mode), owner ID, and group ID for the `custom-metadata.xml` file. These values are the same as those for the object. The header has this format: X-ArcPermissionsUidGid: mode=*posix-mode*; uid=*uid*; gid=*gid*
X-ArcSize	The size of the `custom-metadata.xml` file, in bytes.
X-ArcTimes	The POSIX **ctime**, **mtime**, and **atime** values for the custom-metadata.xml file. These values are the same as those for the object. The header has this format: X-ArcTimes: ctime=*ctime*; mtime=*mtime*; atime=*atime*

Response body

The body of the HTTP response contains the custom metadata.

Example: Retrieving custom metadata for an object

Here's a sample HTTP **GET** request that retrieves custom metadata for an object named `wind.jpg` in the `images` directory and saves the results in the `wind.custom-metadata.xml` file.

Request with curl command line

```
curl -i "http://default.default.hcp.example.com/fcfs_metadata/images/wind.jpg/
    custom-metadata.xml" > wind.custom-metadata.xml
```

Request in Python using PycURL

```
import pycurl
filehandle = open("wind.custom-metadata.xml", 'wb')
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_metadata/images/wind.jpg/custom-metadata.xml")
curl.setopt(pycurl.WRITEFUNCTION, filehandle.write)
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
filehandle.close()
curl.close()
filehandle = open("custom-metadata.xml", 'rb')
print filehandle.read()
filehandle.close()
```

Request headers

```
GET /fcfs_metadata/images/wind.jpg/custom-metadata.xml HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcClusterTime: 1333828702
Content-Type: text/xml
Content-Length: 317
X-ArcPermissionsUidGid: mode=0100644; uid=0; gid=0
X-ArcTimes: ctime=1323449971; mtime=1323449971; atime=1323449971
X-ArcServicedBySystem: hcp.example.com
X-ArcSize: 317
```

Deleting custom metadata

You use the HTTP **DELETE** method to delete the custom metadata for an object.

Request contents

The **DELETE** request must specify the URL of the `custom-metadata.xml` file.

Request-specific return codes

The table below describes the return codes that have specific meaning for this request. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

Code	Meaning	Description
200	OK	HCP successfully deleted the custom metadata.
204	No Content	The specified object does not have custom metadata.
404	Not Found	HCP could not find the object for which you are trying to delete the custom metadata.
409	Conflict	HCP could not delete the custom metadata because the custom metadata is currently being written to the namespace.

Request-specific response headers

This request does not have any request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Example: Deleting custom metadata

Here's a sample HTTP **DELETE** request that deletes the custom metadata for the object named `earth.jpg`.

Request with curl command line

```
curl -iX DELETE "http://default.default.hcp.example.com/fcfs_metadata/images/
    earth.jpg/custom-metadata.xml"
```

Request in Python using PycURL

```
import pycurl
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com/ \
  fcfs_metadata/images/earth.jpg/custom-metadata.xml")
curl.setopt(pycurl.CUSTOMREQUEST, "DELETE")
curl.perform()
print curl.getinfo(pycurl.RESPONSE_CODE)
curl.close()
```

Request headers

```
DELETE /fcfs_metadata/images/earth.jpg/custom-metadata.xml HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
X-ArcServicedBySystem: hcp.example.com
X-ArcClusterTime: 1333828702
Content-Length: 0
```

Checking the available storage and software version

You use the HTTP **HEAD** method to check the amount of space currently available for storing additional objects in the namespace and the version number of the HCP software.

Request contents

The **HEAD** request must specify the namespace URL.

Request-specific return codes

A valid request results in a 200 (OK) return code. For descriptions of all possible return codes, see "HTTP return codes" on page 204.

An invalid URL typically results in a failure to resolve the specified hostname and does not return an error code.

Request-specific response headers

The table below describes the request-specific response headers. For information on all HCP-specific response headers, see "HCP-specific HTTP response headers" on page 209.

Header	Description
X-ArcAvailableCapacity	The amount of storage space in the default namespace currently available for storing additional objects, in bytes. Storage space is used for object data, metadata, any redundant data required by the DPL, and the metadata query engine index. The header has this format: X-ArcAvailableCapacity: *available-bytes*
X-ArcTotalCapacity	The total amount of storage space in the HCP system, in bytes. This value includes both used and available space. The header has this format: X-ArcTotalCapacity: *total-bytes*
X-ArcSoftwareVersion	The version number of the HCP software.

Note: The values returned in the X-ArcAvailableCapacity and X-ArcTotalCapacity headers can exceed 32-bit integers. You should ensure that any variables used to hold these values can handle the larger numbers.

Example: Checking the available storage

Here's a sample HTTP **HEAD** request that checks the amount of available storage for the default namespace in the system named `hcp.example.com`.

Request with curl command line

```
curl -I "http://default.default.hcp.example.com"
```

Request in Python using PycURL

```
import pycurl
import StringIO
cin = StringIO.StringIO()
curl = pycurl.Curl()
curl.setopt(pycurl.URL, "http://default.default.hcp.example.com")
curl.setopt(pycurl.HEADER, 1)
curl.setopt(pycurl.NOBODY, 1)
curl.setopt(pycurl.WRITEFUNCTION, cin.write)
curl.perform()
```

```
print curl.getinfo(pycurl.RESPONSE_CODE)
print cin.getvalue()
curl.close()
```

Request headers

```
HEAD / HTTP/1.1
Host: default.default.hcp.example.com
```

Response headers

```
HTTP/1.1 200 OK
Content-Type: text/xml
X-ArcClusterTime: 1333828702
X-ArcAvailableCapacity: 552466767872
X-ArcTotalCapacity: 562099757056
X-ArcSoftwareVersion: 6.0.1.64
Content-Length: 1167
```

HTTP usage considerations

The following sections present considerations that affect the use of the HTTP protocol for namespace access. For additional considerations that are not specific to the HTTP protocol, see Chapter 9, "General usage considerations," on page 191.

HTTP permission checking

The namespace configuration specifies the level of permission checking on HTTP requests:

- **No permission checking** — HCP doesn't check permissions for any operations.
- **Permission checking only on the first file or directory** — If you're adding an object to the namespace, HCP checks the permissions only for the target directory. If you're performing an operation on an existing object, HCP checks the permissions only for that object.
- **Strict POSIX permission checking** — For all operations, HCP checks permissions for the target object as well as for all directories in the path to that object.

Strict permission checking enhances the security of stored data but results in slower performance. With no permission checking, performance is unaffected, but the security benefit is lost.

To learn the level of HTTP permission checking in effect for the namespace, see your namespace administrator.

HTTP persistent connections

HCP supports HTTP persistent connections. Following a request for an operation, HCP keeps the connection open for 60 seconds so a subsequent request can use the same connection.

Persistent connections enhance performance because they avoid the overhead of opening and closing multiple connections. In conjunction with persistent connections, using multiple threads so that operations can run concurrently provides still better performance.

If the persistent connection timeout period is too short, tell your namespace administrator.

Note: With persistent connections, if a single node has more than 254 concurrent open connections, those above the first 254 may have to wait as long as ten minutes to be serviced. This includes connections where the request explicitly targeted the node, as well as connections where the HCP DNS name resolved to the target node.

To avoid this issue, either don't use persistent connections or ensure that no more than 254 threads are working against a single node at any given time.

Storing zero-sized files with HTTP

When you store a zero-sized file with HTTP, the resulting object has no data. Because HTTP causes a flush and a close even when no data is present, this object is WORM and is treated like any other object in the namespace.

Using HTTP with objects open for write

These considerations apply to using HTTP to access objects that are open for write:

- If you try to write to the object, HCP returns a 409 (Conflict) error.

- If try to retrieve, delete, or check the existence of an object that's open for write, HCP returns a 404 (Not Found) error code and does not perform the operation.

Note: Depending on the timing, the delete request may result in a busy error. In that case, wait one or two seconds and then try the request again.

Failed HTTP write operations

An HTTP write operation is considered to have failed if either of these is true:

- The target node failed while the object was open for write.
- The TCP connection broke (for example, due to a front-end network failure or the abnormal termination of the client application) while the object was open for write.

Also, in some circumstances, a write operation is considered to have failed if another node or other hardware failed while the object was open for write.

If a write fails, HCP does not create a new object.

Tip: If a write operation fails, retry the request.

HTTP connection failure handling

You should retry an HTTP request if either of these happens:

- You (or an application) cannot establish an HTTP connection to the HCP system.
- The connection breaks while HCP is processing a request. In this case, the most likely cause is that the node processing the request became unavailable.

When retrying the request:

- If the original request used the DNS name of the HCP system in the URL, repeat the request in the same way.

- If the original request used the IP address of a specific node, retry the request using either the IP address of a different node or the DNS name of the system.

If the connection breaks while HCP is processing a **GET** request, you may not know whether the returned data is all or only some of the object data. In this case, you can check the number of returned bytes against the content length returned in the HTTP Content-Length response header. If the numbers match, the returned data is complete.

Data chunking with HTTP write operations

In some cases, the size of the data to be sent to the namespace through HTTP cannot be known at the start of a **PUT** request. For example, the size is unknown if data is dynamically generated and the **PUT** request starts before all data is available. This scenario would occur if you do not have enough memory or disk space to stage dynamically generated data locally, so the application streams the **PUT** request as the data is generated.

In such cases, you can send the data using **chunked** HTTP transfer coding. Each chunk is sent with a known size, except for the last chunk, which is sent with a size of 0 (zero).

If possible, you should avoid chunking data because it increases the overhead required for the **PUT** operation.

Multithreading with HTTP

With HTTP, only one client can write to a given object at one time. A multithreaded client can write to multiple objects at the same time but cannot have multiple threads writing to the same object.

Multiple clients can use HTTP to read the same object concurrently. Similarly, a multithreaded client can use multiple threads to read a single object. However, because the reads can occur out of order, you generally get better performance by using one thread per object.

HCP has a limit of 255 concurrent HTTP connections per node, with another 20 queued.

Note: HTTP and WebDAV share the same connection pool.

5

WebDAV

WebDAV is one of the industry-standard protocols HCP supports for namespace access. To access the namespace through WebDAV, you can write applications that use any standard WebDAV client library, or you can use a command-line tool, such as **cadaver**, that supports WebDAV. You can also use WebDAV to access the default namespace directly from a web browser or other WebDAV client.

Using the WebDAV protocol, you can store, view, retrieve, and delete objects. You can also add and delete custom metadata, as well as change certain system metadata for existing objects.

HCP is compliant with WebDAV level 2, as specified by RFCs 2518 and 4918.

The HCP implementation of the WebDAV protocol is separate from the HCP implementation of the HTTP protocol. Therefore, HCP extensions to HTTP do not apply to WebDAV.

For you to access the default namespace through WebDAV, this protocol must be enabled in the namespace configuration. If you cannot access the namespace in this way, see your namespace administrator.

This chapter explains how to use WebDAV for namespace access.

The examples in this chapter use **cadaver**, which is freely available open-source software. You can download **cadaver** from http://www.webdav.org/cadaver. The examples use a version of **cadaver** that was available at the time this book was written.

Note: This chapter uses the object and directory terminology that's used elsewhere in this book. An object is equivalent to a WebDAV resource. A directory is equivalent to a WebDAV collection.

WebDAV methods

HCP supports most standard WebDAV methods, as indicated in the table below.

Method	Description
Supported methods	
PUT	Use this method to: • Store an object in the namespace • Add or replace custom metadata for an existing object When you store an object in the namespace, HCP uses the ETag response header to return the cryptographic hash value for the object.
GET	Use this method to retrieve an object, metafile, or directory from the namespace.
HEAD	Use this method to check whether an object, directory, or symbolic link exists in the namespace.
MKCOL	Use this method to create a new directory in the namespace.
PROPPATCH	Use this method to: • Change system metadata associated with an object. • Store dead properties as custom metadata (when this capability is enabled in the namespace configuration). For information on this, see "Using the custom-metadata.xml file to store dead properties" on page 160.
PROPFIND	Use this method to retrieve metadata associated with an object, including both system metadata and dead properties stored as custom metadata. You can use **PROPFIND** to retrieve dead properties only when the namespace is configured to store dead properties as custom metadata.
COPY	Use this method to copy an object from one location to another. A request to copy an object fails if an object with the same name already exists at the target location.
MOVE	Use this method to move an object from one location to another. A request to move an object fails if an object with the same name already exists at the target location. Additionally, a **MOVE** request fails if the source object is under retention.
DELETE	Use this method to delete an object, directory, symbolic link, or custom metadata from the namespace.

(Continued)

Method	Description
LOCK	Use this method to lock an object on a single node.
UNLOCK	Use this method to unlock an object on a single node.
OPTIONS	Use this method to see which WebDAV methods are supported for the specified object, directory, or symbolic link.
Unsupported methods	
POST	N/A
TRACE	N/A

URLs for WebDAV access to the default namespace

Depending on the method you're using and what you want to do, the URL you specify can identify any of:

- The namespace as a whole
- A directory
- An object
- A symbolic link
- A metadirectory
- A metafile for an object or directory

Note: To access the namespace through WebDAV directly from a Windows client, add the namespace as a network share, using any of the URL formats described in "URL formats" below. When you share the namespace in this way, it appears to be part of the local file system in Windows Explorer in the same way it does with CIFS access. For information on CIFS access to the default namespace, see "Namespace access with CIFS" on page 168.

URL formats

The following sections show the URL formats you can use for default namespace access. In these formats, you can identify the HCP system by either DNS name or IP address. For information on the relative advantages of DNS names and IP addresses, see "DNS name and IP address considerations" on page 194.

If the HCP system does not support DNS, you can use the client `hosts` files to enable access to the default namespace by hostname. For information on configuring hostnames on the client system, see "Using a hosts file" on page 193.

Note: The URL formats and examples that follow show *http*. Your namespace administrator can configure the namespace to require SSL security for the HTTP protocol. In this case, you need to specify *https* instead of *http* in your URLs.

URL for the namespace as a whole

A URL that identifies the default namespace as a whole has one of these formats:

http://default.default.*hcp-domain-name***/webdav**

http://*node-ip-address***/webdav**

For example:

http://default.default.hcp.example.com/webdav

URLs for objects, directories, and symbolic links

To access an object, directory, or symbolic link in the default namespace, you use a URL that includes the `fcfs_data` directory. The format for this is one of:

http://default.default.*hcp-domain-name***/webdav/fcfs_data**
[*/directory-path*[*/object-name*]]

http://*node-ip-address***/webdav/fcfs_data**[*/directory-path*
[*/object-name*]]

Here's a sample URL that identifies a directory:

http://default.default.hcp.example.com/webdav/fcfs_data/Corporate/Employees

Here's a sample URL that identifies an object:

http://192.168.210.16/webdav/fcfs_data/Corporate/Employees/
2193_John_Doe

You cannot tell from a URL whether it represents an object, directory, or symbolic link.

URLs for metafiles and metadirectories

To access a metafile or metadirectory, you use a URL that includes the `fcfs_metadata` metadirectory. The format for this is one of:

http://default.default.*hcp-domain-name***/webdav/fcfs_metadata/***metadirectory-path*[*/metafile-name*]

http://*node-ip-address***/webdav/fcfs_metadata**[*/metadirectory-path* [*/metafile-name*]]

Note: You can browse the `fcfs_metadata` metadirectory with WebDAV only when viewing the namespace in a web browser. For more information on this, see "Browsing the namespace with WebDAV" on page 153.

Here's a sample URL that identifies a metadirectory:

http://192.168.210.16/webdav/fcfs_metadata/Corporate/Employees/2193_John_Doe

Here's a sample URL that identifies a metafile:

http://default.default.hcp.example.com/webdav/fcfs_metadata/Corporate/
Employees/2193_John_Doe/shred.txt

URL considerations

The following considerations apply to specifying URLs in WebDAV requests against the default namespace. For considerations that apply specifically to naming new objects, see "Object naming considerations" on page 16.

URL length

The portion of a URL after `fcfs_data` or `fcfs_metadata`, excluding any appended query parameters, is limited to 4,095 bytes. If a WebDAV request includes a URL that violates that limit, HCP returns a status code of 414.

Object names with non-ASCII, nonprintable characters

When you store an object or directory with non-ASCII, nonprintable characters in its name, those characters are percent encoded in the name displayed back to you. In the `core-metadata.xml` file for an object, those characters are also percent encoded, but the percent signs (%) are not displayed.

Regardless of how the name is displayed, the object or directory is stored with its original name, and you can access it either by its original name or by the name with the percent-encoded characters.

Percent-encoding for special characters

Some characters have special meaning when used in a URL and may be interpreted incorrectly when used for other purposes. To avoid ambiguity, percent-encode the special characters listed in the table below.

Character	Percent-encoded Value
Space	%20
Tab	%09
New line	%0A
Carriage return	%0D
+	%2B
%	%25
#	%23
?	%3F
&	%26

Percent-encoded values are not case sensitive.

Quotation marks with URLs in command lines

When using a command-line tool to access the default namespace through WebDAV, you work in a Unix, Mac OS X, or Windows shell. Some characters in the commands you enter may have special meaning to the shell. For example, the ampersand (&) often indicates that a process should be put in the background.

To avoid the possibility of the Windows, Unix, or Mac OS X shell misinterpreting special characters in a URL, always enclose the entire URL in double quotation marks.

Browsing the namespace with WebDAV

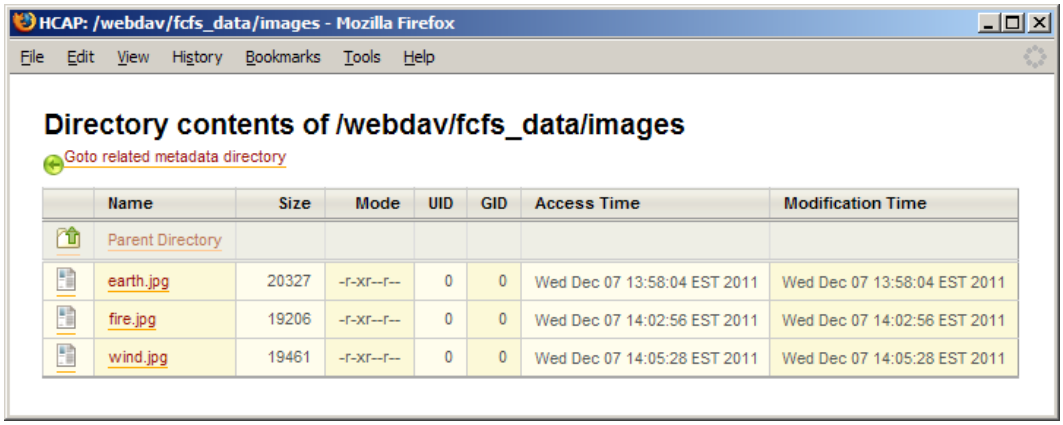
To view the namespace content in a web browser with WebDAV, enter a WebDAV namespace access URL in the browser address field:

- If you enter the URL for the entire namespace, the browser lists the two root directories, `fcfs_data` and `fcfs_metadata`.
- If you enter the URL for a directory or metadirectory, the browser lists the contents of that directory.

Note: Some browsers may not be able to successfully render pages for directories that contain a very large number of objects, directories, and symbolic links.

- If you enter the URL for an object, the browser downloads the object data and either opens it in the default application for the content type or prompts to open or save it.
- If you enter the URL for a metafile, the browser downloads and displays the contents of that metafile.

For the first two cases, HCP provides an XML stylesheet that determines the appearance of the browser display. The sample browser window below shows what this looks like for the `images` directory.



Tip: You can use the view-source option in the web browser to see the XML that HCP returns.

WebDAV properties

WebDAV properties are name/value pairs that provide information about an object, directory, or symbolic link. To store and retrieve property values, you use the **PROPPATCH** and **PROPFIND** methods, respectively.

Properties exist in XML namespaces. To fully identify a property, you need to specify the namespace it's in as well as the property name.

All the properties defined in the WebDAV specification are in the DAV: namespace. So, for example, the standard WebDAV property named creationdate is in the DAV: namespace.

For information on the standard WebDAV properties, see RFC 2518 or 4918.

Live and dead properties

Properties can be **live** or **dead**. Live properties are properties that are known to the WebDAV server (in this case, HCP). The server can respond to changes you make to these properties and can also dynamically modify their values.

Dead properties are properties that are not known to the WebDAV server. The server stores and retrieves these properties but does not do anything else with them. Dead properties can be in any XML namespace.

A **PROPFIND** request for all properties returns both live and dead properties. However, when responding to such a request, the server may omit live properties whose values are expensive to calculate.

In response to a **PROPFIND** request for all properties, HCP may omit some standard WebDAV properties but always includes all properties defined in the HCP XML namespace. For more information on these properties, see "HCP-specific metadata properties for WebDAV" on page 155.

Storage properties

HCP supports RFC 4331, which defines two additional live properties in the DAV: HTTP URL namespace. These properties, which are described in the table below, provide storage statistics for the namespace.

Property	Description
quota-available-bytes	The amount of storage space, in bytes, currently available for storing additional objects

(Continued)

Property	Description
quota-used-bytes	The amount of storage space, in bytes, currently occupied by all objects in the namespace

HCP-specific metadata properties for WebDAV

HCP recognizes all the live properties defined in the DAV: namespace. Additionally, it provides its own XML namespace with live properties that allow you to store and retrieve HCP system metadata. This namespace is specified as:

http://www.hds.com/hcap/webdav/

Note: In a WebDAV **PROPPATCH** or **PROPFIND** request, the HCP XML namespace must be specified exactly as shown above. The namespace name must include **hcap**, not hcp, in the URL.

As with the standard WebDAV properties, you use the **PROPPATCH** and **PROPFIND** methods to change and retrieve the values of the HCP-specific properties.

Metadata properties for objects

The table below describes the metadata properties HCP provides for objects. For more information on the possible values for these properties, see Chapter 3, "Object properties," on page 33.

Metadata Property	Description
access-time	The value of the POSIX **atime** attribute for the object. You can retrieve, but not change, the value of this property.
change-time	The value of the POSIX **ctime** attribute for the object. You can retrieve, but not change, the value of this property.
creation-time	The date and time the object was stored in the namespace. You can retrieve, but not change, the value of this property.
dpl	The object DPL. You can retrieve, but not change, the value of this property.

(Continued)

Metadata Property	Description
gid	The ID of the owning group for the object. To change the owning group, specify a valid group ID.
hash-scheme	The name of the cryptographic hash algorithm used to calculate the cryptographic hash value for the object. You can retrieve, but not change, the value of this property.
hash-value	The cryptographic hash value for the object. You can retrieve, but not change, the value of this property.
index	The index setting for the object, either **true** (index) or **false** (don't index). To change this value, specify **true** or **false**.
mode	The object permissions as an octal value. To change the permissions, specify a valid octal value for permissions.
replication	An indication of whether the object has been replicated, either **true** (replicated) or **false** (not replicated). You can retrieve, but not change, the value of this property.
replication-collision	An indication of whether the object is flagged as a replication collision, either **true** (flagged) or **false** (not flagged).
retention-value	0, -1, -2, or seconds since January 1, 1970 at 00:00:00. To change this value, you can specify any of these values, any of the valid values for the **retention-string** property, or an offset. For information on offsets, see "Specifying an offset" on page 49.
retention-string	**Deletion Allowed**, **Deletion Prohibited**, **Initial Unspecified**, or a date and time in this format: *yyyy-MM-dd***T***hh:mm:ssZ** *Z* represents the offset from UTC and is specified as: (**+**|**-**)*hhmm* To change this value, specify any of these values, any of the valid values for the **retention-value** property, or an offset.

(Continued)

Metadata Property	Description
retention-class	The name of the retention class for the object (such as Hlth-107). To change this value, specify a valid retention class name.
retention-hold	The hold status for the object, either **true** (on hold) or **false** (not on hold). To change this value, specify **Hold** or **Unhold**.
shred	The shred setting for the object, either **true** (shred) or **false** (don't shred). To change this value, specify **true** or **false**.
uid	The user ID of the object owner. To change the object owner, specify a valid user ID.
update-time	The value of the POSIX **mtime** attribute for the object. You can retrieve, but not change, the value of this property.

Metadata properties for directories

The table below describes the metadata properties HCP provides for directories. For more information on the possible values for these properties, see Chapter 3, "Object properties," on page 33.

Metadata Property	Description
access-time	The value of the POSIX **atime** attribute for the directory. You can retrieve, but not change, the value of this property.
change-time	The value of the POSIX **ctime** attribute for the directory. You can retrieve, but not change, the value of this property.
creation-time	The date and time the directory was created in the namespace. You can retrieve, but not change, the value of this property.
gid	The ID of the owning group for the directory. To change the owning group, specify a valid group ID.

(Continued)

Metadata Property	Description
index	The index setting for the directory, either **true** (index) or **false** (don't index). To change this value, specify **true** or **false**.
mode	The directory permissions as an octal value. To change the permissions, specify a valid octal value for permissions.
retention	Any valid directory retention setting. To change this value, specify any valid retention setting for a directory. For these settings, see "Changing retention settings" on page 45.
retention-class	The name of the retention class for the directory (such as Hlth-107). To change this value, specify a valid retention class name.
shred	The shred setting for the directory, either **true** (shred) or **false** (don't shred). To change this value, specify **true** or **false**.
uid	The user ID of the directory owner. To change the directory owner, specify a valid user ID.
update-time	The value of the POSIX **mtime** attribute for the directory. You can retrieve, but not change, the value of this property.

PROPPATCH example

Here's a sample WebDAV **PROPPATCH** request that changes the retention setting for the object named `wind.jpg` in the `images` directory.

Request with cadaver command line

```
propset images/wind.jpg retention-string 2015-09-30T17:00:00-0400
```

Request XML body

```
<?xml version="1.0" encoding="utf-8" ?>
<ns0:propertyupdate xmlns:ns0="DAV:">
    <ns0:set>
        <ns0:prop>
            <ns1:retention-string xmlns:ns1="http://www.hds.com/hcap/webdav/">
```

```
                2015-09-30T17:00:00-0400
            </ns1:retention-string>
        </ns0:prop>
    </ns0:set>
</ns0:propertyupdate>
```

Response XML body

```
<?xml version="1.0" encoding="utf-8"?>
<D:multistatus xmlns:D="DAV:"
xmlns:HCAP="http://www.hds.com/hcap/webdav/"
xmlns="http://www.hds.com/hcap/webdav/">
    <D:response>
        <D:href>http://default.default.hcp.example.com/webdav/fcfs_data/images/
            wind.jpg
        </D:href>
        <D:propstat>
            <D:prop>
                <ns1:retention-string
                xmlns:ns1="http://www.hds.com/hcap/webdav/"/>
            </D:prop>
            <D:status>HTTP/1.1 200 OK</D:status>
        </D:propstat>
    </D:response>
</D:multistatus>
```

PROPFIND example

Here's a sample WebDAV **PROPFIND** request that returns the UID for the object named `wind.jpg` in the `images` directory.

Request with cadaver command line

```
propget images/wind.jpg uid
```

Request XML body

```
<?xml version="1.0" encoding="utf-8" ?>
<ns0:propfind xmlns:ns0="DAV:">
    <ns0:prop>
        <ns1:uid xmlns:ns1="http://www.hds.com/hcap/webdav/" />
    </ns0:prop>
</ns0:propfind>
```

Response XML body

```
<?xml version="1.0" encoding="utf-8"?>
<D:multistatus xmlns:D="DAV:"
xmlns:HCAP="http://www.hds.com/hcap/webdav/">
    <D:response>
        <D:href>http://default.default.hcp.example.com/webdav/fcfs_data/images/
            wind.jpg
        </D:href>
        <D:propstat>
            <D:prop>
                <HCAP:uid>0</HCAP:uid>
            </D:prop>
            <D:status>HTTP/1.1 200 OK</D:status>
        </D:propstat>
    </D:response>
</D:multistatus>
```

Using the custom-metadata.xml file to store dead properties

HCP can be configured to store WebDAV dead properties in the `custom-metadata.xml` file for an object. When HCP is configured this way, you use the **PROPPATCH** and **PROPFIND** methods to store and retrieve dead properties just as you do for live properties.

This is the only way HCP can store dead properties. When the namespace is not configured to use `custom-metadata.xml` files for dead properties and you try to store dead properties, HCP returns an error (return code 404).

To have the namespace configured to store dead properties in the `custom-metadata.xml` file, see your namespace administrator.

Note: HCP lets you set dead properties on directories. It uses an internal mechanism to store these properties.

Using the **PROPPATCH** method, you can change individual dead properties in the `custom-metadata.xml` file. You do not need to replace the entire file as you do when you use the file for custom metadata (as described in "Custom metadata" on page 60).

You can use the `custom-metadata.xml` file for an object to store either custom metadata or dead properties, but not both:

- If, after using the `custom-metadata.xml` file to store custom metadata, you try to store dead properties, HCP returns a 409 (Conflict) error. In this case, you must delete the existing `custom-metadata.xml` file before you can store dead properties for the object.

- If, after storing dead properties, you replace the `custom-metadata.xml` file with a new `custom-metadata.xml` file, the dead properties you stored are lost.

WebDAV usage considerations

The following sections present considerations that affect the use of the WebDAV protocol for namespace access. For additional considerations that are not specific to the WebDAV protocol, see Chapter 9, "General usage considerations," on page 191.

Basic authentication with WebDAV

The namespace can be configured to require basic authentication for WebDAV access. If basic authentication is enabled, you're prompted for a username and password when you access the namespace through WebDAV. For the username and password to use, see your namespace administrator.

WebDAV permission checking

The namespace configuration specifies the level of permission checking on WebDAV requests:

- **No permission checking** — HCP doesn't check permissions on any operations.

- **Permission checking only on the first object** — If you're adding an object to the default namespace, HCP checks the permissions only for the target directory. If you're performing an operation on an existing object, HCP checks the permissions only for that object.

- **Strict permission checking** — For all operations, HCP checks permissions for the target object as well as for all directories in the path to that object.

Strict permission checking enhances the security of stored data but results in slower performance. With no permission checking, performance is unaffected, but the security benefit is lost.

To learn the level of WebDAV permission checking in effect, see your namespace administrator.

WebDAV persistent connections

HCP supports persistent connections with WebDAV. Following a request for an operation, HCP keeps the connection open for 60 seconds so a subsequent request can use the same connection.

Persistent connections enhance performance because they avoid the overhead of opening and closing multiple connections. In conjunction with persistent connections, using multiple threads so that operations can run concurrently provides still better performance.

If the persistent connection timeout period is too short, tell your namespace administrator.

Note: With persistent connections, if a single node has more than 254 concurrent open connections, those above the first 254 may have to wait as long as ten minutes to be serviced. This includes connections where the request explicitly targeted the node, as well as connections where the HCP DNS name resolved to the target node.

To avoid this issue, either don't use persistent connections or ensure that no more than 254 threads are working against a single node at any given time.

WebDAV client timeouts with long-running requests

Some WebDAV operations can take a long time to complete. For example, a **GET** request on a very large directory can take quite a while to complete.

During such an operation, HCP does not communicate back to the client. As a result, the connection may time out on the client.

If an operation may take a long time, you should adjust the connection timeout setting on the client before making the request.

WebDAV object locking

To accommodate WebDAV clients that require locking functionality, HCP supports the **LOCK** and **UNLOCK** methods. However, a **LOCK** request locks the specified object only on the target node. As a result, locking an object does not prevent other nodes from modifying it.

Storing zero-sized files with WebDAV

When you store a zero-sized file with WebDAV, the resulting object has no data. Because WebDAV causes a flush and a close even when no data is present, this object is WORM and is treated like any other object in the namespace.

Using WebDAV with objects open for write

These considerations apply to using WebDAV to access objects that are open for write:

- If you try to write to the object, HCP returns a 409 (Conflict) error.
- If try to retrieve, delete, or check the existence of an object that's open for write, HCP returns a 404 (Not Found) error code and does not perform the operation.

Note: Depending on the timing, the delete request may result in a busy error. In that case, wait one or two seconds and then try the request again.

Failed WebDAV write operations

A WebDAV write operation is considered to have failed if either of these is true:

- The target node failed while the object was open for write.
- The TCP connection broke (for example, due to a front-end network failure or the abnormal termination of the client application) while the object was open for write.

Also, in some circumstances, a write operation is considered to have failed if another node or other hardware failed while the object was open for write.

If a write fails, HCP does not create a new object.

Tip: If a write operations fails, retry the request.

Multithreading with WebDAV

With WebDAV, only one client can write to a given object at one time. A multithreaded client can write to multiple objects at the same time but cannot have multiple threads writing to the same object.

Multiple clients can use WebDAV to read the same object concurrently. Similarly, a multithreaded client can use multiple threads to read a single object. However, because the reads can occur out of order, you generally get better performance by using one thread per object.

HCP has a limit of 255 concurrent WebDAV connections per node, with another 20 queued.

Note: HTTP and WebDAV share the same connection pool.

WebDAV return codes

The table below describes the possible return codes for WebDAV requests against the default namespace.

Code	Meaning	Description
200	OK	**GET**, **HEAD**, **PROPPATCH**, or **LOCK**: HCP successfully completed the request.
201	Created	**PUT**, **MKCOL**, **COPY**, or **MOVE**: HCP successfully completed the request. (For **COPY** or **MOVE**, no object existed at the target location.)
204	No Content	**COPY**, **MOVE**, or **DELETE**: HCP successfully completed the request. (For **COPY** or **MOVE**, a deletable object existed at the target location.) **GET**, **HEAD**, or **DELETE** of custom metadata: The specified object exists but does not have custom metadata.
206	Partial Content	**GET**: HCP successfully retrieved the data in the byte range specified in the request.

(Continued)

Code	Meaning	Description
207	Multi-Status	**PROPPATCH**, **PROPFIND**, or **DELETE** for a directory: An operation generated multiple return codes. The response body contains an XML document that shows the return codes and the names of the objects to which they apply.
400	Bad Request	All methods: The request is not well-formed. Correct the request and try again. **PROPPATCH** or **PROPFIND**: The request XML is invalid. **PUT**: For a request to add custom metadata, the namespace is configured with custom metadata XML checking enabled, and the request includes custom metadata that is not well-formed XML.
403	Forbidden	For all methods, one of: • The namespace does not exist. • The WebDAV protocol is not enabled for the default namespace. • The URL specifies *https* and the namespace configuration does not support SSL. • You don't have permission to perform the requested operation. **MKDIR**: You cannot create a directory in the specified location. **PROPPATCH**: The requested change is not allowed. **COPY** or **MOVE**: The specified source and destination locations are the same. **DELETE**: The specified object is under retention.
404	Not Found	**GET**, **HEAD**, **PROPPATCH**, **PROPFIND**, **COPY**, **MOVE**, **DELETE**, **LOCK**, or **UNLOCK**: HCP could not find the object, metafile, or directory specified in the request.
405	Method Not Allowed	**MKCOL**: HCP could not create the directory because it already exists. **DELETE**: HCP could not delete the specified object or `custom-metadata.xml` file because it is currently being written to the namespace.

(Continued)

Code	Meaning	Description
409	Conflict	**PUT**: HCP could not store the object because it already exists. **PUT**, **MKCOL**, **COPY**, or **MOVE**: One or more directories in the target path do not exist. **PROPPATCH**: HCP could not store dead properties in the specified `custom-metadata.xml` file because it already contains custom metadata.
412	Precondition Failed	**COPY** or **MOVE**: The operation failed because either: • HCP could not correctly copy or move the object metadata • The target object already exists and could not be deleted **LOCK**: HCP could not lock the specified object.
414	Request URI Too Long	All methods: The portion of the URL following `fcfs_data` or `fcfs_metadata` is longer than 4,095 bytes.
416	Requested Range Not Satisfiable	**GET**: For a byte-range request, either: • The specified start position is greater than the size of the requested data. • The size of the specified range is zero.
423	Locked	**PUT**, **PROPPATCH**, **COPY**, **MOVE**, **DELETE**, or **LOCK**: HCP could not perform the requested operation because the target object is locked.
500	Internal Server Error	All methods: An internal error occurred. Try the request again, gradually increasing the delay between each successive attempt. If this happens repeatedly, please contact your namespace administrator.
503	Service Unavailable	One of: • HCP is temporarily unable to handle the request, probably to due to system overload, maintenance, or upgrade. • HCP tried to read the object from another system in the replication topology but could not. In either case, try the request again, gradually increasing the delay between each successive attempt.

(Continued)

Code	Meaning	Description
507	Insufficient Storage	**PUT**: Not enough space is available to store the object. Try the request again after objects are deleted from the repository or the system storage capacity is increased. **PROPPATCH**, **MKCOL**, or **COPY**: Not enough space is available to complete the request. Try the request again after objects are deleted from the repository or the system storage capacity is increased.

6

CIFS

CIFS is one of the industry-standard protocols HCP supports for namespace access. To access the namespace through CIFS, you can write applications that use any standard CIFS client library, or you can use the Windows GUI or a **Command Prompt** window to access the namespace directly.

Using the CIFS protocol, you can store, view, retrieve, and delete objects. You can also change certain system metadata for existing objects.

For you to access the namespace through CIFS, this protocol must be enabled in the namespace configuration. If you cannot access the namespace in this way, see your namespace administrator.

This chapter explains how to use CIFS for namespace access.

Namespace access with CIFS

You access a namespace through CIFS by mapping a directory in the default namespace to a network drive on a CIFS client. For both data and metadata, you can map the root directory (`fcfs_data` or `fcfs_metadata`) or any directory or metadirectory under it. Additionally, you can have multiple directories mapped at the same time.

Once mapped, the namespace appears to be part of the local file system, and you can perform any of the operations HCP supports for CIFS. On a Windows client, this includes dragging and dropping files and directories to the namespace and, likewise, objects and directories from the namespace.

When mapping the namespace, you can use either the DNS name of the HCP system or the IP address of a node in the HCP system. Here's the format for each method:

\\cifs.*hcp-domain-name***\(fcfs_data|fcfs_metadata)[***directory-path***]**

*****node-ip-address***\(fcfs_data|fcfs_metadata)[***directory-path***]**

Examples:

\\cifs.hcp.example.com\fcfs_data

\\192.168.210.16\fcfs_data\images

\\cifs.hcp.example.com\fcfs_metadata

For information on the relative advantages of DNS names and IP addresses, see "DNS name and IP address considerations" on page 194.

Note: When working with objects and metafiles at the same time, you need at least two separate shares of the namespace — one to a directory and one to a metadirectory.

CIFS examples

The following sections show examples of using CIFS to access the default namespace. Each example shows both a Windows command and Python code that implements the same command.

These examples assume that the `fcfs_data` directory is mapped to the X: drive and the `fcfs_metadata` metadirectory is mapped to the Y: drive.

CIFS example 1: Storing an object

This example stores an object named `wind.jpg` in the existing `images` directory by copying a file of the same name from the local file system.

Windows command

```
copy wind.jpg x:\images
```

Python code

```
import shutil
shutil.copy("wind.jpg", "x:\\images\\wind.jpg")
```

CIFS example 2: Changing a retention setting

This example extends the retention period for the `wind.jpg` object by one year. If this object is still open due to lazy close, changing the retention setting closes it.

For information on lazy close, see "CIFS lazy close" on page 172.

Windows command

```
echo +1y > y:\images\wind.jpg\retention.txt
```

Python code

```
retention_value = "+1y"
retention_fh = file("y:\\images\\wind.jpg\\retention.txt")
try:
  retention_fh.write(retention_value)
finally:
  retention_fh.close()
```

CIFS example 3: Retrieving an object

This example retrieves the object named `wind.jpg` from the namespace and stores the resulting file with the same name in the existing `RetrievedFiles` directory.

Windows command

```
copy x:\images\wind.jpg RetrievedFiles
```

Python code

```
import shutil
shutil.copy("x:\\images\\wind.jpg", "RetrievedFiles\\wind.jpg")
```

CIFS example 4: Retrieving deletable objects

This example retrieves all objects that can be deleted from the `images` directory and lists them in the `HCP\DeletableObjects` directory. These are the objects listed in the `expired` metadirectory. They are retrieved without any data.

For more information on the `expired` metadirectory, see "Metadirectories for directories" on page 18.

Windows command

```
copy  y:\images\.directory-metadata\info\expired  HCP\DeletableObjects
```

Python code

```
import shutil
import glob
expiredFileDir = "y:\\images\\.directory-metadata\\info\\expired\\"
for expiredFile in glob.glob(expiredFileDir + "*"):
  shutil.copy(expiredFile, "HCP\\DeletableObjects")
```

CIFS usage considerations

The following sections present considerations that affect the use of the CIFS protocol for namespace access. For additional considerations that are not specific to the CIFS protocol, see Chapter 9, "General usage considerations," on page 191.

CIFS case sensitivity

The Windows operating system is case preserving but not case sensitive. The HCP CIFS implementation, by default, is both case preserving and case sensitive. One result of this discrepancy is that Windows applications that don't observe differences in case may not be able to access objects by name.

For example, suppose a Windows application adds a file named `File.txt` to the namespace by using the CIFS protocol. CIFS preserves case, so the namespace then contains an object named `File.txt`. Now suppose the application tries to retrieve that object using the name `file.txt`. CIFS is case sensitive, so it passes the request to HCP with only the name `file.txt`. It doesn't include any case variations on the name, such as `File.TXT` or `FILE.txt`. As a result, HCP cannot find the object.

You can ask your namespace administrator to make the CIFS implementation case insensitive. However, this change has two consequences that affect object reads:

- It slows performance.
- If the namespace contains multiple objects with names that differ only in case, HCP may return the wrong object.

CIFS permission translations

When you view an object on a Windows client, CIFS translates the POSIX permission settings used in the namespace into settings Windows understands. The table below shows how CIFS maps POSIX permissions to Windows permissions.

Permissions in the namespace	Permissions in Windows
r--	Read
-w-	Write
--x	*None*
rw-	Read Write
r-x	Read Read & Execute
-wx	*None*
rwx	Read Write Read & Execute Modify Full Control
---	*None*

Changing directory permissions when using Active Directory

When using Active Directory for user authentication with CIFS, you cannot use Windows properties to change permissions for a directory. Although the operation appears to work, no changes are actually made.

Creating an empty directory with atime synchronization in effect

When you use Windows Explorer to create a new directory, Windows automatically names it `New Folder`. This is also true for directories you create in the namespace.

Normally, you can rename an empty directory in the namespace. However, if **atime** synchronization is in effect, you cannot do this. As a result, the name of the new directory remains `New Folder`.

For more information on **atime** synchronization, see "atime synchronization with retention" on page 51.

CIFS lazy close

When writing a file to the namespace, CIFS can cause a flush at any time. After each flush or write, HCP waits a short amount of time for the next one. HCP considers the resulting object to be complete and closes it if no write occurs within that time. This event is called **lazy close**.

If you set retention on an object during the lazy close period, HCP closes the object immediately. The object becomes WORM, and retention applies, even if the object contains no data. However, if the directory that contains the object and its corresponding metadirectory are shared on two different nodes in the HCP system, setting retention during the lazy close period does not close the object.

Storing zero-sized files with CIFS

When you store a zero-sized file with CIFS, the resulting object has no data. After lazy close occurs, the object becomes WORM and is treated like any other object in the namespace.

Out-of-order writes with CIFS

CIFS can write the data for an object out of order. If HCP receives an out-of-order write for a large file (200,000 bytes or larger), it discards the cryptographic hash value it already calculated. The object then has no hash value until one of these occurs:

- HCP returns to the object at a later time and calculates the hash value for it.
- A user or application opens or downloads the `hash.txt` metafile for the object, which causes HCP to calculate the hash value. However, because HCP calculates this value asynchronously, the value may not be immediately available. This is particularly true for large objects.

Using CIFS with Objects open for write

These considerations apply to objects that are open for write through any protocol:

- While an object is open for write through one IP address, you cannot open it for write through any other IP address.
- You can read an object that is open for write from any IP address, even though the object data may be incomplete. A read against the node hosting the write may return more data than a read against any other node.
- While an object is open for write, you cannot delete it.

Note: Depending on the timing, the delete request may result in a busy error. In that case, wait one or two seconds and then try the request again.

- While an object that's open for write has no data:
  - It is not WORM
  - It may or may not have a cryptographic hash value
  - It is not subject to retention
  - It cannot have custom metadata
  - It is not indexed

Failed CIFS write operations

A CIFS write operation is considered to have failed if the target node failed while the object was open for write. Also, in some circumstances, a write operation is considered to have failed if another node or other hardware failed while the object was open for write.

A CIFS write operation is *not* considered to have failed if the TCP connection broke. This is because HCP doesn't see the failure. In this case, lazy close applies, and the object is considered complete.

Objects left by failed CIFS write operations:

- May have none, some, or all of their data
- If partially written, may or may not have a cryptographic hash value
- If the failure was on the HCP side, remain open and:
  - Are not WORM
  - Cannot have custom metadata
  - Are not indexed
  - Are not replicated
- If the failure was on the client side, are WORM after the lazy close

If a write operation fails, delete the object and try the write operation again.

Note: If the object is WORM, any retention setting applies. In this case, you may not be able to delete the object.

Temporary files created by Windows clients

During certain operations, Windows clients may create temporary files. In the namespace, these files correspond to successfully created objects.

As with any other new object stored through the CIFS protocol, the retention period for such an object is determined by its parent directory. If the object ends up being under retention, it remains in the namespace after the operation completes and cannot be deleted until its retention period expires.

Tip: When writing applications, be sure to take into account that objects created for temporary files may be closed and made WORM unexpectedly. Once this happens, the application cannot reopen the object.

Multithreading with CIFS

With CIFS, multiple concurrent threads can write to the same object, but only if they are working against the same node. Multiple concurrent threads can read the same object on the same or different nodes.

Note: With a single share of the namespace, concurrent threads are always working against the same node.

HCP doesn't limit the number of concurrent CIFS threads per node but does limit the total number of outstanding requests using all protocols to 500 per node.

Note: CIFS and NFS share the same thread pool.

CIFS return codes

The table below describes the possible return codes for CIFS requests against the namespace.

Code	Description
NT_STATUS_ACCESS_DENIED	The requested operation is not allowed. Reasons for this return code include attempts to: • Rename an object • Rename a directory that contains one or more objects • Overwrite an object • Modify the content of an object • Delete an object that's under retention • Delete a directory that contains one or more objects • Add a file (other than a file containing custom metadata), directory, or symbolic link anywhere in the metadata structure • Delete a metafile or metadirectory • Create a hard link
NT_STATUS_IO_DEVICE_ERROR	The requested operation would shorten the retention period of the specified object, which is not allowed.
NT_STATUS_RETRY	HCP tried to read the object from another system in the replication topology but could not.

7

NFS

NFS is one of the industry-standard protocols HCP supports for namespace access. To access the namespace through NFS, you can write applications that use any standard NFS client library, or you can use the command line in an NFS client to access the namespace directly.

Using the NFS protocol, you can store, view, retrieve, and delete objects. You can also change certain system metadata for existing objects.

For you to access the namespace through NFS, this protocol must be enabled in the namespace configuration. If you cannot access the namespace in this way, see your namespace administrator.

This chapter explains how to use NFS for namespace access.

Namespace access with NFS

You access the namespace through NFS by mounting a namespace directory on an NFS client. For both data and metadata, you can mount the root directory (`fcfs_data` or `fcfs_metadata`) or any directory or metadirectory under it. Additionally, you can have multiple directories mounted at the same time.

Once mounted, the namespace appears to be part of the local file system, and you can perform any of the operations HCP supports for NFS.

When mounting the namespace, you can use either the DNS name of the HCP system or the IP address of a node in the system. Here's the format for each method:

```
mount -o tcp,vers=3,timeo=600,hard,intr
  -t nfs nfs.hcp-domain-name:/(fcfs_data|fcfs_metadata)
  [/directory-path] mount-point-path

mount -o tcp,vers=3,timeo=600,hard,intr
  -t nfs node-ip-address:/(fcfs_data|fcfs_metadata)
  [/directory-path] mount-point-path
```

The parameters shown are recommended but not required.

Examples:

```
mount -o tcp,vers=3,timeo=600,hard,intr -t nfs
    nfs.hcp.example.com:/fcfs_data datamount

mount -o tcp,vers=3,timeo=600,hard,intr -t nfs
    192.168.210.16:/fcfs_data/images HCP-images

mount -o tcp,vers=3,timeo=600,hard,intr -t nfs
    nfs.hcp.example.com:/fcfs_metadata metadatamount
```

Note: When mounting the namespace, do not specify the rsize and wsize options. Omitting these options causes HCP to use the optimal values based on system configuration.

For information on the relative advantages of DNS names and IP addresses, see "DNS name and IP address considerations" on page 194.

Note: When working with objects and metafiles at the same time, you need at least two separate mounts of the namespace — one to a directory and one to a metadirectory.

NFS examples

The following sections show examples of using NFS to access the namespace. Each example shows both a Unix command and Python code that implements the same command.

These examples assume that the `fcfs_data` directory is mounted at `datamount` and the `fcfs_metadata` metadirectory is mounted at `metadatamount`.

NFS example 1: Adding a file

This example stores an object named `wind.jpg` in the existing `images` directory by copying a file of the same name from the local file system.

Unix command

```
cp  wind.jpg  /datamount/images/wind.jpg
```

Python code

```
import shutil
shutil.copy("wind.jpg", "/datamount/images/wind.jpg")
```

NFS example 2: Changing a retention setting

This example extends the retention period for the `wind.jpg` object by one year. If this object is still open due to lazy close, changing the retention setting closes it. For information on lazy close, see "NFS lazy close" on page 181.

Unix command

```
echo  +1y  >  /metadatamount/images/wind.jpg/retention.txt
```

Python code

```
retention_value = "+1y"
retention_fh = file("/datamount/images/wind.jpg/retention.txt")
try:
  retention_fh.write(retention_value)
finally:
  retention_fh.close()
```

NFS example 3: Using atime to set retention

This example changes the value of the POSIX **atime** attribute for the `wind.jpg` object. If the namespace is configured to synchronize **atime** values with retention settings and the object has a retention setting that specifies a date or time in the future, this also changes the retention setting for the object.

For more information on **atime** synchronization, see "atime synchronization with retention" on page 51.

Unix command

```
touch -a -t 201505171200 /datamount/images/wind.jpg
```

Python code

```
import os
mTime = os.path.getmtime("/datamount/images/wind.jpg")
aTime = 1431878400 #12:00 May 17th 2015
os.utime("/datamount/images/wind.jpg", (aTime, mTime))
```

NFS example 4: Creating a symbolic link in the namespace

This example creates a symbolic link named `big_dipper` that references an object named `ursa_major.jpg`.

Unix command

```
ln -s /datamount/images/constellations/ursa_major.jpg/
    datamount/constellations/common_names/big_dipper
```

Python code

```
import os
os.symlink("/datamount/images/constellations/ursa_major.jpg",
  "/datamount/constellations/common_names/big_dipper"
```

NFS example 5: Retrieving an object

This example retrieves the object named `wind.jpg` from the namespace and stores the resulting file in the `retrieved_files` directory.

Unix command

```
cp /datamount/images/wind.jpg retrieved_files/wind.jpg
```

Python code

```
import shutil
shutil.copy("/datamount/images/wind.jpg", "retrieved_files/ \
  wind.jpg")
```

NFS example 6: Retrieving deletable objects

This example retrieves all objects that can be deleted from the `images` directory and lists them in the `hcp/deletable_objects` directory. These are the objects listed in the `expired` metadirectory. They are retrieved without any data.

For more information on the `expired` metadirectory, see "Metadirectories for directories" on page 18.

Unix command

```
cp  metadatamount/images/.directory-metadata/info/expired/*
    hcp/deletable_objects
```

Python code

```
import shutil
import glob
expiredFileDir = "/metadatamount/images/.directory-metadata/info/
  expired/"
for expiredFile in glob.glob(expiredFileDir + "*"):
  shutil.copy(expiredFile, "hcp/deletable_objects")
```

NFS usage considerations

The following sections present considerations that affect the use of the NFS protocol for namespace access. For additional considerations that are not specific to the NFS protocol, see Chapter 9, "General usage considerations," on page 191.

NFS lazy close

When writing a file to the namespace, NFS can cause a flush at any time and never issues a close. After each flush or write, HCP waits a short amount of time for the next one. If no write occurs within that time, HCP considers the resulting object to be complete and automatically closes it. This event is called **lazy close**.

If you set retention on an object during the lazy close period, HCP closes the object immediately. The object becomes WORM, and retention applies, even if the object contains no data. However, if the directory that contains the object and its corresponding metadirectory are mounted on two different nodes in the HCP system, setting retention during the lazy close period does not close the object.

Storing zero-sized files with NFS

When you store a zero-sized file with NFS, the resulting object has no data. After lazy close occurs, the object becomes WORM and is treated like any other object in the namespace.

Out-of-order writes with NFS

NFS can write the data for an object out of order. If HCP receives an out-of-order write for a large file (200,000 bytes or larger), it discards the hash value. The object then has no hash value until either of these occurs:

- HCP returns to the object at a later time and calculates the hash value for it.
- A user or application opens or downloads the `hash.txt` metafile for the object, which causes HCP to calculate the hash value. However, because HCP calculates this value asynchronously, the value may not be immediately available. This is particularly true for large objects.

Using NFS with objects open for write

These considerations apply to objects that are open for write through any protocol:

- While an object is open for write through one IP address, you cannot open it for write through any other IP address.
- You can read an object that is open for write from any IP address, even though the object data may be incomplete. A read against the node hosting the write may return more data than a read against any other node.

- While an object is open for write, you cannot delete it.

Note: Depending on the timing, the delete request may result in a busy error. In that case, wait one or two seconds and then try the request again.

- While an object that's open for write has no data:
  - It is not WORM
  - It may or may not have a cryptographic hash value
  - It is not subject to retention
  - It cannot have custom metadata
  - It is not indexed
  - It is not replicated

Failed NFS write operations

An NFS write operation is considered to have failed if the target node failed while the object was open for write. Also, in some circumstances, a write operation is considered to have failed if another node or other hardware failed while the object was open for write.

An NFS write operation is *not* considered to have failed if the TCP connection broke. This is because HCP doesn't see the failure. In this case, lazy close applies, and the object is considered complete.

Objects left by failed NFS write operations:

- May have none, some, or all of their data
- If partially written, may or may not have a cryptographic hash value
- If the failure was on the HCP side, remain open and:
  - Are not WORM
  - Cannot have custom metadata
  - Are not indexed

  - Are not replicated

- If the failure was on the client side, are WORM after the lazy close

If a write operation fails, delete the object and try the write operation again.

Note: If the object is WORM, any inherited retention setting applies. In this case, you may not be able to delete the object.

NFS reads of large objects

While HCP is reading very large objects (thousands of megabytes or more) through NFS, the system performance decreases.

Walking large directory trees

HCP occasionally reuses inode numbers. Normally, this has no impact. However, it can affect programs that walk the directory tree, like the Unix **du** command. If you run such a program against a very large directory tree, it may not go down certain subdirectory paths.

One way to prevent such problems is to work on directory segments, instead of the entire directory tree. For example, when you use the **du** command you can run the command against smaller segments of the directory hierarchy; then add the returned values together to get the total.

NFS delete operations

While an object is open for write through NFS on a given node, it cannot be deleted through NFS on other nodes.

NFS mounts on a failed node

If an HCP node fails, NFS mounts that target the failed node lose their connections to the namespace. To recover from a node failure, unmount the namespace at the current mount point. Then do one of these:

- Mount the namespace on a different node. You can do this by specifying either the DNS name of the HCP system or a specific node IP address in the **mount** command. If you specify a DNS name, HCP automatically selects a node from among the healthy ones.

- When the failed node reboots, remount the namespace on the same node.

Tip: You can use the NFS automounter on the client to automatically remount the namespace. Be sure to use the DNS name of the HCP system when you do this.

Multithreading with NFS

With NFS, multiple concurrent threads can write to the same object, but only if they are working against the same node. Multiple concurrent threads can read the same object on the same or different nodes.

Note: With a single mount point, concurrent threads are always working against the same node.

HCP doesn't limit the number of concurrent NFS threads per node but does limit the total number of outstanding requests using all protocols to 500 per node.

Note: CIFS and NFS share the same thread pool.

NFS return codes

The table below describes the possible return codes for NFS requests against the namespace.

Code	Description
EACCES	The requested operation is not allowed. Reasons for this return code include attempts to: • Rename an object • Rename a directory that contains one or more objects • Overwrite an object • Modify the content of an object • Add a file (other than a file containing custom metadata), directory, or symbolic link anywhere in the metadata structure • Delete a metafile or metadirectory
EAGAIN	HCP tried to read the object from another system in the replication topology but could not.
EIO	The requested operation is not allowed. This code is returned in response to attempts to: • Shorten the retention period of an object • Create a hard link
ENOTEMPTY	For an **rm** request to delete a directory, the specified directory cannot be deleted because it is not empty.
EROFS	For an **rm** request to delete an object, the specified object cannot be deleted because it is under retention.

8

SMTP

SMTP is one of the industry-standard protocols HCP supports for namespace access. This protocol is used only for storing email. Using SMTP, you (or an application) can send individual emails to the namespace. Your namespace administrator can also configure HCP to automatically store emails forwarded by mail servers.

For a user or application to send an individual email to the namespace, network connectivity must exist between the HCP system and the email server.

All email objects stored through SMTP can be accessed immediately through any other protocol.

For you to access the namespace through SMTP, this protocol must be enabled in the namespace configuration. If you cannot access the namespace in this way, see your namespace administrator.

This chapter explains how to send individual emails to the namespace and describes the naming conventions HCP uses when storing email objects.

Storing individual emails

To send an individual email to the namespace through SMTP, you include a namespace email address in the To, Cc, or Bcc list for the email. The email address has either of these formats:

username@*hcp-domain-name*

username@[*node-ip-address*]

username can be any well-formed email username. In the second format, the square brackets ([]) around the IP address are required.

Tip: For the username in the email address, use your own email username or the name of the application sending the email.

Examples:

jcrocus@hcp.example.com

hr-app-017@[192.168.210.16]

For information on the relative advantages of DNS names and IP addresses, see "DNS name and IP address considerations" on page 194.

Note: You can also store an email by first saving it and then using another protocol, such as HTTP, to store it in the namespace.

Naming conventions for email objects

HCP handles email objects the same way it handles other objects, except that for email stored through SMTP, HCP automatically generates directory paths and object names. It generates the paths directly under a parent directory that's specified in the namespace configuration. To learn the parent directory path, see your namespace administrator.

The namespace configuration also determines the ownership, permissions, retention setting, shred setting, and index setting for all email objects stored using the SMTP protocol.

Email directory and object names

The generated path and object name for email stored using SMTP consists of, in order:

- The email path specified in the namespace configuration, ending with a forward slash (/):

  Example: email/

- A system-generated numeric ID followed by a forward slash (/):

  Example: 341/

- The date and time the email was stored, in this format, followed by a hyphen (-):

  year/month/day/hour/minute/hour-minute-second.millisecond

  Example: 2013/03/02/02/47/02-47-22.186-

- An internally generated message ID followed by a hyphen:

  Example: 1D34A84A-

- A repeat of the system-generated numeric ID followed by a hyphen:

  Example: 341-

- A counter to ensure that all 1objects stored in the same millisecond have unique names followed by an at sign (@):

  Example: 0@

- The domain name of the sender contained in the From field of the mail header, followed by a hyphen (-):

  Example: example.com-

- The email suffix specified in the namespace configuration:

  Example: mbox.eml

Here's the complete path and object name for a sample email message:

/fcfs_data/email/341/2013/03/02/02/47/02-47-22.186-1D34A84A-341-0@example.com-mbox.eml

Note: The message ID that the mail server generates for an email ingested through the SMTP protocol can include one or more forward slashes (/) or colons (:). Before storing an email, HCP replaces each such slash or colon with a hyphen (-).

Email attachments

The namespace can be configured to store each email together with or separately from its attachments, if any. When stored together, the result is the single email object named as described above.

When stored separately, each attachment is in the same directory as the email object. The name of the attachment object is formed from the name of the email object (without the suffix) concatenated with a hyphen (-) and the name of the attached file.

Here's an example of the complete path and object names that result from storing two attachments separately from the email with which they arrive:

- Email:

  /fcfs_data/email/365/2013/03/02/17/12/17-12-29.522-4FE72776-365-0@example.com-mbox.eml

- First attachment:

  /fcfs_data/email/365/2013/03/02/17/12/17-12-29.522-4FE72776-365-0@example.com-Wetlands Guidelines 2011-10-01.pdf

- Second attachment:

  /fcfs_data/email/365/2013/03/02/17/12/17-12-29.522-4FE72776-365-0@example.com-Anytown-Lot53645-A.jpg

9

General usage considerations

This chapter contains usage considerations that affect the HTTP, WebDAV, CIFS, and NFS protocols in general. For considerations that apply to specific protocols, see the usage considerations in the individual protocol chapters.

Choosing an access protocol

The protocol you choose to use to access the namespace depends on a variety of factors. Some have to do with the protocols themselves, others with the environment in which you're working. For example, your client operating system may dictate the choice of protocol. Or, you may need new applications to be compatible with existing applications that already use a given protocol.

In terms of performance, for email archiving only, SMTP is the fastest protocol. For all other purposes, HTTP is the fastest with the lowest amount of overhead, and WebDAV is a close second. Both of these protocols are suitable for transferring large amounts of data. CIFS and NFS are significantly slower than HTTP and WebDAV.

In terms of features:

- With both HTTP and WebDAV:
  - Client libraries are available for many different programming languages.
  - You can store custom metadata in the namespace.
  - You can use SSL security for data transfers. The namespace configuration determines whether this feature is available.
  - You can retrieve object data by byte ranges.
- With HTTP:
  - Each operation can be completed in a single transaction, which provides better performance.
  - You can override metadata defaults when you add an object to the namespace.
  - HCP automatically creates any new directories in the paths for objects you add to the namespace.
  - You can identify objects by their cryptographic hash values.
- With WebDAV:
  - Some operations on directories, such as, **COPY**, **MOVE**, and **DELETE**, are performed in a single call.

  - You can recursively delete a directory and its subdirectories.

- With CIFS and NFS:

  - You get file-system semantics.

  - Multiple concurrent threads can write to the same object.

In terms of drawbacks:

- CIFS and NFS have lazy close (see "CIFS lazy close" on page 172 or "NFS lazy close" on page 181).

- With CIFS and NFS, performance degrades when write operations target directories with large numbers of objects (greater than 100,000).

- With CIFS and NFS, you need to use multiple mounts of the namespace to have HCP spread the load across the nodes in the system.

Using a hosts file

Typically, the HCP system is configured for DNS. If this is not the case, for access to the default namespace by hostname, the client `hosts` file must contain mappings from the hostname identifier for the default namespace to one or more HCP system IP addresses.

To find out whether the HCP system is configured for DNS, see your namespace administrator.

Note: For information on considerations for using hostnames and IP addresses, see "DNS name and IP address considerations" on page 194.

The location of the `hosts` file depends on the client operating system:

- On Windows, by default: c:\windows\system32\drivers\etc\hosts

- On Unix: /etc/hosts

- On Mac OS X: /private/etc/host

Hostname mappings

Each line in a hosts file is a mapping of a fully qualified hostname to an IP address. So, for example, if one of the IP addresses for the HCP system is 192.168.210.16, you would add this line to the hosts file to enable access to the default namespace:

```
192.168.210.16      default.default.hcp.example.com
```

The following considerations apply to hosts file entries:

- Each entry must appear on a separate line.
- Multiple hostnames in a single line must be separated by white space. With some versions of Windows, these must be single spaces.
- Each hostname can map to multiple IP addresses.

You can include comments in a hosts file either on separate lines or following a mapping on the same line. Each comment must start with a number sign (#). Blank lines are ignored.

For the IP addresses for the HCP system, contact your namespace administrator.

Hostname mapping considerations

An HCP system has multiple IP addresses. You can map the hostname for the default namespace to more than one of these IP addresses in the hosts file. The way multiple mappings are used depends on the client platform. For information on how your client handles multiple mappings in a hosts file, see your client documentation.

Note: If any of the HCP system IP addresses are unavailable, timeouts may occur when using a hosts file for system access.

DNS name and IP address considerations

You can access the namespace by specifying either the DNS name of the HCP system or the IP address of a node in the system. When you specify the DNS name, HCP selects the node for you from the currently available nodes. HCP uses a round-robin method to select the node, which spreads the load across the nodes.

Note: A node is not considered to be available for storing data if it has lost its connection to the corporate network.

When you specify IP addresses, your application must take responsibility for balancing the load among nodes. Also, you risk trying to connect (or reconnect) to a node that is not available. However, in several cases using IP addresses to connect to specific nodes can have advantages over using DNS names.

These considerations apply to deciding which technique to use:

- If your client uses a hosts file to map HCP hostnames to IP addresses, the client system has full responsibility for converting any DNS names to IP addresses. Therefore, HCP cannot spread the load or prevent attempts to connect to an unavailable node.
- If your client caches DNS information, connecting by DNS name may result in the same node being used repeatedly.
- When you access the HCP system by DNS name, HCP ensures that requests are distributed among nodes, but it does not ensure that the resulting loads on the nodes are evenly balanced.
- When multiple applications access the HCP system by DNS name concurrently, HCP is less likely to spread the load evenly across the nodes than with a single application.

Tip: When accessing the namespace by DNS name, you can ping the HCP system periodically to check whether you're getting connections to different nodes.

Directory structures

Because of the way HCP stores objects, the directory structures you create and the way you store objects in them can have an impact on performance. Here are some guidelines for creating effective directory structures:

- Plan your directory structures before storing objects. Make sure all namespace users are aware of these plans.
- Avoid structures that result in a single directory getting a large amount of traffic in a short time. For example, if you ingest objects rapidly, consider structures that do not store objects by date and time.

- If you do store objects by date and time, consider the number of objects ingested during a given period of time when planning the directory structure. For example, if you ingest several hundred files per second, you might use a directory structure such as year/month/day/hour/minute/second. If you ingest just a few files per second, a less fine-grained structure would be better.

- Follow these guidelines on directory depth and size:

  - Try to balance the namespace directory tree width and depth.

  - Do not create directory structures that are more than 20 levels deep. Instead, create flatter directory structures.

  - Avoid placing a large number of objects (greater than 100,000) in a single directory. Instead, create multiple directories and evenly distribute the objects among them.

Non-WORM objects

The namespace can contain objects that are not WORM:

- Objects that are open for write and have no data are not WORM.

- Empty objects written through CIFS and NFS are not WORM.

- Objects left by certain failed write operations are not WORM.

Objects that are not WORM are not subject to retention. You can delete these objects through any protocol. You can also overwrite them through the HTTP and WebDAV protocols without first deleting them.

Moving or renaming objects

You cannot move or rename an object in the default namespace. If a client tries either of these operations, the operation fails.

If this occurs, many clients automatically try to copy and delete the object instead. (This is how the HCP WebDAV **MOVE** method works.) If deletion is not allowed (for example, because the object is under retention), the original object remains in place, regardless of whether the copy is created.

When a copy is created and the original object is deleted, the move or rename operation appears to have been successful.

Deleting objects under repair

If you try to delete an object while HCP is repairing it, HCP returns a error response and the object is not deleted. For HTTP and WebDAV, the return value is a 409 (Conflict) error and for CIFS and NFS, the request may time out. When you get such errors, wait a few minutes and then try the request again.

Deleting directories

You can delete a directory only when it is empty. Some clients, however, can appear to delete nonempty directories, as long as those directories don't contain objects under retention. In such cases, what's really happening is that the client is using a single call to HCP to first delete the objects in the directory and then delete the now empty directory.

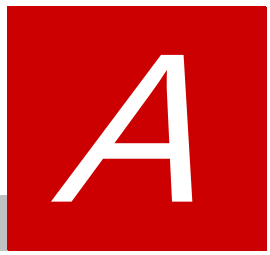
Multithreading

HCP lets multiple threads access the namespace simultaneously. Using multiple threads can enhance performance, especially when accessing many small files across multiple directories.

Here are some guidelines for the effective use of multithreading:

- Concurrent threads, both reads and writes, should be directed against different directories. If that's not possible, multiple threads working against a single directory are still better than a single thread.
- To the extent possible, concurrent threads should work against different nodes. If that's not possible, multiple threads working against a single node are still better than a single thread.

Tip: For better performance, consider limiting the number of concurrent read threads per node to 200 and concurrent write threads per node to 50 for small objects. For large objects, consider using fewer threads.



HTTP reference

This appendix contains a reference of HTTP methods available for accessing the namespace and the possible return codes and HCP-specific response headers.

For detailed information on using HTTP, see Chapter 4, “HTTP,” on page 69.

HTTP methods

The table below provides a quick reference to the HTTP methods you use to access and manage the namespace.

Method	Summary	Elements	Return codes / HCP-specific headers

CHMOD	Changes POSIX permissions for:
	• Objects
	• Directories
		• For objects, a URL with either:
		  ○ The object path
		  ○ The cryptographic hash value for the object
		• For directories, a URL with the directory path
		• This URL query parameter:
		  permissions=*octal-permission-value*
			Return codes
			Success: 200
			Error: 300, 400, 403, 404, 414, 500, 503
			Response headers
			X-ArcErrorMessage (if an error occurred and more information is available)
			X-ArcServicedBySystem

CHOWN	Changes POSIX owner and group IDs for:
	• Objects
	• Directories
		• For objects, a URL with either:
		  ○ The object path
		  ○ The cryptographic hash value for the object
		• For directories, a URL with the directory path
		• For both, these query parameters:
		  ○ **uid=***user-id*
		  ○ **gid=***group-id*
			Return codes
			Success: 200
			Error: 300, 400, 403, 404, 414, 500, 503
			Response headers
			X-ArcErrorMessage (if an error occurred and more information is available)
			X-ArcServicedBySystem

DELETE	Deletes:
	• Objects
	• Empty directories
	• Custom metadata
	• Symbolic links
		• For objects, a URL with either:
		  ○ The object path
		  ○ The cryptographic hash value for the object
		• For directories, custom metadata, and symbolic links, a URL with the applicable path
			Return codes
			Success: 200
			Error: 204, 300, 400, 403, 404, 409, 414, 500, 503
			Response headers
			X-ArcClusterTime
			X-ArcErrorMessage (if an error occurred and more information is available)
			X-ArcServicedBySystem

(Continued)

Method	Summary	Elements	Return codes / HCP-specific headers
GET	Retrieves: • Objects • Directory listings • HCP-specific metadata • Custom metadata	• For objects, a URL with either: o The object path o The cryptographic hash value for the object • For directories, HCP-specific metadata, and custom metadata, a URL with the directory or metafile path • To receive object data or custom metadata in gzip format, an Accept-Encoding request header that contains gzip or specifies * • To choose not to wait for delayed retrievals of objects or custom metadata, this query parameter: **nowait** • To retrieve object data and custom metadata together: o This query parameter: **type=whole-object** o To control the order of the returned information, an X-ArcCustomMetadataFirst request header with a value of **true** or **false** (the default) • For objects, optionally, an HTTP Range header specifying any of these zero-indexed byte ranges: o *start-position–end-position* o *start-position–* o *–offset-from-end*	**Return codes** *Success:* 200, 206 *Error:* 204, 300, 400, 403, 404, 406, 414, 416, 500, 503 **Response headers** *All*: X-ArcClusterTime X-ArcErrorMessage (if an error occurred and more information is available) X-ArcPermissionsUidGid X-ArcServicedBySystem X-ArcSize X-ArcTimes *Objects and directories*: X-ArcObjectType *Objects and custom metadata*: X-ArcCustomMetadataContentType X-ArcCustomMetadataFirst X-ArcDataContentType If response is in gzip-compressed format: Content-Encoding X-ArcContentLength

(Continued)

Method	Summary	Elements	Return codes / HCP-specific headers
HEAD	• Checks existence of: ○ Objects ○ Directories ○ Custom metadata • Checks available space and software version	• For objects, a URL with either: ○ The object path ○ The cryptographic hash value of the object • For directories or custom metadata, a URL with the directory or metafile path • For available space and software version, the namespace URL	**Return codes** *Success:* 200 *Error:* 204, 300, 400, 403, 404, 414, 500, 503 **Response headers** *All*: X-ArcClusterTime X-ArcErrorMessage (if an error occurred and more information is available) *All existence checks*: X-ArcPermissionsUidGid X-ArcServicedBySystem X-ArcSize X-ArcTimes *Space and version check*: X-ArcAvailableCapacity X-ArcTotalCapacity X-ArcSoftwareVersion
MKDIR	Creates a new empty directory	• A URL with the directory path • To specify metadata when creating the directory, any combination of these query parameters: ○ **gid=***group-id* ○ **uid=***user-id* ○ **directory_permissions=***octal-permission-value* ○ **atime=***unix-time-value* ○ **mtime=***unix-time-value*	**Return codes** *Success:* 201 *Error:* 400, 403, 409, 414, 500, 503 **Response headers** X-ArcErrorMessage (if an error occurred and more information is available) X-ArcServicedBySystem

(Continued)

Method	Summary	Elements	Return codes / HCP-specific headers
PUT	• Stores objects • Changes these metadata values for existing objects: ○ Retention setting ○ Shred setting ○ Index setting • Stores or replaces custom metadata	• To store objects: ○ A URL with the object path ○ A body containing object data ○ To send gzip-compressed data, a Content-Encoding request header with a value of gzip and a chunked transfer encoding ○ To store object data and custom metadata together: • An X-ArcSize request header with the object size in bytes • The **type=whole-object** query parameter • Custom metadata appended to the object data ○ To specify system metadata, any combination of: ◦ **gid=***user-id* ◦ **uid=***group-id* ◦ **directory_permissions=** *octal-permission-value* ◦ **file_permissions=** *octal-permission-value* ◦ **atime=***unix-time-value* ◦ **mtime=***unix-time-value* • **index**=(0|1) ◦ **retention=***retention-setting* • **shred=**(0|1) • To change HCP-specific metadata, a URL with the path for one of these metafiles: ○ retention.txt ○ shred.txt ○ index.txt • To store or replace custom metadata only: ○ A URL with the path for the `custom-metadata.xml` file ○ To send gzip-compressed data, a Content-Encoding request header with a value of gzip and a chunked transfer encoding	**Return codes** *Success:* 201 *Error:* 400, 403, 404, 414, 500, 503 **Response headers** *All*: X-ArcClusterTime X-ArcErrorMessage (if an error occurred and more information is available) X-ArcServicedBySystem *When adding object data and custom metadata together:* X-ArcCustomMetadataHash X-ArcHash

(Continued)

Method	Summary	Elements	Return codes / HCP-specific headers

TOUCH	Sets these POSIX attributes for objects and directories:
	• **atime**
	• **mtime**
		• A URL with either:
		  ○ The object path
		  ○ The cryptographic hash value of the object
		• Either or both of these query parameters:
		  ¡ **atime=***unix-time-value*
		  ¡ **mtime=***unix-time-value*
			Return codes
			Success: 200
			Error: 300, 400, 403, 404, 414, 500, 503
			Response headers
			X-ArcErrorMessage (if an error occurred and more information is available)
			X-ArcServicedBySystem

HTTP return codes

All responses to HTTP requests include a return code. The table below describes the possible return codes for all HTTP namespace requests.

Code	Meaning	Methods	Description
200	OK	CHMOD CHOWN DELETE GET HEAD TOUCH	HCP successfully performed the requested operation.
201	Created	MKDIR PUT	HCP successfully added an object, directory, or custom metadata to the namespace or replaced the custom metadata for an object.
204	No Content	DELETE, GET, or HEAD of custom metadata	The specified object does not have custom metadata.
206	Partial content	GET with a Range header	HCP successfully retrieved the data in the byte range specified in the request.
300	Multiple Choice	CHMOD CHOWN DELETE GET HEAD TOUCH	For a request by cryptographic hash value, HCP found two or more objects with the specified hash value.

(Continued)

Code	Meaning	Methods	Description

400	Bad request	All	One of:
			• The URL in the request is not well-formed.
			• The request specifies a cryptographic hash value that's not valid for the specified cryptographic hash algorithm.
			• A **PUT** request has a **type=whole-object** query parameter, and either:
			  ○ The request does not have an X-ArcSize header.
			  ○ The X-ArcSize header value is greater than the content length.
			• A **CHMOD**, **CHOWN**, or **TOUCH** request is missing a required query parameter.
			• For a **PUT** request to store custom metadata:
			  ○ The namespace has custom metadata XML checking enabled, and the request includes custom metadata that is not well-formed XML.
			  ○ The request is trying to store custom metadata for a directory or symbolic link.
			• A **PUT** request is trying to store a metafile for an object that does not exist.
			• A **PUT** request has a Content-Encoding header that specifies gzip, but the data is not gzip-compressed.
			• The request contains an unsupported query parameter or an invalid value for a query parameter.
			If more information about the error is available, the response headers include the HCP-specific X-ArcErrorMessage.

(Continued)

Code	Meaning	Methods	Description
403	Forbidden	All	One of: • The namespace does not exist. • The access method (HTTP or HTTPS) is disabled. • The namespace is not configured to allow the operation. • You do not have permission to perform the requested operation. • For a **CHMOD** or **CHOWN** request, the URL specifies a symbolic link. • For a **DELETE** request to delete an object, the object is under retention. • For a **DELETE** request to delete a directory, the directory is not empty. If more information about the error is available, the response headers include the HCP-specific X-ArcErrorMessage.
404	Not Found	CHMOD CHOWN DELETE GET HEAD TOUCH	One of: • HCP could not find the specified object, metafile, or directory. • For operations on custom metadata, HCP could not find the object to which the operation applies. For a request by cryptographic hash value, this return code can indicate that the object has not been indexed by the search facility selected for use with the Search Console. If the HDDS search facility is selected for use with the Search Console and the request specifies a cryptographic hash value, this return code can indicate that the value was found in HDDS but the object could not be retrieved from HCP.
406	Not Acceptable	GET	The request has an Accept-Encoding header that does not include gzip or specify *.

(Continued)

Code	Meaning	Methods	Description
409	Conflict	DELETE MKDIR PUT	One of: • **DELETE**: HCP could not delete the specified object, directory, or custom metadata because it is currently being written to the namespace. • **MKDIR**, **PUT**: HCP could not add the directory or object to the namespace because the directory or object already exists. • **PUT** of custom metadata: The object for which the custom metadata is being stored was ingested using CIFS or NFS, and the lazy close period for the object has not expired.
413	File Too Large	PUT of an object or custom metadata	One of: • Not enough space is available to store the data. Try the request again after objects are deleted from the namespace or the system storage capacity is increased. • The request is trying to store an object that is larger than two TB. HCP cannot store objects larger than two TB. • The request is trying to store custom metadata that is larger than one GB. HCP cannot store custom metadata larger than one GB.
414	Request URI too long	All	The portion of the URL following `fcfs_data` or `fcfs_metadata` is longer than 4,095 bytes.
415	Unsupported Media Type	PUT	The request has a Content-Encoding header with a value other than gzip.
416	Requested range not satisfiable	GET with a Range header	One of: • The specified start position is greater than the size of the requested data. • The size of the specified range is zero.
500	Internal server error	All	An internal error occurred. Try the request again, gradually increasing the delay between each successive attempt. If this error happens repeatedly, please contact your namespace administrator.

(Continued)

Code	Meaning	Methods	Description
503	Service Unavailable	All	One of: • For a request by cryptographic hash value, the cryptographic hash algorithm specified in the request is not the one the namespace is using. • For a request by cryptographic hash value, HCP cannot process the hash value because no search facility is selected for use with the Search Console. • For a request by cryptographic hash value, the request URL specifies a namespace other than the default namespace. • For a **GET** request to retrieve object data or custom metadata, the request specifies the **nowait** query parameter, and HCP determined that the request would have taken a significant amount of time to return the object data or custom metadata. • HCP is temporarily unable to handle the request, probably due to system overload, maintenance, or upgrade. • For a **GET** or **HEAD** request, HCP tried to read the object from another system in the replication topology but could not. For the last three cases, try the request again, gradually increasing the delay between each successive attempt. If the error persists, see your namespace administrator. If more information about the error is available, the response headers include the HCP-specific X-ArcErrorMessage.

HCP-specific HTTP response headers

HTTP responses can include one or more HCP-specific headers that provide information relevant to the request. The table below describes the HCP-specific HTTP response headers. It does not include deprecated headers.

Header	Methods	Description
X-ArcAvailableCapacity	HEAD to check storage capacity and software version	The amount of storage space currently available for storing additional objects, in bytes. The header has this format: X-ArcAvailableCapacity: *available-bytes* *available-bytes* is the total space available for all data, including object data, metadata, any redundant data required by the DPL, and the metadata query engine index.
X-ArcClusterTime	DELETE GET HEAD PUT	The time at which HCP sent the response to the request, in seconds since January 1, 1970, at 00:00:00.
X-ArcContentLength	GET with compressed transmission	The length, before compression, of the returned data.
X-ArcCustomMetadata ContentType	GET of object data and custom metadata	Always text/xml.
X-ArcCustomMetadata First	GET of object data and custom metadata	One of: • **true** if the custom metadata precedes the object data • **false** if the object data precedes the custom metadata
X-ArcCustomMetadata Hash	PUT that stores object data and custom metadata together	The cryptographic hash algorithm HCP uses and the cryptographic hash value of the stored custom metadata, in this format: X-ArcCustomMetadataHash: *hash-algorithm hash-value* You can use the returned hash value to verify that the stored custom metadata is the same as the metadata you sent. To do so, compare this value with a hash value that you generate from the original custom metadata.

(Continued)

Header	Methods	Description
X-ArcDataContentType	GET of object data and custom metadata	The Internet media type of the object, such as text/plain or image/jpg.
X-ArcErrorMessage	All	Detailed information about the cause of an error. This header is returned only if a request results in a 400, 403, or 503 error code and HCP has specific information about the cause.
X-ArcHash	PUT of object data or custom metadata	The cryptographic hash algorithm HCP uses and the cryptographic hash value of the stored object or metafile, in this format: X-ArcHash: *hash-algorithm hash-value* If the request stored object data and custom metadata together, this value is the hash of the object data only. You can use the returned hash value to verify that the stored data is the same as the data you sent. To do so, compare this value with a hash value that you generate from the original data.
X-ArcPermissionsUidGid	GET HEAD of object, directory, or metafile	The POSIX permissions (mode), owner ID, and group ID for the object, directory, or metafile in the format: X-ArcPermissionsUidGid: mode=*mode*; uid=*user-id*; gid=*group-id*
X-ArcServicedBySystem	All methods except HEAD to check storage capacity and software version	The domain name of the HCP system responding to the request. If the target HCP system participates in replication and is unable to respond to the request, this value may identify another system in the replication topology.
X-ArcSoftwareVersion	HEAD to check storage capacity and software version	The version number of the HCP software.
X-ArcSize	GET HEAD	The size of the object, directory, or metafile, in bytes. For directories, the value of this header is always -1.
X-ArcTimes	GET HEAD of object, directory, or metafile	The POSIX **ctime**, **mtime**, and **atime** values for the object, directory, or metafile, in the format: X-ArcTimes: ctime=*ctime*; mtime=*mtime*; atime=*atime*

(Continued)

Header	Methods	Description
X-ArcTotalCapacity	HEAD to check storage capacity and software version	The total amount of storage space available to the namespace, in bytes. The header has this format: X-ArcTotalCapacity: *total-bytes* *total-bytes* is the total space available for all data stored in the namespace, including object data, metadata, any redundant data required by the DPL, and the metadata query engine index. The value includes both used and unused space.

B

Java classes for examples

This appendix contains the implementation of these Java classes that are used in examples in this book:

- **GZIPCompressedInputStream** — This class is used by the WriteToHCP method of the HTTPCompression class. This method is defined in "Example 3: Sending object data in compressed format (Java®)" on page 83.

- **WholeIOInputStream** — This class is used by the WholeWriteToHCP method of the WholeIO class. This method is defined in "Example 5: Storing object data with custom metadata (Java)" on page 85.

- **WholeIOOutputStream** — This class is used by the WholeReadFromHCP method of the WholeIO class. This method is defined in "Example 6: Retrieving object data and custom metadata together (Java)" on page 101.

GZIPCompressedInputStream class

```
package com.hds.hcp.examples;

import java.io.IOException;
import java.io.InputStream;
import java.util.zip.CRC32;
import java.util.zip.Deflater;
import java.util.zip.DeflaterInputStream;

public class GZIPCompressedInputStream extends DeflaterInputStream {

  /**
   * This static class is used to hijack the InputStream
   * read(b, off, len) function to be able to compute the CRC32
   * checksum of the content as it is read.
   */
  static private class CRCWrappedInputStream extends InputStream {
    private InputStream inputStream;

    /**
     * CRC32 of uncompressed data.
     */
    protected CRC32 crc = new CRC32();

    /**
     * Construct the object with the InputStream provided.
     * @param pInputStream - Any class derived from InputStream class.
     */
    public CRCWrappedInputStream(InputStream pInputStream) {
      inputStream = pInputStream;

      crc.reset(); // Reset the CRC value.
    }

    /**
     * Methods in this group are the InputStream equivalent methods
     * that just call the method on the InputStream provided during
     * construction.
     */
    public int available() throws IOException
      { return inputStream.available(); };
    public void close() throws IOException { inputStream.close(); };
    public void mark(int readlimit) { inputStream.mark(readlimit); };
    public boolean markSupported()
      { return inputStream.markSupported(); };
    public int read() throws IOException { return inputStream.read(); };
    public int read(byte[] b) throws IOException
      { return inputStream.read(b); };
    public void reset() throws IOException { inputStream.reset(); };
    public long skip(long n) throws IOException
      { return inputStream.skip(n); };
```

```
  /*
   * This function intercepts all read requests in order to
   * calculate the CRC value that is stored in this object.
   */
  public int read(byte b[], int off, int len) throws IOException {
    // Do the actual read from the input stream.
    int retval = inputStream.read(b, off, len);

    // If we successfully read something, compute the CRC value
    // of it.
    if (0 <= retval) {
      crc.update(b, off, retval);
    }

    // All done with the intercept.  Return the value.
    return retval;
  };

  /*
   * Function to retrieve the CRC value computed thus far while the
   * stream was processed.
   */
  public long getCRCValue() { return crc.getValue(); };
} // End class CRCWrappedInputStream.

/**
 * Create a new input stream with the default buffer size of
 * 512 bytes.
 * @param pInputStream - InputStream to read content for
 *   compression.
 * @throws IOException if an I/O error has occurred.
 */
public GZIPCompressedInputStream(InputStream pInputStream)
  throws IOException {
    this(pInputStream, 512);
}

/**
 * Create a new input stream with the specified buffer size.
 * @param pInputStream - InputStream to read content for
 *   compression.
 * @param size - The output buffer size.
 * @exception - IOException if an I/O error has occurred.
 */
public GZIPCompressedInputStream(InputStream pInputStream, int size)
  throws IOException {
    super(new CRCWrappedInputStream(pInputStream),
      new Deflater(Deflater.DEFAULT_COMPRESSION, true), size);

    mCRCInputStream = (CRCWrappedInputStream) super.in;
}

// Indicator for if EOF has been reached for this stream.
```

```
private boolean mReachedEOF = false;

// Holder for the hijacked InputStream that computes the
// CRC32 value.
private CRCWrappedInputStream mCRCInputStream;

/*
 * GZIP header structure and positional variable.
 */
private final static int GZIP_MAGIC = 0x8b1f;

private final static byte[] mHeader = {
    (byte) GZIP_MAGIC, // Magic number (short)
    (byte)(GZIP_MAGIC >> 8), // Magic number (short)
    Deflater.DEFLATED, // Compression method (CM)
    0, // Flags (FLG)
    0, // Modification time MTIME (int)
    0, // Modification time MTIME (int)
    0, // Modification time MTIME (int)
    0, // Modification time MTIME (int)
    0, // Extra flags (XFLG)
    0  // Operating system (OS)  FYI.  UNIX/Linux OS is 3
};

private int mHeaderPos = 0;  // Keeps track of how much of the
                             // header has already been read.

/*
 * GZIP trailer structure and positional indicator.
 *
 * Trailer consists of 2 integers: CRC32 value and original file
 * size.
 */
private final static int TRAILER_SIZE = 8;
private byte mTrailer[] = null;
private int mTrailerPos = 0;

/***
 * Overridden functions against the DeflatorInputStream.
 */

/*
 * Function to indicate whether there is any content available to
 * read. It is overridden because there are the GZIP header and
 * trailer to think about.
 */
public int available() throws IOException {
  return (mReachedEOF ? 0 : 1);
}

/*
 * This read function is the meat of the class. It handles passing
 * back the GZIP header, GZIP content, and GZIP trailer in that
```

```
 * order to the caller.
 */
public int read(byte[] outBuffer, int offset, int maxLength)
  throws IOException, IndexOutOfBoundsException {

  int retval = 0; // Contains the number of bytes read into
                  // outBuffer and will be the return value of
                  // the function.
  int bIndex = offset;  // Used as current index into outBuffer.
  int dataBytesCount = 0; // Used to indicate how many data bytes
                          // are in the outBuffer array.

  // Make sure we have a buffer.
  if (null == outBuffer) {
    throw new NullPointerException("Null buffer for read");
  }

  // Make sure offset is valid.
  if (0 > offset || offset >= outBuffer.length)
  {
    throw new IndexOutOfBoundsException(
      "Invalid offset parameter value passed into function");
  }

  // Make sure the maxLength is valid.
  if (0 > maxLength || outBuffer.length - offset < maxLength)
    throw new IndexOutOfBoundsException(
      "Invalid maxLength parameter value passed into function");

  // Asked for nothing; you get nothing.
  if (0 == maxLength)
    return retval;

  /**
   * Put any GZIP header in the buffer if we haven't already returned
   * it from previous calls.
   */
  if (mHeaderPos < mHeader.length)
  {
    // Get how much will fit.
    retval = Math.min(mHeader.length - mHeaderPos, maxLength);

    // Put it there.
    for (int i = retval; i > 0; i--)
    {
      outBuffer[bIndex++] = mHeader[mHeaderPos++];
    }

    // Return the number of bytes copied if we exhausted the
    // maxLength specified.
    // NOTE: Should never be >, but...
    if (retval >= maxLength) {
      return retval;
```

```
  }
}

/**
 * At this point, the header has all been read or put into the
 * buffer.
 *
 * Time to add some GZIP compressed data, if there is still some
 * left.
 */
if (0 != super.available()) {

  // Get some data bytes from the DeflaterInputStream.
  dataBytesCount = super.read(outBuffer, offset+retval,
                              maxLength-retval);

  // As long as we didn't get EOF (-1), update the buffer index and
  // retval.
  if (0 <= dataBytesCount) {
    bIndex += dataBytesCount;
    retval += dataBytesCount;
  }

  // Return the number of bytes copied during this call if we
  // exhausted the maxLength requested.
  // NOTE: Should never be >, but...
  if (retval == maxLength) {
    return retval;
  }

  // If we got here, we should have read all that can be read from
  // the input stream, so make sure the input stream is at EOF just
  // in case someone tries to read it outside this class.
  byte[] junk = new byte[1];
  if (-1 != super.read(junk, 0, junk.length)) {
    // Should never happen, but...
    throw new IOException(
   "Unexpected content read from input stream when EOF expected");
  }
}

/**
 *  Got this far; time to write out the GZIP trailer.
 */

// Have we already set up the GZIP trailer in a previous
// invocation?
if (null == mTrailer) {
  // Time to prepare the trailer.
  mTrailer = new byte[TRAILER_SIZE];

  // Put the content in it.
  writeTrailer(mTrailer, 0);
```

```
  }

  // If there are still GZIP trailer bytes to be returned to the
  // caller, do as much as will fit in the outBuffer.
  if (mTrailerPos < mTrailer.length) {

    // Get the number of bytes that will fit in the outBuffer.
    int trailerSize = Math.min(mTrailer.length - mTrailerPos,
                               maxLength - bIndex);

    // Move them in.
    for (int i = trailerSize; i > 0; i--)
    {
      outBuffer[bIndex++] = mTrailer[mTrailerPos++];
    }

    // Return the total number of bytes written during this call.
    return retval + trailerSize;
  }

  /**
   * If we got this far, we have already been asked to read
   * all content that is available.
   *
   * So we are at EOF.
   */
  mReachedEOF = true;
  return -1;
}

/***
 * Helper functions to construct the trailer.
 */

/*
 * Write GZIP member trailer to a byte array, starting at a given
 * offset.
 */
private void writeTrailer(byte[] buf, int offset) throws IOException
{
  writeInt((int)mCRCInputStream.getCRCValue(), buf, offset);
                                     // CRC32 of uncompr. data
  writeInt(def.getTotalIn(), buf, offset + 4);
                                     // Number of uncompr. bytes
}

/*
 * Write integer in Intel byte order to a byte array, starting at
 * a given offset.
 */
private void writeInt(int i, byte[] buf, int offset)
  throws IOException {
    writeShort(i & 0xffff, buf, offset);
```

```
            writeShort((i >> 16) & 0xffff, buf, offset + 2);
        }

        /*
         * Write short integer in Intel byte order to a byte array,
         * starting at a given offset
         */
        private void writeShort(int s, byte[] buf, int offset)
          throws IOException {
            buf[offset] = (byte)(s & 0xff);
            buf[offset + 1] = (byte)((s >> 8) & 0xff);
        }
    }
```

WholeIOInputStream class

```
package com.hds.hcp.examples;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

/**
 * This class defines an InputStream that is composed of both a data
 * file and a custom metadata file.
 *
 * The class is used to provide a single stream of data and custom
 * metadata to be transmitted over HTTP for type=whole-object PUT
 * operations.
 */
public class WholeIOInputStream extends InputStream {

  /*
   * Constructor.  Passed in an InputStream for the data file and the
   * custom metadata file.
   */
  WholeIOInputStream(
      InputStream inDataFile, InputStream inCustomMetadataFile) {
    mDataFile = inDataFile;
    mCustomMetadataFile = inCustomMetadataFile;

    bFinishedDataFile = false;
  }

  // Private member variables.
  private Boolean bFinishedDataFile;  // Indicates when all data file
                                      // content has been read.
  private InputStream mDataFile, mCustomMetadataFile;

  /*
   * Base InputStream read function that reads from either the data
   * file or custom metadata file, depending on how much has been read so
   * far.
```

```
   */
  public int read() throws IOException {
    int retval = 0;  // Assume nothing read.

    // Do we still need to read from the data file?
    if (! bFinishedDataFile ) {
      // Read from the data file.
      retval = mDataFile.read();

      // If reached the end of the stream, indicate it is time to read
      // from the custom metadata file.
      if (-1 == retval) {
        bFinishedDataFile = true;
      }
    }

    // This should not be coded as an "else" because it may need to be
    // run after the data file has reached EOF.
    if ( bFinishedDataFile ) {
      // Read from the custom metadata file.
      retval = mCustomMetadataFile.read();
    }

    return retval;
  }
}
```

WholeIOOutputStream class

```
package com.hds.hcp.examples;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

/**
 * This class defines an OutputStream that will create both the data
 * file and the custom metadata file for an object. The copy() method
 * is used to read an InputStream and create the two output files based
 * on the indicated size of the data file portion of the stream.
 *
 * The class is used to split and create content retrieved over HTTP as
 * a single stream for type=whole-object GET operations.
 */
public class WholeIOOutputStream extends OutputStream {

  // Constructor.  Passed output streams for the data file and the
  // custom metadata file. Allows specification of whether the custom
  // metadata comes before the data.
  WholeIOOutputStream(OutputStream inDataFile,
      OutputStream inCustomMetadataFile,
      Boolean inCustomMetadataFirst) {
```

```
    bCustomMetadataFirst = inCustomMetadataFirst;

    // Set up first and second file output streams based on whether
    // custom metadata is first in the stream.
    if (bCustomMetadataFirst) {
      mFirstFile = inCustomMetadataFile;
      mSecondFile = inDataFile;
    } else {
      mFirstFile = inDataFile;
      mSecondFile = inCustomMetadataFile;
    }

    bFinishedFirstPart = false;
  }

  // Member variables.
  private Boolean bFinishedFirstPart;
  private Boolean bCustomMetadataFirst;
  private OutputStream mFirstFile, mSecondFile;

  /**
   * This routine copies content in an InputStream to this
   * output stream. The first inDataSize number of bytes are written
   * to the data file output stream.
   *
   * @param inStream - InputStream to copy content from.
   * @param inFirstPartSize - number of bytes of inStream that should
   * be written to the first output stream.
   * @throws IOException
   */
  public void copy(InputStream inStream, Integer inFirstPartSize)
    throws IOException {
    int streamPos = 0;

    int readValue = 0;
    // Keep reading bytes until EOF has been reached.
    while (-1 != (readValue = inStream.read())) {
      // Have we read all the bytes for the data file?
      if (streamPos == inFirstPartSize)
      {
        // Yes.
        bFinishedFirstPart = true;
      }

      // Write the bytes read.
      write(readValue);
      streamPos++;
    }
  }

  /**
   * This is the core write function for the InputStream implementation.
   * It writes to either the data file stream or the custom metadata
```

```
     * file stream.
     */
    public void write(int b) throws IOException {
      // Write to first or second file depending on where we are in the
      // stream.
      if (! bFinishedFirstPart ) {
        mFirstFile.write(b);
      } else {
        mSecondFile.write(b);
      }
    }

    /**
     * flush() method to flush all files involved.
     */
    public void flush() throws IOException {
      mFirstFile.flush();
      mSecondFile.flush();
      super.flush();
    }

    /**
     * close() method to first close the data file and custom metadata
     * file. Then close itself.
     */
    public void close() throws IOException {
      mFirstFile.close();
      mSecondFile.close();
      super.close();
    }
}
```

Glossary

A

access protocol

See namespace access protocol.

Active Directory (AD)

A Microsoft product that, among other features, provides user authentication services.

AD

See Active Directory (AD).

anonymous access

A method of access to a namespace wherein the user or application gains access without presenting any credentials. *See also* authenticated access.

appendable object

An object to which data can be added after it has been successfully stored. Appending data to an object does not modify the original fixed-content data, nor does it create a new version of the object. Once the new data is added to the object, that data also cannot be modified.

Appendable objects are supported only with the CIFS and NFS protocols.

atime

In POSIX file systems, metadata that specifies the date and time a file was last accessed. In HCP, POSIX metadata that initially specifies the date and time at which an object was ingested. HCP does not automatically change the **atime** value when the object is accessed.

Users and applications can change this metadata, thereby causing it to no longer reflect the actual storage time. Additionally, HCP can be configured to synchronize **atime** values with retention settings.

authenticated access

A method of access to a namespace wherein the user or application presents credentials to gain access. *See also* anonymous access.

authentication

See user authentication.

C

capacity

The total amount of primary storage space in HCP, excluding the space required for system overhead and the operating system. This is the amount of space available for all data to be stored in primary running storage and primary spindown storage, including the fixed-content data, metadata, any redundant data required to satisfy service plans, and the metadata query engine index.

CIFS

Common Internet File System. One of the namespace access protocols supported by HCP. CIFS lets Windows clients access files on a remote computer as if the files were part of the local file system.

compliance mode

The retention mode in which objects under retention cannot be deleted through any mechanism. This is the more restrictive retention mode.

cryptographic hash value

A system-generated metadata value calculated by a cryptographic hash algorithm from object data. This value is used to verify that the content of an object has not changed.

ctime

POSIX metadata that specifies the date and time of the last change to the metadata for an object. For a directory, this is the time of the last change to the metadata for any object in the directory.

custom metadata

User-supplied information about an HCP object. Users and applications can use custom metadata to understand and repurpose object content.

D

Data Migrator

See HCP Data Migrator (HCP-DM).

data protection level (DPL)

The number of copies of the data for an object HCP must maintain in the repository. The DPL for an object is determined by the service plan that applies to the namespace containing the object.

dead properties

For WebDAV only, arbitrary name/value pairs that the server stores but does not use or modify in any way.

default namespace

A namespace that supports only anonymous access through the HTTP protocol. An HCP system can have at most one default namespace. The default namespace is used mostly with applications that existed before release 3.0 of HCP.

default tenant

The tenant that manages the default namespace.

disposition

The automatic deletion of an expired object by HCP.

DNS

See domain name system (DNS).

domain

A group of computers and devices on a network that are administered as a unit.

domain name system (DNS)

A network service that resolves domain names into IP addresses for client access.

DPL

See data protection level (DPL).

E

See HCP S Series Node.

enterprise mode

The retention mode in which these operations are allowed:

- Privileged delete
- Changing the retention class of an object to one with a shorter duration
- Reducing retention class duration
- Deleting retention classes

This is the less restrictive retention mode.

expired object

An object that is no longer under retention.

F

fixed-content data

A digital asset ingested into HCP and preserved in its original form as the core part of an object. Once stored, fixed-content data cannot be modified.

G

GID

POSIX group identifier.

H

hash value

See cryptographic hash value.

HCP

See Hitachi Content Platform (HCP).

HCP Data Migrator (HCP-DM)

An HCP utility that can transfer data from one location to another, delete data from a location, and change object metadata in a namespace. Each location can be a local file system, an HCP namespace, a default namespace, or an HCAP 2.*x* archive.

HCP-DM

See HCP Data Migrator (HCP-DM).

HCP-FS

See HCP file system (HCP-FS).

HCP file system (HCP-FS)

The HCP runtime component that represents each object in a namespace as a set of files. One of these files contains the object data. The others contain the object metadata.

HCP metadata query API

See metadata query API.

HCP namespace

A namespace that supports user authentication for data access through the HTTP, HS3, and CIFS protocols. HCP namespaces also support storage usage quotas, access control lists, and versioning. An HCP system can have multiple HCP namespaces.

HCP node

See node.

HDDS

See Hitachi Data Discovery Suite (HDDS).

HDDS search facility

One of the search facilities available for use with the HCP Search Console. This facility interacts with Hitachi Data Discovery Suite.

Hitachi Content Platform (HCP)

A distributed object-based storage system designed to support large, growing repositories of fixed-content data. HCP provides a single scalable environment that can be used for archiving, business continuity, content depots, disaster recovery, e-discovery, and other services. With its support for multitenancy, HCP securely segregates data among various constituents in a shared infrastructure. Clients can use a variety of industry-standard protocols and various HCP-specific interfaces to access and manipulate objects in an HCP repository.

Hitachi Data Discovery Suite (HDDS)

A Hitachi product that enables federated searches across multiple HCP systems and other supported systems.

hold

A condition that prevents an object from being deleted by any means and from having its metadata modified, regardless of its retention setting, until it is explicitly released.

HS3 API

One of the namespace access protocols supported by HCP. HS3 is a RESTful, HTTP-based API that is compatible with Amazon S3.

HTTP

HyperText Transfer Protocol. One of the namespace access protocols supported by HCP.

HTTPS

HTTP with SSL security. *See* HTTP *and* SSL.

I

index

An index of the objects in namespaces that is used to support search operations. Each of the two search facilities, the metadata query engine and the HDDS search facility, creates and maintains its own separate index.

index setting

The property of an object that determines whether the metadata query engine indexes the custom metadata associated with the object.

M

metadata

System-generated and user-supplied information about an object. Metadata is stored as an integral part of the object it describes, thereby making the object self-describing.

metadata query API

A RESTful HTTP interface that lets you search HCP for objects that meet specified metadata-based or operation-based criteria. With this API, you can search not only for objects currently in the repository but also for information about objects that are no longer in the repository.

metadata query engine

One of the search facilities available for use with HCP. The metadata query engine works internally to perform searches and return results either through the metadata query API or to the HCP Metadata Query Engine Console (also known as the HCP Search Console).

Metadata Query Engine Console

The web application that provides interactive access to the HCP search functionality provided by the metadata query engine.

metadirectory

A directory in the `fcfs_metadata` directory hierarchy. Metadirectories contain metafiles.

metafile

A file containing metadata about an object. Metafiles enable file-system access to portions of the object metadata.

mtime

POSIX metadata that specifies the date and time of the last change to the object data. Because you cannot change the content of an object, **mtime** is, by default, the date and time at which the object was added to a namespace. Users and applications can change this metadata, thereby causing it to no longer reflect the actual storage time.

N

namespace

A logical partition of the objects stored in an HCP system. A namespace consists of a grouping of objects such that the objects in one namespace are not visible in any other namespace. Namespaces are configured independently of each other and, therefore, can have different properties.

namespace access protocol

A protocol that can be used to transfer data to and from namespaces in an HCP system. HCP supports the HTTP, HS3, WebDAV, CIFS, NFS, and SMTP protocols for access to HCP namespaces and the default namespace. HCP also supports the NDMP protocol for access to the default namespace.

NDMP

Network Data Management Protocol. The namespace access protocol HCP supports for backing up and restoring objects in the default namespace.

NFS

Network File System. One of the namespace access protocols supported by HCP. NFS lets clients access files on a remote computer as if the files were part of the local file system.

node

A server running HCP software and networked with other such servers to form an HCP system.

O

object

An exact digital representation of data as it existed before it was ingested into HCP, together with the system and custom metadata that describes that data. An object is handled as a single unit by all transactions and internal processes, including shredding, indexing, and replication.

object-based query

In the metadata query API, a query that searches for objects based on object metadata. This includes both system metadata and the content of custom metadata. The query criteria can also include the object location (that is, the namespace and/or directory that contains the object).

Object-based queries search only for objects that currently exist in the repository.

operation-based query

In the metadata query API, a query that searches not only for objects currently in the repository but also for information about objects that have been deleted by a user or application or deleted through disposition.

Criteria for operation-based queries can include object status (for example, created or deleted), change time, index setting, and location (that is, the namespace and/or directory that contains the object).

P

permission

In POSIX permissions, the ability granted to the owner, the members of a group, or other users to access an object, directory, or symbolic link. A POSIX permission can be read, write, or execute.

POSIX

Portable Operating System Interface for UNIX. A set of standards that define an application programming interface (API) for software designed to run under heterogeneous operating systems. HCP-FS is a POSIX-compliant file system, with minor variations.

privileged delete

A delete operation that works on an object regardless of whether the object is under retention, except if the object is on hold. This operation is available only to users and applications with explicit permission to perform it.

protocol

See namespace access protocol.

Q

query

A request submitted to HCP to return metadata for objects that satisfy a specified set of criteria. Also, to submit such a request.

query API

See metadata query API.

R

replication

The process of keeping selected HCP tenants and namespaces and selected default-namespace directories in two HCP systems in sync with each other. This entails copying object creations, deletions, and metadata changes from each system to the other or from one system to the other.

repository

The aggregate of the namespaces defined for an HCP system.

REST

Representational State Transfer. A software architectural style that defines a set of rules (called constraints) for client/server communication. In a REST architecture:

- Resources (where a resource can be any coherent and meaningful concept) must be uniquely addressable.
- Representations of resources (for example, in XML format) are transferred between clients and servers. Each representation communicates the current or intended state of a resource.

- Clients communicate with servers through a uniform interface (that is, a set of methods that resources respond to) such as HTTP.

retention class

A named retention setting. The value of a retention class can be a duration, Deletion Allowed, Deletion Prohibited, or Initial Unspecified.

retention hold

See hold.

retention mode

A namespace property that affects which operations are allowed on objects under retention. A namespace can be in either of two retention modes: compliance or enterprise.

retention period

The period of time during which an object cannot be deleted (except by means of a privileged delete).

retention setting

The property that determines the retention period for an object.

S

Search Console

The web application that provides interactive access to HCP search functionality. When the Search Console uses the HCP metadata query engine for search functionality, it is called the Metadata Query Engine Console.

search facility

An interface between the HCP Search Console and the search functionality provided by the metadata query engine or HDDS. Only one search facility can be selected for use with the Search Console at any given time.

shred setting

The property that determines whether an object will be shredded or simply removed when it's deleted from HCP.

shredding

The process of deleting an object and overwriting the locations where all its copies were stored in such a way that none of its data or metadata can be reconstructed. Also called **secure deletion**.

SMTP

Simple Mail Transfer Protocol. The namespace access protocol HCP uses to receive and store email data directly from email servers.

SSL

Secure Sockets Layer. A key-based Internet protocol for transmitting documents through an encrypted link.

SSL server certificate

A file containing cryptographic keys and signatures. When used with the HTTP protocol, an SSL server certificate helps verify that the web site holding the certificate is authentic. An SSL server certificate also helps protect data sent to or from that site.

system metadata

System-managed properties that describe the content of an object. System metadata includes policies, such as retention and data protection level, that influence how transactions and internal processes affect the object.

T

tenant

An administrative entity created for the purpose of owning and managing namespaces. Tenants typically correspond to customers or business units.

U

UID

POSIX user ID.

Unix

Any UNIX-like operating system (such as UNIX itself or Linux).

user authentication

The process of checking that the combination of a specified username and password is valid when a user tries to access a namespace.

W

WebDAV

Web-based Distributed Authoring and Versioning. One of the namespace access protocols supported by HCP. WebDAV is an extension of HTTP.

WORM

Write once, read many. A data storage property that protects the stored data from being modified or overwritten.

X

XML

Extensible Markup Language. A standard for describing data content using structural tags called elements.

Index

Symbols

.directory-metadata metadirectory 19
.lost+found directory 18

Numbers

0 (retention setting) 40, 44, 46
-1 (retention setting) 40, 44, 46
-2 (retention setting) 40, 44, 46

A

Accept-Encoding header 75
access-time, WebDAV property 155, 157
adding
    See storing; creating
appendable objects
    about 3
    and atime synchronization 51, 52
    change times for 34
assigning objects to retention classes 48
atime attribute
    about 36
    changing with HTTP 127
    changing with NFS (example) 180
    overriding default values with HTTP 115, 120
    synchronization with retention setting 51–57
atime HTTP query parameter 115, 120, 127
atime synchronization
    about 51–52
    with appendable objects 51, 52
    creating empty directories 172
    example 56–57
    how it works 54–56
    with retention classes 52
    triggering for existing objects 52–53
attachments, email 190
available space, checking 141–143

B

basic authentication, WebDAV 161
browsing the namespace
    HTTP 76
    WebDAV 152–153
byte range, retrieving with HTTP 89–92

C

cadaver 147
capacity, checking storage 141–143
case sensitivity, CIFS 170–171
change time
    appendable objects 35
    HCP metadata 34–35
change-time, WebDAV property 155, 157
changing
    atime attribute with NFS (example) 180
    HCP-specific metadata with HTTP 123–125
    index settings 59–60
    ownership 39
    permissions 39
    POSIX metadata with HTTP 125–130
    retention settings 45–48
    retention settings with CIFS (example) 169
    retention settings with NFS (example) 179
    shred settings 58
checking
    custom metadata existence 134–136
    directory existence 106–108
    object existence 86–89
    storage capacity and software version 141–143
choosing namespace access protocols 192–193
CIFS
    about 5, 192, 193
    case sensitivity 170–171
    changing retention settings (example) 169
    examples 168–170
    failed write operations 174

lazy close 172
mapping the namespace 168
multithreading 175, 197
namespace access 168
open objects 173
out-of-order writes 173
ownership of new objects 38–39
permission translations 171
permissions for new objects 38–39
retrieving deletable objects (example) 170
retrieving objects (example) 169–170
return codes 176
storing objects (example) 169
storing zero-sized files 172
supported operations 11–12
usage considerations 170–175
Windows temporary files 174–175
client timeouts, WebDAV 162
collections, WebDAV 3, 147
See also directories; metadirectories
collision handling
See replication collisions
connection failure handling with HTTP 145–146
Content-Encoding header 75
core-metadata.xml metafile
for directories 23
for objects 29
created.txt metafile
for directories 22
for objects 26
creating
See also storing
empty directories in Windows 172
empty directories with HTTP 105–106
symbolic links with NFS (example) 180
creation-time, WebDAV property 155, 157
cryptographic hash algorithm 210
cryptographic hash values
about 35
finding 74
metafile for 27
namespace access with 73–74
ctime attribute
about 36
and object change time 35
cURL 69
custom metadata
See also custom-metadata.xml metafile
about 2, 60
checking existence with HTTP 134–136
collisions 66–67
deleting with HTTP 139–141
files 60
metafile for 21, 30
with objects under retention 61
retrieving in compressed format 137
retrieving together with object data 90, 101–102
retrieving with HTTP 136–139
searching by 60
sending in compressed format 132
storing in compressed format 75
storing together with object data 78, 84–86
storing with HTTP 131–134
custom-metadata.xml metafile
See also custom metadata
about 30, 60
dead properties 160

D

data access 5–8
data chunking with HTTP 146
Data Migrator 8
data protection level 35
date and time, specifying for retention setting 48–49
dead properties 160
default namespace
See also namespaces
about 3–4
access 5–8
accessing by DNS name 194
accessing by IP address 194–195
browsing with HTTP 76
browsing with WebDAV 152–153
checking storage capacity and software version 141–143
CIFS access 168
HTTP access 70–72
NFS access 178
sending email to 188
SMTP access 188
WebDAV access 149–152
default tenant 4
delayed retrieval
custom metadata 137
objects 90
deletable objects
retrieving with CIFS (example) 170
retrieving with NFS (example) 181
deleting
custom metadata with HTTP 139–141
directories with HTTP 112–113, 197
objects under repair 197
objects under retention 40
objects with HTTP 102–105

open objects with NFS 184
symbolic links 12
Deletion Allowed 40, 44, 46
Deletion Prohibited 40, 44, 46
directories
See also metadirectories
changing HCP-specific metadata with HTTP 123–125
changing POSIX metadata with HTTP 125–130
changing retention settings 46
checking existence with HTTP 106–108
creating with HTTP 105–106
deleting 197
deleting with HTTP 112–113
email 189–190
listing contents with HTTP 108–112
metadirectories for 18–20
metafiles for 21–25
overriding default metadata with HTTP 119–121
permissions 37
renaming empty 52, 172
retrieving HCP-specific metadata with HTTP 121–123
retrieving POSIX metadata with HTTP 123
structuring 195–196
WebDAV properties for 157–158
directory_permissions, HTTP query parameter 115, 119
DNS names, namespace access by 194
DPL 35
dpl, WebDAV property 155
dpl.txt metafile
for directories 25
for objects 27
du with large directory trees 184

E

email
object naming 188–190
sending to the namespace 188
empty directories
creating in Windows 172
renaming 52, 172
error codes
See return codes
examples
changing atime attribute with NFS 180
changing HCP-specific metadata with HTTP 124–125
changing POSIX metadata with HTTP 128–130
changing retention settings with CIFS 169
changing retention settings with NFS 179
checking custom metadata existence with HTTP 135–136
checking object existence with HTTP 88–89
checking storage capacity with HTTP 142–143
creating directories with HTTP 106
creating symbolic links with NFS 180
deleting custom metadata with HTTP 140–141
deleting directories with HTTP 113
deleting objects with HTTP 104–105
listing directory contents with HTTP 110–112
retrieving custom metadata with HTTP 138–139
retrieving deletable objects with CIFS 170
retrieving deletable objects with NFS 181
retrieving HCP-specific metadata with HTTP 122–123
retrieving object data and custom metadata together 101–102
retrieving objects in compressed format 98–100
retrieving objects with CIFS 169–170
retrieving objects with HTTP 96–102
retrieving objects with NFS 180–181
sending object data in compressed format 82–84
specifying metadata on directory creation with HTTP 120–121
specifying metadata on object creation with HTTP 117–119
storing custom metadata with HTTP 133–134
storing object data and custom metadata together 84–86
storing objects with CIFS 169
storing objects with HTTP 81–86
storing objects with NFS 179
WebDAV PROPFIND 159–160
WebDAV PROPPATCH 158–159
expired metadirectory 19, 41, 170, 181

F

failed nodes, NFS mounts on 184
failed write operations
CIFS 174
HTTP 145
NFS 183–184
WebDAV 163
fcfs_data directory 16, 18–19
fcfs_metadata metadirectory 16
file system

See HCP-FS
file_permissions, HTTP query parameter 115
files
See objects; metafiles
fixed-content data 2

G

gid
See also group IDs of owning groups
HTTP query parameter 115, 119, 126
WebDAV property 155, 157
group IDs of owning groups
See also gid
about 36
changing with HTTP 126
overriding default with HTTP 115, 119
gzip
compressing custom metadata data for retrieval 137
compressing custom metadata for transmission 132
compressing data for transmission 75
compressing object data and custom metadata for retrieval 90
compressing object data and custom metadata for transmission 78
compressing object data for retrieval 90, 98–100
compressing object data for transmission 77, 82–84
storing compressed data 75

H

hash algorithm
See cryptographic hash algorithm
hash value
See cryptographic hash value
hash.txt metafile 27
hash-scheme, WebDAV property 155
hash-value, WebDAV property 156
HCP
about 1–9
checking software version 141–143
HCP Data Migrator 8
HCP metadata query API
See metadata query API
HCP namespaces 3–4
HCP Search Console 7–8
HCP search facility
index settings with 58
HCP-DM 8
HCP-FS 4–5, 15–16
HCP-specific metadata
about 34–35
changing with HTTP 123–125
HDDS search facility 7
Hitachi Content Platform 1–9
Hold 48
holding objects 41–42
HTTP
See also individual HTTP methods
about 5, 192–193
browsing the namespace 76
changing HCP-specific metadata 123–125
checking existence of custom metadata 134–136
checking existence of directories 106–108
checking existence of objects 86–89
compliance level 69
connection failure handling 145–146
creating empty directories 105–106
data chunking 146
delayed retrievals 90, 137
deleting custom metadata 139–141
deleting directories 112–113
deleting objects 102–105
failed write operations 145
HCP-specific response headers 209–211
listing directory contents 108–112
methods 200–204
multithreading 146, 197
namespace access with cryptographic hash values 73–74
naming objects 72
open objects 144–145
ownership of new objects 38
permission checking 143–144
permissions for new objects 38
persistent connections 144
query parameters for metadata 114–115
retrieving custom metadata 136–139
retrieving HCP-specific metadata 121–123
retrieving objects 89–102
retrieving POSIX metadata 123
return codes 204–208
specifying metadata on directory creation 119–121
specifying metadata on object creation 114–119
storing objects 77–86
storing zero-sized files 144
supported operations 11–12
URLs for namespace access 70–73
usage considerations 143–146
HTTP CHMOD

See also HTTP
example 128–129
modifying permissions 126
reference 200
return codes 127–128
HTTP CHOWN
See also HTTP
example 129–130
modifying object owner and group 126
reference 200
return codes 127–128
HTTP DELETE
See also HTTP
deleting custom metadata 139–141
deleting directories 112–113
deleting objects 102–105
examples 104–105, 113, 140–141
reference 200
response headers 104, 112, 140
return codes 103–104, 112, 140
HTTP GET
See also HTTP
byte-range requests 89–92
connection failure 146
delayed custom metadata 137
delayed objects 90
examples 96–102, 110–112, 122–123, 138–139
listing directory contents 108–112
nowait parameter 90, 137
reference 201
response headers 94–95, 109, 122, 138
retrieving custom metadata 136–139
retrieving custom metadata in compressed format 137
retrieving data in compressed format 75
retrieving HCP-specific metadata 121–123
retrieving object data and custom metadata in compressed format 90
retrieving object data and custom metadata together 90
retrieving object data in compressed format 90
retrieving objects 89–102
retrieving POSIX metadata 123
return codes 92–93, 108, 121, 137
HTTP HEAD
See also HTTP
checking existence of custom metadata 134–136
checking existence of directories 106–108
checking existence of objects 86–89
checking storage capacity and software version 141–143
examples 88–89, 107–108, 135–136, 142–143
reference 202
response headers 88, 107, 135, 142
return codes 87, 107, 135
HTTP MKDIR
See also HTTP
creating empty directories 105–106
example 106
reference 202
response headers 106, 120
return codes 105, 120
HTTP POST 69
HTTP PUT
See also HTTP
changing HCP-specific metadata 123–125
examples 81–86, 117–119, 120–121, 124–125, 133–134
overriding default directory metadata 119–121
overriding default object metadata with HTTP 118–119
Range request header 91–92
reference 203
response headers 81, 117, 124, 133
return codes 79–80, 116, 124, 132–133
sending custom metadata in compressed format 132
sending data in compressed format 75
sending object data in compressed format 77
storing custom metadata 131–134
storing object data and custom metadata together 78
storing objects 77–86
HTTP response headers
HCP-specific 209–211
for multiple matching objects 74–75
HTTP TOUCH
See also HTTP
example 130
modifying atime and mtime 127
reference 204
return codes 127–128

I

index settings
See also index.txt metafile
about 58–59
changing 59–60
changing with HTTP 124

with HCP search facility 58
with metadata query engine 58
overriding default with HTTP 59, 115
index, HTTP query parameter 115
index.txt metafile
See also index settings
changing with HTTP 124
for directories 25
for objects 27
indexes 8
info metadirectory 19
Initial Unspecified 40, 44, 46
IP addresses, namespace access by 194–195

L

large directory trees with du 184
large objects, reading with NFS 184
lazy close
CIFS 172
NFS 181–182
libcurl 69
listing directory contents with HTTP 108–112
locking, WebDAV 162

M

Mac OS X hosts file 193
mapping the namespace with CIFS 168
metadata
See also metafiles
about 33
change time 34–35
changing HCP-specific with HTTP 123–125
changing POSIX with HTTP 125–130
custom 2
HCP specific 34–35
HTTP query parameters for 114–115
modifying 34
POSIX 35–36
retrieving HCP-specific with HTTP 121–123
retrieving POSIX with HTTP 123
specifying on directory creation with HTTP 119–121
specifying on object creation with HTTP 118–119
system 2
types 34
WebDAV properties 155–158
metadata query API 6–7
metadata query engine
about 7
index settings with 58
Metadata Query Engine Console 7
metadirectories
.directory-metadata 19
about 4
for directories 18–20
expired 19
fcfs_metadata 16
info 19
for objects 20
settings 19
metafiles
See also metadata
about 4, 21
core-metadata.xml for directories 23
core-metadata.xml for objects 29
created.txt for directories 22
created.txt for objects 26
custom-metadata.xml 30, 60
for directories 21–25
dpl.txt for directories 25
dpl.txt for objects 27
hash.txt 27
index.txt for directories 25
index.txt for objects 27
for objects 26–30
replication.txt 27
retention.txt for directories 25
retention.txt for objects 28
retention-classes.xml 24
shred.txt for directories 25
shred.txt for objects 28
tpof.txt 26
URLs for 71–72
methods
HTTP 200–204
WebDAV 148–149
mode, WebDAV property 156, 157
modifying metadata 34
mounting the namespace with NFS 178
moving objects 196
mtime attribute
about 36
changing with HTTP 127
overriding default values with HTTP 115, 120
mtime, HTTP query parameter 115, 120, 127
multiple matching objects, response headers for 74–75
multithreading
CIFS 175
general guidelines 197
HTTP 146
NFS 185
WebDAV 163–164

N

namespace access protocols
See also CIFS; HTTP; NFS; SMTP; WebDAV
about 5–6
choosing 192–193
namespaces
See also default namespace
about 3
default and HCP 3–4
operations on 10–13
replicated 9–10
naming
email objects 188–190
objects 16–17
objects using HTTP 72
objects using WebDAV 151
NDMP 5
NFS
about 5, 192, 193
changing atime (example) 180
changing retention settings (example) 179
creating symbolic links (example) 180
delete operations 184
examples 179–181
failed write operations 183–184
large directory trees 184
lazy close 181–182
mounting the namespace 178
mounts on failed nodes 184
multithreading 185, 197
namespace access 178
open objects 182–183
out-of-order writes 182
ownership of new objects 39
permissions for new objects 39
reading large objects 184
retrieving deletable objects (example) 181
retrieving objects (example) 180–181
return codes 186
storing objects (example) 179
storing zero-sized files 182
supported operations 11–12
usage considerations 181–185
nodes
about 9
namespace access by IP address 194–195
NFS mounts on failed 184
non-ASCII, nonprintable characters 72, 151
non-WORM objects 196
nowait HTTP parameter 90, 137

O

objects
about 2–3
appendable 3
assigning to retention classes 48
change time 34–35
changing HCP-specific metadata with HTTP 123–125
changing ownership 39
changing permissions 39
changing POSIX metadata with HTTP 125–130
changing retention settings 45
checking existence with HTTP 86–89
content collisions 62–63
creation date 34
deleting with HTTP 102–105
email 189–190
email attachments 190
holding 41–42
index settings 59–60
indexing 58–60
metadirectories for 20
metafiles for 26–30
moving 196
names with non-ASCII, nonprintable characters 72, 151
naming 16–17
naming with HTTP 72
naming with WebDAV 151
non-WORM 196
open, and CIFS 173
open, and HTTP 144–145
open, and NFS 182–183
open, and WebDAV 163–??
open,access using WebDAV ??–163
overriding default metadata with HTTP 114–119
ownership 35, 36
ownership for new 38–39
permissions 35, 36–37
permissions for new 38–39
renaming 196
replicated 9–10, 27
representation 4–5
retention 39–57
retrieving HCP-specific metadata with HTTP 121–123
retrieving in compressed format 90, 98–100
retrieving object data and custom metadata together 90, 101–102
retrieving POSIX metadata with HTTP 123
retrieving with CIFS (example) 169–170

retrieving with HTTP 89–102
retrieving with NFS (example) 180–181
sending in compressed format 77, 82–84
shred settings 57–58
shredding 57–58
storing 2–3
storing in compressed format 75
storing object data and custom metadata together 78, 84–86
storing with CIFS (example) 169
storing with HTTP 77–86
storing with NFS (example) 179
URLS for 71
WebDAV metadata properties for 155–157
octal permission values 38
offsets, specifying for retention setting 49–51
open objects
access using WebDAV ??–163
and CIFS 173
and HTTP 144–145
and NFS 182–183
and WebDAV 163–??
operations
prohibited 12–13
supported 11–12
out-of-order writes
CIFS 173
NFS 182
overriding
atime values with HTTP 115, 120
default directory metadata with HTTP 119–121
default index settings with HTTP 59, 115
default object metadata with HTTP 118–119
default retention settings with HTTP 41
default shred settings with HTTP 57
mtime values with HTTP 115, 120
object owner with HTTP 115, 119
owning group with HTTP 119
permissions for directories with HTTP 115, 119
permissions for objects with HTTP 115
retention settings with HTTP 115
shred settings with HTTP 115
ownership
about 36
changing 39
changing with HTTP 126
new objects 38–39
overriding default with HTTP 115, 119

P

partial objects, retrieving with HTTP 89–92
percent encoding
returned object names 72, 151
in URLs 72–73, 152
permission checking
HTTP 143–144
WebDAV 161
permissions
about 36–37
atime synchronization, effect on 53
changing 39
changing with HTTP 126
CIFS translations of 171
HTTP checking 143–144
new objects 38–39
octal values 38
overriding for directories with HTTP 115, 119
overriding for objects with HTTP 115
viewing 37–38
WebDAV checking 161
persistent connections
HTTP 144
WebDAV 161–162
POSIX metadata
See also atime attribute; ctime attribute; group IDs of owning groups; permissions; mtime attribute; user IDs of object owners
about 35–36
changing with HTTP 125–130
privileged delete 40
prohibited operations 12–13
properties, WebDAV 155–158

Q

query API
See metadata query API
quotation marks with URLs 73, 152

R

releasing objects from hold 42
renaming
empty directories 52
objects 196
replication
about 9–10
collision handling 62–??
object status 27, 35
topologies 9
WebDAV property 156
replication collisions
about 62
custom metadata 66–67

object content 62–63
system metadata 63–66
replication.txt metafile 27
replication-collision, WebDAV property 156
resources, WebDAV 3, 147
response headers
See HTTP response headers
retention
See also retention classes; retention settings; retention.txt metafile
about 39–40
hold 41–42
HTTP query parameter 115
periods 40
WebDAV property 158
retention classes
See also retention; retention settings; retention.txt metafile
about 42–43
assigning to objects 48
with atime synchronization 52
deleted 43, 45
list of 21
retention settings
See also retention; retention classes; retention.txt metafile
changing 45–48
changing with CIFS (example) 169
changing with HTTP 124
changing with NFS (example) 179
default 40–41
metafile for (directories) 25
metafile for (objects) 28
overriding default with HTTP 41, 115
in retention.txt 43–45
specifying a date and time 48–49
specifying an offset 49–51
synchronization with atime attribute 51–57
retention.txt metafile
See also retention; retention classes; retention settings
use with appendable objects 52
changing retention settings 46–51
changing retention settings with HTTP 124
for directories 25
for objects 28
retention settings in 43–45
retention-class, WebDAV property 156, 158
retention-classes.xml metafile 24
retention-hold, WebDAV property 156
retention-string, WebDAV property 156
retention-value, WebDAV property 156
retrieving
custom metadata with HTTP 136–139
deletable objects with CIFS (example) 170
deletable objects with NFS (example) 181
directory listings 108–112
HCP-specific metadata 121–123
object data and custom metadata together 90, 101–102
objects with CIFS (example) 169–170
objects with HTTP 89–102
objects with NFS (example) 180–181
part of an object with HTTP 89–92
return codes
CIFS 176
HTTP 204–208
NFS 186
WebDAV 164–166
root user 34

S

sample
custom metadata file 60
data structure 18
metadata structure 30–31
Search Console 7–8
search facilities
about 7–8
indexes 8
search, object naming considerations 17
secure deletion 57
sending email to the namespace 188
settings metadirectory 19
shred
HTTP query parameter 115
WebDAV property 156, 157, 158
shred settings
See also shred.txt metafile
about 57
changing 58
changing with HTTP 124
metafile for (directories) 25
metafile for (objects) 28
overriding default with HTTP 57, 115
shred.txt metafile
See also shred settings
changing shred settings with HTTP 124
for directories 25
for objects 28
shredding 57
See also shred.txt metafile; shred settings
SMTP
about 5
connectivity 187
default index settings 59

default ownership and permissions 39
default retention settings 41
default shred settings 57
email naming 188–190
ownership of new objects 39
permissions for new objects 39
sending email to the namespace 188
supported operations 11–12
software version, checking 141–143
status codes
See return codes
storage capacity, checking 141–143
storing
See also creating
custom metadata with HTTP 131–134
object data and custom metadata together 78, 84–86
objects 2–3
objects with CIFS (example) 169
objects with HTTP 77–86
objects with NFS (example) 179
zero-sized files with CIFS 172
zero-sized files with HTTP 144
zero-sized files with NFS 182
zero-sized files with WebDAV 162
structuring directories 195–196
supported operations 11–12
symbolic links
creating with NFS (example) 180
with HTTP DELETE 102
with HTTP GET 89
with HTTP HEAD 86
with HTTP TOUCH 126
limitations on 12
operations 12
synchronization, atime 51–57
system metadata
about 2
collisions 63–66

T

temporary files, Windows 174–175
tenants 4
tpof.txt metafile 26
transmitting data in compressed format 75

U

uid
See also user IDs of object owners
changing with HTTP 126
HTTP query parameter 115, 119, 126
WebDAV property 157, 158
Unhold 48
Unix hosts file 193
update-time, WebDAV property 157, 158
URLs
formats for 70–72, 149–151
HTTP access to the namespace 70–73
maximum length 72
for metafiles 71–72
percent encoding 72–73, 152
WebDAV access to the namespace 149–152
usage considerations
CIFS 170–175
general 191–197
HTTP 143–146
NFS 181–185
WebDAV 161–164
user IDs of object owners 36
See also uid
users, root 34
UTF-8 encoding 17

V

viewing permissions 37–38

W

WebDAV
See also individual WebDAV methods
about 5, 192–193
accessing open objects ??–163
basic authentication 161
browsing the namespace 152–153
client timeouts 162
compliance level 147
dead properties 160
failed write operations 163
locking 162
metadata properties 155–158
methods 148–149
multithreading 163–164, 197
naming objects 151
open objects 163–??
ownership of new objects 38
permission checking 161
permissions for new objects 38
persistent connections 161–162
PROPFIND example 159–160
PROPPATCH example 158–159
return codes 164–166
storing zero-sized files 162
supported operations 11–12
URLs for namespace access 149–152
usage considerations 161–164
Windows client access 149
WebDAV COPY 148

WebDAV DELETE 148
WebDAV GET 148
WebDAV HEAD 148
WebDAV LOCK 149, 162
WebDAV MKCOL 148
WebDAV MOVE 148
WebDAV OPTIONS 149
WebDAV POST 149
WebDAV PROPFIND
about 148
example 159–160
WebDAV PROPPATCH
about 148
example 158–159
WebDAV PUT 148
WebDAV TRACE 149
WebDAV UNLOCK 149, 162
whole-object URL query parameter 78
Windows
case sensitivity 170–171
creating empty directories 172
hosts file 193
permissions 38
temporary files 174–175
WebDAV access from 149
write operations
failed CIFS 174
failed HTTP 145
failed NFS 183–184
failed WebDAV 163

X

X-ArcAvailableCapacity response header 142, 209
X-ArcContentLength response header 209
X-ArcCustomMetadataContentType response header 209
X-ArcCustomMetadataFirst response header 209
X-ArcCustomMetadataHash response header 209
X-ArcDataContentType response header 210
X-ArcErrorMessage response header 210
X-ArcHash response header 210
X-ArcPermissionsUidGuid response header 210
X-ArcServicedBySystem response header 210
X-ArcSize response header 210
X-ArcSoftwareVersion response header 142, 210
X-ArcTimes response header 210
X-ArcTotalCapacity response header 142, 211
X-DocCount response header 74
X-DocURI response header 74

Z

zero-sized files, storing
CIFS 172
HTTP 144
NFS 182
WebDAV 162

Hitachi Data Systems

Corporate Headquarters
2845 Lafayette Street
Santa Clara, California 95050-2627
U.S.A.
www.hds.com

Regional Contact Information

Americas
+1 408 970 1000
info@hds.com

Europe, Middle East, and Africa
+44 (0) 1753 618000
info.emea@hds.com

Asia Pacific
+852 3189 7900
hds.marketing.apac@hds.com

Hitachi Data Systems

MK-95ARC012-19