MarkLogic Server

Scripting Administrative Tasks Guide

MarkLogic 10
May, 2019

Last Revised: 10.0-8, October, 2021

Table of Contents

Scripting Administrative Tasks Guide

1.0 Scripting Administrative Tasks in MarkLogic Server

This guide describes how to use the administration APIs to create scripts that configure and manage the operation of MarkLogic Server on your system. This guide is intended for a technical audience, specifically the system administrator in charge of MarkLogic Server.

The topics in this chapter are:

1.1 The Administration APIs

The APIs most often used to automate administrative operations are summarized in the tables below.

Built-in Functions	
Admin (xdmp:)	This API provides functions for database and forest maintenance and file system access.
Security (xdmp:)	This API provides functions for getting information on users, roles, and document access permissions.

Library Module Functions	
Admin (admin:)	This API provides functions for creating, configuring, and deleting forests, databases, hosts, groups, and App Servers. Specific usage information for this API is provided in the next chapter, “Using the Admin API” on page 8.
Security (sec:)	This API provides functions for creating, configuring, and deleting users, roles, amps, and privileges.
PKI (pki:)	This API provides functions for creating, configuring, and deleting the SSL objects used for secure access to App Servers.

1.2 Common Use Cases

The MarkLogic Server Admin APIs provide a flexible toolkit for creating new and managing existing configurations of MarkLogic Server.

The common use cases of the Admin APIs include.

- Configuring multiple, identical instances of MarkLogic Server (across a cluster or multiple clusters and/or to maintain consistency between dev/cert/test/prod environments).
- Automating Server Maintenance. For example, backups based on criteria other than a schedule.
- Managing server resources. For example, delete-only forest rotation.
- Making Bulk Updates to Server Configuration. For example, changing roles across a large subgroup of users.
- Generating notifications (log and/or email) for specific server events.

The chapters “Server Configuration Scripts” on page 35 and “Server Maintenance Operations” on page 51 provide code samples that demonstrate the various uses of the MarkLogic Server Administration APIs.

1.3 Chapter Summary

The Using the Admin API chapter describes the fundamentals of using the Admin and other APIs that allow you to perform administrative operations on MarkLogic Server.

The Server Configuration Scripts chapter demonstrates how to write simple scripts that create and configure MarkLogic Server objects.

The Server Maintenance Operations chapter describes how to write scripts to monitor and manage an existing MarkLogic Server configuration.

2.0 Using the Admin API

MarkLogic Server includes an Admin Library Module that provides XQuery functions that enable you to perform many administrative tasks on MarkLogic Server. With the Admin API, you can write XQuery programs to create or modify databases, forests, App Servers, and perform all kinds of administrative and configuration tasks on MarkLogic Server.

The Admin API is different from the other administrative APIs listed in "The Administration APIs" on page 6 in that the Admin API functions operate on an in-memory cluster configuration that is extracted from and then saved to the configuration files on the server's file system. This design impacts what configuration tasks you can accomplish within a single transaction, as described in "Transactional Considerations When Using the Admin API" on page 9 and "Techniques for Making Multiple Changes to a Configuration" on page 11.

This chapter describes how you can use the Admin API to perform administrative tasks for MarkLogic Server, and includes the following sections:

2.1 Tasks You Can Perform Using the Admin API

The Admin API provides an alternative way to make configuration changes to MarkLogic Server compared to using the Admin Interface. The Admin API is a large library of XQuery functions, providing a programmatic interface to do most things that the Admin Interface does, including:

- Creating and modifying databases
- Creating and modifying forests
- Creating and modifying groups
- Creating and modifying app servers
- Modifying host configurations

2.2 Transactional Considerations When Using the Admin API

The Admin API modifies MarkLogic Server configuration files that are stored on the file system, not in a database. Because the configuration files are not stored in a database, database-level locking does not occur on the configuration files and the transactional semantics are different from updates to a database. For this reason, only one process should update the configuration files at a time:

- Only run one Admin API script that modifies the configuration at a time; do not have concurrent users simultaneously modifying the configuration.
- No other session should modify the configuration with the Admin Interface concurrent to modifying the configuration with the Admin API.

Additionally, avoid doing multiple save operations to the configuration files in the same transaction. Instead, perform each step in memory before passing the changed configuration to the next function, as described in “Techniques for Making Multiple Changes to a Configuration” on page 11.

2.3 Privileges Required For Running Admin APIs

The APIs in the Admin Library module are protected by the following two execute privileges:

- `http://marklogic.com/xdmp/privileges/admin-module-read`
- `http://marklogic.com/xdmp/privileges/admin-module-write`

Any user who runs XQuery code containing the Admin API functions needs to have the first privilege (via a role) for reading anything in the configuration and needs both privileges for writing anything to the configuration. This allows you control access to your configuration information. For details about security in MarkLogic Server, see *Security Guide*.

2.4 General Steps for Scripting Administrative Tasks

Because the Admin API is implemented as an XQuery module, you can write XQuery programs which will modify your MarkLogic Server configuration. Without using the Admin API, you must use the Admin Interface to perform most administrative tasks.

This section outlines the general steps needed to execute functions in the Admin API. These steps assume you are running against either an HTTP App Server or an XDBC App Server using XCC to issue the XQuery requests. This section includes the following parts:

- Importing the Admin Library Module
- Getting the Current Configuration in Memory
- Techniques for Making Multiple Changes to a Configuration
- Saving the Configuration Changes
- When Separate Transactions are Needed

2.4.1 Importing the Admin Library Module

Because the Admin API is an XQuery library module, you must import it into your XQuery program. The Admin API is installed in the `<marklogic-dir>/Modules/MarkLogic/admin.xqy` file. To import the Admin API module, include the following in your XQuery prolog:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
   at "/MarkLogic/admin.xqy";
```

2.4.2 Getting the Current Configuration in Memory

The `admin:get-configuration` function gets an in-memory representation of the configuration for the purpose of modifying the configuration. It is designed to only be used with the Admin API, not to get the configuration for other purposes. To optimize performance, the `admin:get-configuration` function gets only the configuration information it needs to process configuration information in a given XQuery request. The configuration information is an XML structure, and it is designed to be used in conjunction with the other Admin API functions; do not try and use the `admin:get-configuration` function outside of the scope of the Admin API.

The typical design pattern to get the configuration is to get the configuration once per XQuery request and bind its value to a variable (for example, in the `let` clause of a `FLWOR` expression). Then you can use that variable through the query to refer to the configuration. Do not try and change the XML representation returned from the `admin:get-configuration` function.

> **Note:** Because it is optimized to only get the parts of the configuration it needs to make a set of changes, you might find that the `admin:get-configuration` function sometimes returns an empty element if you run it without performing any other functions. It is designed to be used only with the Admin API functions.

2.4.3 Saving the Configuration Changes

Once you use the various APIs to modify the configuration, you must save the configuration to make it take effect in the MarkLogic Server cluster. There are two APIs to save the configuration:

- `admin:save-configuration`
- `admin:save-configuration-without-restart`

Both functions take a configuration element, and then save that element to the configuration files. The configuration file changes are propagated to the entire cluster. The difference between the two functions is that the first one will automatically restart any MarkLogic Server instances when the configuration changes are “cold” and require a restart. The `admin:save-configuration-without-restart` function does not automatically restart MarkLogic Server; if a restart is needed for the changes to take effect, it returns the host IDs of the hosts needing a restart; if no restart is required, it returns the empty sequence.

You should only call these functions (one or the other of two) once per XQuery program (that is, once per transaction).

2.5 Techniques for Making Multiple Changes to a Configuration

When writing a program that creates an entire configuration, it is most efficient to write modules that create and configure multiple server objects. This section describes some useful techniques to use when writing modules that make multiple changes to a configuration. The topics are:

- Saving versus Passing the Configuration
- Creating and Configuring Objects in a Single Transaction
- Making Transactions Idempotent
- When Separate Transactions are Needed

2.5.1 Saving versus Passing the Configuration

There are a number of ways to manage multiple changes to the configuration. The simplest, but most cumbersome, approach is to get and save the configuration for each `admin` function in a separate transaction. For example:

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
         at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
return admin:save-configuration(admin:function-call-1($config, ....));

xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
         at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
return admin:save-configuration(admin:function-call-2($config, ....));
```

Another approach is to pass the configuration returned from a previous `admin` function to the next `admin` function. For example, in the following pseudo-code example, the `$config` variable returned from `admin:get-configuration` holds the initial configuration of MarkLogic Server. The `$config` variable is progressively updated as it is returned from each `admin` function and passed as input to the next `admin` function. The final `$config` variable holds the configuration from all of the `admin` function calls and is saved by `admin:save-configuration`.

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
         at "/MarkLogic/admin.xqy";

(: Get the configuration :)
let $config := admin:get-configuration()
```

```
(: Update the configuration :)
let $config := admin:function-call-1($config, ....)
let $config := admin:function-call-2($config, ....)
(: etc....... :)

(: Save the configuration :)
return admin:save-configuration($config5)
```

2.5.2 Creating and Configuring Objects in a Single Transaction

The `admin:`*`object`*`-get-id` functions (`admin:forest-get-id`, `admin:database-get-id`, `admin:group-get-id`, `admin:appserver-get-id`, `admin:host-get-id`) allow you to create and configure a server object in a single transaction. The main difference between the `admin:`*`object`*`-get-id` functions and their `xdmp:`*`object`* counterparts is that the `admin:`*`object`*`-get-id` functions return object IDs from the configuration, whether it has been saved or not, whereas their `xdmp:`*`object`* counterparts return the object IDs of existing objects in MarkLogic Server. In this way, the `admin:`*`object`*`-get-id` functions provide a faster alternative to what would otherwise be a need to create the object, save the configuration, configure the object, and save the configuration.

For example, the following is an abbreviated version of the code shown in “Creating and Configuring Databases in a Single Transaction” on page 39 :

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
         at "/MarkLogic/admin.xqy";

(: Get the configuration :)
let $config := admin:get-configuration()

(: Add new database to the configuration :)

let $config := admin:database-create(
    $config,
    "Sample-Database",
    xdmp:database("Security"),
    xdmp:database("Schemas"))

(: Obtain the database ID of “Sample-Database” :)

let $Sample-Database := admin:database-get-id(
    $config,
    "Sample-Database")

(: Attach the "SampleDB-Forest" forest to “Sample-Database” :)

let $config := admin:database-attach-forest(
    $config,
    $Sample-Database,
    xdmp:forest("SampleDB-Forest"))
```

```
(: Add an index to "Sample-Database" :)

let $rangespec := admin:database-range-element-index(
    "string",
    "http://marklogic.com/wikipedia",
    "name",
    "http://marklogic.com/collation/",
    fn:false() )

let $config := admin:database-add-range-element-index(
    $config,
    $Sample-Database,
    $rangespec)

return
   admin:save-configuration($config)
```

Note: The IDs returned by the `admin:`*`object`*`-get-id` functions should only be passed to other functions in the Admin library because they operate on the configuration of objects, rather than the existing objects in MarkLogic Server. Functions outside the Admin library only understand the existing objects in MarkLogic Server.

2.5.3 Making Transactions Idempotent

There may be circumstances in which you want to modify a configuration without the need to uninstall it first. For example, you may want to update your configuration program to change an App Server setting or add an index to a database without removing and reinstalling the entire configuration. To enable this option, you need to write idempotent transactions, so that the transaction doesn't fail if some of the objects already exist or have already been deleted.

For example, the transaction that creates and configures a database shown above in "Creating and Configuring Objects in a Single Transaction" on page 12 will fail if any of the objects it attempts to create already exist. To make this transaction idempotent, you need to check for the existence of objects before creating them. If an object exists, then no change is made to the configuration. This allows you to modify the transaction later to create, modify, or delete individual objects without the entire transaction failing.

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
         at "/MarkLogic/admin.xqy";

declare namespace xdmpdb = "http://marklogic.com/xdmp/database";

(: Get the configuration :)
let $config := admin:get-configuration()

(: Get all of the existing databases :)
```

```
let $ExistingDatabases :=
    for $id in admin:get-database-ids($config)
       return admin:database-get-name($config, $id)

(: Check to see if "Sample-Database" already exists.
   If not, create new database :)

let $config :=
    if ("Sample-Database" = $ExistingDatabases)
    then $config
    else
      admin:database-create(
        $config,
        "Sample-Database",
        xdmp:database("Security"),
        xdmp:database("Schemas"))

(: Obtain the database ID of “Sample-Database” :)

let $Sample-Database := admin:database-get-id(
    $config,
    "Sample-Database")

(: Get all of the forests attached to “Sample-Database” :)

let $AttachedForests :=
    admin:forest-get-name(
       $config,
       (admin:database-get-attached-forests(
          $config,
          $Sample-Database) ))

(: Check to see if the "SampleDB-Forest" forest is already attached
   to “Sample-Database”. If not, attach the forest to the database :)

let $config :=
    if ("SampleDB-Forest" = $AttachedForests)
    then $config
    else
       admin:database-attach-forest(
          $config,
          $Sample-Database,
          xdmp:forest("SampleDB-Forest"))

(: Define a new range element index :)

let $rangespec := admin:database-range-element-index(
    "string",
    "http://marklogic.com/wikipedia",
    "name",
    "http://marklogic.com/collation/",
    fn:false() )

(: Get the existing range element indexes for “Sample-Database” :)
```

```
let $ExistingREindexes :=
    fn:data(admin:database-get-range-element-indexes(
      $config,
      $Sample-Database)/xdmpdb:localname)

(: Check to see if the "name" range element index already exists
   for "Sample-Database". If not, add the index :)

let $config :=
    if ("name" = $ExistingREindexes)
    then $config
    else
       admin:database-add-range-element-index(
          $config,
          $Sample-Database,
          $rangespec)

return
   admin:save-configuration($config)
```

2.5.4 When Separate Transactions are Needed

There are cases where you must make configuration changes in separate transactions. If you are creating a new object that is to be used by another object, each task must be done in a separate transaction. For example, to create a forest and attach it to a database, you must first create the forest and save the configuration in one transaction, then attach the forest to the database and save the configuration in a separate transaction. Similarly, if you are deleting an object and then deleting an object that uses that object, the deletes must occur and the configuration must be saved in separate transactions.

You can define separate transactions in a single module by separating the transactions with semi-colons. Alternatively you can create separate modules for each transaction or execute each admin create/delete function using a `different-transaction` isolation of an `xdmp:eval` or `xdmp:invoke`.

Specifically, for the following functions, the forest/database/group must first exist or be deleted/reassigned before calling the function or an exception is thrown:

Function	Object Dependency
`admin:database-attach-forest`	Forest must exist.
`admin:appserver-set-database`	Content database must exist.
`admin:appserver-set-modules-database`	Modules database must exist.
`admin:host-set-group`	Group must exist.
`admin:http-server-create`	Group and databases must exist.

Function	Object Dependency
`admin:xdbc-server-create`	Group and databases must exist.
`admin:webdav-server-create`	Group and databases must exist.
`admin:group-delete`	All hosts must be reassigned to another group.
`admin:database-delete`	All app servers that use the database must be deleted or reassigned another database.
`admin:forest-delete`	All databases that attach the forest must be deleted or attached to another forest.

See “Sequence for Creating Server and Security Objects” on page 16 and “Sequence for Deleting Server and Security Objects” on page 17 for the specific order in which objects should be created and deleted. For details on transactions, see Understanding Transactions in MarkLogic Server in the *Application Developer's Guide*.

2.6 Sequence for Creating Server and Security Objects

When configuring a complete MarkLogic application environment, certain server objects must exist before others can be created. The sequence for creating server objects is shown in the `configure-server` function in the `configure-server.xqy` module in the sample configuration program. The sequence for creating security objects is shown in the `install.xqy` module.

In general, the sequence for creating new server objects is:

1. Create and configure forests

2. *Save configuration*

3. Create and configure databases

4. *Save configuration*

5. Create and configure App Servers

6. *Save configuration*

You can create the security objects before, after or anything in between creating the server objects.

If you are creating new roles and assigning them to new users, the sequence for creating the new security objects is:

1. Create roles

2. Create users

2.7 Sequence for Deleting Server and Security Objects

The sequence for deleting server objects is demonstrated in the `remove-server` function in the `configure-server.xqy` module. The security objects, roles and users, can be deleted in any order.

In general, the sequence for deleting server objects is the reverse order in which they were created.

> **Note:** Deleting an App Server automatically restarts MarkLogic Server. The `remove-server` function uses the `admin:save-configuration-without-restart` function to defer the server restart until all of the operations in the `remove-server` have completed.

1. Delete App Servers

2. Save configuration without restarting server

3. Delete Databases

4. Save configuration without restarting server

5. Delete forests

6. Save configuration and restart server

2.8 Executing Queries in Select Databases

By default, queries are executed in the database set for your App Server.

You can use the `xdmp:eval` or `xdmp:invoke` function to execute queries in a database other than the database set for your App Server. The use of the `xdmp:eval` function to execute queries in the Security and Sample-Modules databases is demonstrated in the `install.xqy` module in the sample configuration program.

For example, you can create a module, named `create-role.xqy`, with the contents:

```
xquery version "1.0-ml";

module namespace
  create-role="http://marklogic.com/sampleConfig/create-role";

import module namespace sec="http://marklogic.com/xdmp/security"
       at "/MarkLogic/security.xqy";

declare function configure-roles()
{
```

```
  sec:create-role(
    "Temporary",
    "Temporary worker access",
    ("filesystem-access"),
    (),
    ("testDocument"))
};
```

You can then call the `create-role.xqy` module from the main module within an `xdmp:eval` function with the `<database>` option that specifies the Security database.

```
xdmp:eval(
  'xquery version "1.0-ml";
  import module namespace create-role=
    "http://marklogic.com/sampleConfig/create-role"
      at "create-role.xqy";
  create-role:configure-roles()',
  (),
  <options xmlns="xdmp:eval">
    <database>{xdmp:database("Security")}</database>
  </options>)
```

3.0 Scripting Cluster Management

You can use the Management REST API to script setting up and adding hosts to a cluster. This chapter covers the following topics:

3.1 Before You Begin

The scripts and examples in this chapter use the `curl` command line tool to make HTTP requests. If you are not familiar with curl, see Introduction to the curl Tool in *REST Application Developer's Guide*. You can replace curl in the example scripts with another tool capable of sending HTTP request.

The examples and scripts assume a Unix shell environment. If you are on Windows and do not have access to a Unix-like shell environment such as Cygwin, you will not be able to use these scripts directly. To understand how to transform the key curl requests for Windows, see Modifying the Example Commands for Windows in *REST Application Developer's Guide*.

You can run the sample scripts in this chapter from any host from which the MarkLogic Server hosts are reachable.

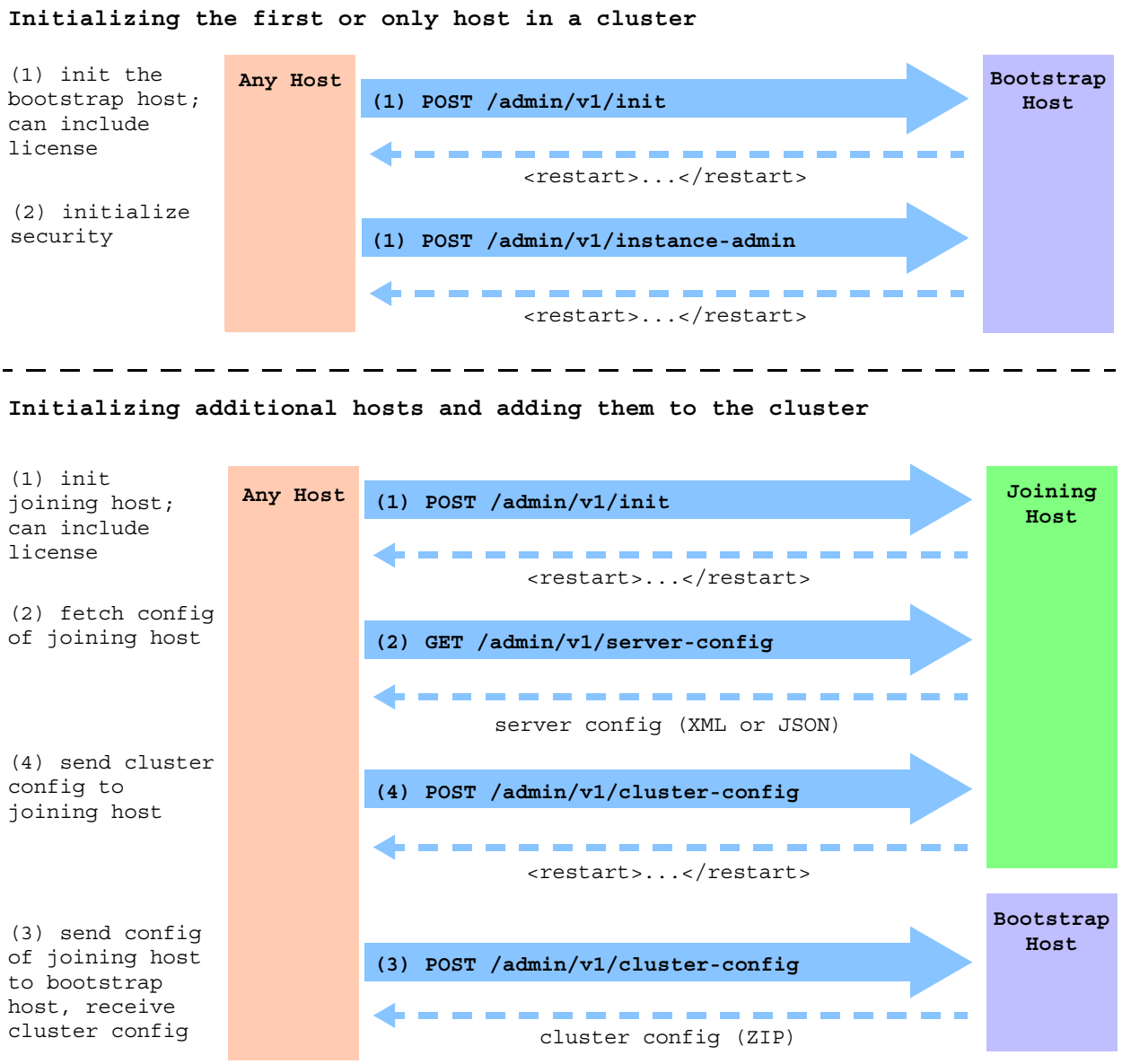
3.2 Using the Management REST API to Create a Cluster

The Management REST API is a set of interfaces for administering, monitoring, and configuring a MarkLogic Server cluster and the resources it contains, such as databases, forests, and App Servers. Though you can interactively set up a cluster during product installation using the Admin Interface, you can only script setting up a cluster and adding hosts to it using the Management REST API.

Cluster creation has two phases: You first fully initialize the first host in the cluster (or a standalone host), and then you add additional hosts. The process of bringing up the first host differs from adding subsequent hosts. You must not apply the "first host" initialize process to a host you expect to eventually add to a cluster.

License installation is optional. If you choose to install a license, you can install it during the basic initialization or add it later. Installing a license later causes an additional restart. Licenses must be installed separately on each host. They are not shared across the cluster.

The diagram below shows the sequence of Management API REST requests required to bring up a multi-host cluster.



When a request causes a restart, MarkLogic Server returns a `restart` element or key-value pair that includes the last startup time of all affected hosts. You can use this information with `GET:/admin/v1/timestamp` to determine when the restart is complete; for details see “Using the Timestamp Service to Verify a Restart” on page 30.

3.3 Setting Up the First Host in a Cluster

Use the procedure outlined in this section to set up the first or only host in a cluster.

> **Note:** You must not use this procedure to bring up the 2nd through Nth host in a cluster. Once you initialize security by calling `POST:/admin/v1/instance-admin`, you cannot add the host to a different cluster without reinstalling MarkLogic Server.

This section covers the following topics:

- Procedure: Set Up the First Host in a Cluster
- Example Script: Set Up the First Host in a Cluster

3.3.1 Procedure: Set Up the First Host in a Cluster

Setting up the first (or only) host in a cluster involves the following Management REST API requests to the host:

- `POST http://`*`bootstrap-host`*`:8001/admin/v1/init`
- `POST http://`*`bootstrap-host`*`:8001/admin/v1/instance-admin`

The following procedure outlines the scriptable steps:

1. Install MarkLogic Server. For details, see Installing MarkLogic in *Installation Guide*. For example:

   ```
   sudo rpm -i /your/location/MarkLogic-8.0-1.x86_64.rpm
   ```

2. Start MarkLogic Server. For details, see Starting MarkLogic Server in *Installation Guide*. For example:

   ```
   sudo /sbin/service MarkLogic start
   ```

3. Initialize MarkLogic Server with a `POST:/admin/v1/init` request. You can optionally install a license during this step. This step causes a restart. For example:

   ```
   curl -X POST -d "" http://${BOOTSTRAP_HOST}:8001/admin/v1/init
   ```

4. Initialize security with a `POST:/admin/v1/instance-admin` request. This step causes a restart. Note that this request passes the Admin password in cleartext, so you should only perform this operation in a secure environment. For example:

   ```
   curl -X POST -H "Content-type: application/x-www-form-urlencoded" \
      --data "admin-username=${USER}" --data "admin-password=${PASS}" \
      --data "wallet-password=${WPASS}" --data "realm=${SEC_REALM}" \
      http://${BOOTSTRAP_HOST}:8001/admin/v1/instance-admin
   ```

When this procedure is completed, MarkLogic Server is fully operational on the host, and you can configure forests, databases, and App Servers, or add additional hosts to the cluster.

Note: Once you successfully complete POST `/admin/v1/instance-admin`, security is initialized, and all subsequent requests require authentication.

3.3.2 Example Script: Set Up the First Host in a Cluster

The following `bash` shell script assumes

- MarkLogic Server has already been installed and started on the host. You can include these steps in your script. They are omitted here for simplicity.
- You are not installing a license.

Use the script by specifying at least two hosts on the command line.

```
this_script [options] bootstrap_host
```

Use the command line options in the following table to tailor the script to your environment:

Option	Description
`-a` *`auth_mode`*	The HTTP authentication method to use for requests that require authentication. Allowed values: `basic`, `digest`, `anyauth`. Default: `anyauth`.
`-p` *`password`*	The password for the administrative user to use for HTTP requests that require authentication. Default: `password`.
`-r` *`sec_realm`*	The authentication realm for the host. For details, see Realm in *Administrator's Guide*. Default: `public`.
`-u` *`username`*	The administrative username to use for HTTP requests that require authentication. Default: `admin`.

This script makes use of the restart checking technique described in "Setting Up the First Host in a Cluster" on page 21. This script performs only minimal error checking and is not meant for production use.

```
#!/bin/bash
#################################################################
# Use this script to initialize the first (or only) host in
# a MarkLogic Server cluster. Use the options to control admin
# username and password, authentication mode, and the security
# realm. If no hostname is given, localhost is assumed. Only
# minimal error checking is performed, so this script is not
# suitable for production use.
#
# Usage:  this_command [options] hostname
```

```
#
#################################################################

BOOTSTRAP_HOST="localhost"
USER="admin"
PASS="password"
WPASS="wpass"
AUTH_MODE="anyauth"
SEC_REALM="public"
N_RETRY=5
RETRY_INTERVAL=10

#########################################################
# restart_check(hostname, baseline_timestamp, caller_lineno)
#
# Use the timestamp service to detect a server restart, given a
# a baseline timestamp. Use N_RETRY and RETRY_INTERVAL to tune
# the test length. Include authentication in the curl command
# so the function works whether or not security is initialized.
#   $1 :  The hostname to test against
#   $2 :  The baseline timestamp
#   $3 :  Invokers LINENO, for improved error reporting
# Returns 0 if restart is detected, exits with an error if not.
#
function restart_check {
  LAST_START=`$AUTH_CURL "http://$1:8001/admin/v1/timestamp"`
  for i in `seq 1 ${N_RETRY}`; do
    if [ "$2" == "$LAST_START" ] || [ "$LAST_START" == "" ]; then
      sleep ${RETRY_INTERVAL}
      LAST_START=`$AUTH_CURL "http://$1:8001/admin/v1/timestamp"`
    else
      return 0
    fi
  done
  echo "ERROR: Line $3: Failed to restart $1"
  exit 1
}


#########################################################
# Parse the command line

OPTIND=1
while getopts ":a:p:r:u:" opt; do
  case "$opt" in
    a) AUTH_MODE=$OPTARG ;;
    p) PASS=$OPTARG ;;
    w) WPASS=$OPTARG ;;
    r) SEC_REALM=$OPTARG ;;
    u) USER=$OPTARG ;;
    \?) echo "Unrecognized option: -$OPTARG" >&2; exit 1 ;;
  esac
done
shift $((OPTIND-1))
```

```
if [ $# -ge 1 ]; then
  BOOTSTRAP_HOST=$1
  shift
fi

# Suppress progress meter, but still show errors
CURL="curl -s -S"
# Add authentication related options, required once security is
initialized
AUTH_CURL="${CURL} --${AUTH_MODE} --user ${USER}:${PASS}" --wpass
${WPASS}


#########################################################
# Bring up the first (or only) host in the cluster. The following
# requests are sent to the target host:
#   (1) POST /admin/v1/init
#   (2) POST
/admin/v1/instance-admin?admin-user=W&admin-password=X&wallet-password=Y&realm
=Z
# GET /admin/v1/timestamp is used to confirm restarts.

# (1) Initialize the server
echo "Initializing $BOOTSTRAP_HOST..."
$CURL -X POST -d "" http://${BOOTSTRAP_HOST}:8001/admin/v1/init
sleep 10

# (2) Initialize security and, optionally, licensing. Capture the last
#     restart timestamp and use it to check for successful restart.
TIMESTAMP=`$CURL -X POST \
   -H "Content-type: application/x-www-form-urlencoded" \
   --data "admin-username=${USER}" --data "admin-password=${PASS}" \
   --wallet-password "wpass=${WPASS}" --data "realm=${SEC_REALM}" \
   http://${BOOTSTRAP_HOST}:8001/admin/v1/instance-admin \
   | grep "last-startup" \
   | sed 's%^.*<last-startup.*>\(.*\)</last-startup>.*$%\1%'`
if [ "$TIMESTAMP" == "" ]; then
  echo "ERROR: Failed to get instance-admin timestamp." >&2
  exit 1
fi

# Test for successful restart
restart_check $BOOTSTRAP_HOST $TIMESTAMP $LINENO

echo "Initialization complete for $BOOTSTRAP_HOST..."
exit 0
```

3.4 Adding an Additional Host to a Cluster

Use the procedure described by this section to configure the 2nd through Nth hosts in a cluster. This section covers the following topics:

- Procedure: Add a Host to a Cluster
- Example Script: Add Hosts to a Cluster

3.4.1 Procedure: Add a Host to a Cluster

Once you configure the first host in a cluster, add additional hosts to the cluster by using the following series of Management REST API requests for each host:

- `POST http://joining-host:8001/admin/v1/init`
- `GET http://joining-host:8001/admin/v1/server-config`
- `POST http://bootstrap-host:8001/admin/v1/cluster-config`
- `POST http://joining-host:8001/admin/v1/cluster-config`

The following procedure outlines the scriptable steps:

1. Install MarkLogic Server on the joining host. For details, see Installing MarkLogic in *Installation Guide*. For example:

   ```
   sudo rpm -i /your/location/MarkLogic-8.0-1.x86_64.rpm
   ```

2. Start MarkLogic Server on the joining host. For details, see Starting MarkLogic Server in *Installation Guide*. For example:

   ```
   sudo /sbin/service MarkLogic start
   ```

3. Initialize MarkLogic Server on the joining host by sending a POST request to `/admin/v1/init`. Authentication is not required. This step causes a restart.

   ```
   curl -X POST -d "" http://${JOINING_HOST}:8001/admin/v1/init
   ```

4. Fetch the configuration of the joining host with a `GET:/admin/v1/server-config` request. Authentication is not required. The following example saves the config to a shell variable. You can also save it to a file by using curl's `-o` option.

   ```
   JOINER_CONFIG=`curl -s -S -X GET -H "Accept: application/xml" \
     http://${JOINING_HOST}:8001/admin/v1/server-config`
   ```

5. Send the joining host configuration data to a host already in the cluster with a `POST:/admin/v1/cluster-config` request. The cluster host responds with cluster configuration data in ZIP format.

   The following example command assumes the input server config is in the shell variable `JOINER_CONFIG` and saves the output cluster configuration to `cluster-config.zip`.

```
curl -s -S --digest --user admin:password -X POST
  -o cluster-config.zip -d "group=Default" \
  --data-urlencode "server-config=${JOINER_CONFIG}" \
  -H "Content-type: application/x-www-form-urlencoded" \
  http://${BOOTSTRAP_HOST}:8001/admin/v1/cluster-config
```

6. Send the cluster configuration ZIP data to the joining host with a `POST:/admin/v1/cluster-config` request. This completes the join sequence. This step causes a restart.

```
curl -s -S -X POST -H "Content-type: application/zip" \
  --data-binary @./cluster-config.zip \
  http://${JOINING_HOST}:8001/admin/v1/cluster-config
```

Once this procedure completes, the joining host is a fully operational member of the cluster.

3.4.2 Example Script: Add Hosts to a Cluster

The following `bash` shell script assumes

- MarkLogic Server has already been fully initialized on the bootstrap host, using either the Admin Interface or the Management REST API. You can include this step in your script. It is omitted here for simplicity.
- MarkLogic Server has already been installed and started on each host joining the cluster. You can include these steps in your script. They are omitted here for simplicity.
- You are not installing a license.
- The joining host should be in the Default group.

The example script completes the cluster join sequence for each host serially. However, you can also add hosts concurrently. For details, see Adding Hosts to a Cluster Concurrently.

Use the script by specifying at least two hosts on the command line. A fully initialized host that is already part of the cluster must be the first parameter.

```
this_script [options] bootstrap_host joining_host [joining_host...]
```

Use the command line options in the following table to tailor the script to your environment:

Option	Description
`-a` *`auth_mode`*	The HTTP authentication method to use for requests that require authentication. Allowed values: `basic`, `digest`, `anyauth`. Default: `anyauth`.
`-p` *`password`*	The password for the administrative user to use for HTTP requests that require authentication. Default: `password`.
`-u` *`username`*	The administrative username to use for HTTP requests that require authentication. Default: `admin`.

The script makes use of the restart checking technique described in “Using the Timestamp Service to Verify a Restart” on page 30. This script performs only minimal error checking and is not meant for production use.

```
#!/bin/bash
################################################################
# Use this script to initialize and add one or more hosts to a
# MarkLogic Server cluster. The first (bootstrap) host for the
# cluster should already be fully initialized.
#
# Use the options to control admin username and password,
# authentication mode, and the security realm. At least two hostnames
# must be given: A host already in the cluster, and at least one host
# to be added to the cluster. Only minimal error checking is performed,
# so this script is not suitable for production use.
#
# Usage:  this_command [options] cluster-host joining-host(s)
#
################################################################

USER="admin"
PASS="password"
AUTH_MODE="anyauth"
N_RETRY=5
RETRY_INTERVAL=10

#########################################################
# restart_check(hostname, baseline_timestamp, caller_lineno)
#
# Use the timestamp service to detect a server restart, given a
# a baseline timestamp. Use N_RETRY and RETRY_INTERVAL to tune
# the test length. Include authentication in the curl command
# so the function works whether or not security is initialized.
#   $1 :  The hostname to test against
```

```
#   $2 :  The baseline timestamp
#   $3 :  Invokers LINENO, for improved error reporting
# Returns 0 if restart is detected, exits with an error if not.
#
function restart_check {
  LAST_START=`$AUTH_CURL "http://$1:8001/admin/v1/timestamp"`
  for i in `seq 1 ${N_RETRY}`; do
    if [ "$2" == "$LAST_START" ] || [ "$LAST_START" == "" ]; then
      sleep ${RETRY_INTERVAL}
      LAST_START=`$AUTH_CURL "http://$1:8001/admin/v1/timestamp"`
    else
      return 0
    fi
  done
  echo "ERROR: Line $3: Failed to restart $1"
  exit 1
}


############################################################
# Parse the command line

OPTIND=1
while getopts ":a:p:u:" opt; do
  case "$opt" in
    a) AUTH_MODE=$OPTARG ;;
    p) PASS=$OPTARG ;;
    u) USER=$OPTARG ;;
    \?) echo "Unrecognized option: -$OPTARG" >&2; exit 1 ;;
  esac
done
shift $((OPTIND-1))

if [ $# -ge 2 ]; then
  BOOTSTRAP_HOST=$1
  shift
else
  echo "ERROR: At least two hostnames are required." >&2
  exit 1
fi
ADDITIONAL_HOSTS=$@

# Curl command for all requests. Suppress progress meter (-s),
# but still show errors (-S)
CURL="curl -s -S"
# Curl command when authentication is required, after security
# is initialized.
AUTH_CURL="${CURL} --${AUTH_MODE} --user ${USER}:${PASS}"


############################################################
# Add one or more hosts to a cluster. For each host joining
# the cluster:
#   (1) POST /admin/v1/init (joining host)
```

```
#   (2) GET /admin/v1/server-config (joining host)
#   (3) POST /admin/v1/cluster-config (bootstrap host)
#   (4) POST /admin/v1/cluster-config (joining host)
# GET /admin/v1/timestamp is used to confirm restarts.

for JOINING_HOST in $ADDITIONAL_HOSTS; do
  echo "Adding host to cluster: $JOINING_HOST..."

  # (1) Initialize MarkLogic Server on the joining host
  TIMESTAMP=`$CURL -X POST -d "" \
     http://${JOINING_HOST}:8001/admin/v1/init \
     | grep "last-startup" \
     | sed 's%^.*<last-startup.*>\(.*\)</last-startup>.*$%\1%'`
  if [ "$TIMESTAMP" == "" ]; then
    echo "ERROR: Failed to initialize $JOINING_HOST" >&2
    exit 1
  fi
  restart_check $JOINING_HOST $TIMESTAMP $LINENO

  # (2) Retrieve the joining host's configuration
  JOINER_CONFIG=`$CURL -X GET -H "Accept: application/xml" \
        http://${JOINING_HOST}:8001/admin/v1/server-config`
  echo $JOINER_CONFIG | grep -q "^<host"
  if [ "$?" -ne 0 ]; then
    echo "ERROR: Failed to fetch server config for $JOINING_HOST"
    exit 1
  fi

  # (3) Send the joining host's config to the bootstrap host, receive
  #     the cluster config data needed to complete the join. Save the
  #     response data to cluster-config.zip.
  $AUTH_CURL -X POST -o cluster-config.zip -d "group=Default" \
        --data-urlencode "server-config=${JOINER_CONFIG}" \
        -H "Content-type: application/x-www-form-urlencoded" \
        http://${BOOTSTRAP_HOST}:8001/admin/v1/cluster-config
  if [ "$?" -ne 0 ]; then
    echo "ERROR: Failed to fetch cluster config from $BOOTSTRAP_HOST"
    exit 1
  fi
  if [ `file cluster-config.zip | grep -cvi "zip archive data"` -eq 1
]; then
    echo "ERROR: Failed to fetch cluster config from $BOOTSTRAP_HOST"
    exit 1
  fi

  # (4) Send the cluster config data to the joining host, completing
  #     the join sequence.
  TIMESTAMP=`$CURL -X POST -H "Content-type: application/zip" \
      --data-binary @./cluster-config.zip \
      http://${JOINING_HOST}:8001/admin/v1/cluster-config \
      | grep "last-startup" \
      | sed 's%^.*<last-startup.*>\(.*\)</last-startup>.*$%\1%'`
  restart_check $JOINING_HOST $TIMESTAMP $LINENO
  rm ./cluster-config.zip
```

```
  echo "...$JOINING_HOST successfully added to the cluster."
done
```

3.5 Adding Hosts to a Cluster Concurrently

The REST Management API is designed to support safe, concurrent cluster topology changes. For example, you can send server configuration data to a bootstrap host from multiple hosts, at the same time.

Only the REST Management API offers safe concurrency support. If you make changes using another interface, such as the Admin Interface, Configuration Manager, XQuery Admin API, or REST Packaging API, no such concurrency guarantees exist. With any other interface, even in combination with the REST Management API, you must ensure that no concurrent change requests occur.

3.6 Using the Timestamp Service to Verify a Restart

When you use the Management REST API to perform an operation that causes MarkLogic Server to restart, the request that caused the restart returns a response that includes the last restart time, similar to the following:

```
<restart xmlns="http://marklogic.com/manage">
  <last-startup host-id="13544732455686476949">
    2013-05-15T09:01:43.019261-07:00
  </last-startup>
  <link>
    <kindref>timestamp</kindref>
    <uriref>/admin/v1/timestamp</uriref>
  </link>
  <message>Check for new timestamp to verify host restart.</message>
</restart>
```

If the operation causes multiple hosts to restart, the data in the response includes a `last-startup` timestamp for all affected hosts.

You can use the `last-startup` timestamp in conjunction with `/admin/v1/timestamp` to detect a successful restart. If MarkLogic Server is operational, a GET request to `/admin/v1/timestamp` returns a 200 (OK) HTTP status code and the timestamp of the last MarkLogic Server startup:

```
$ curl --anyauth --user user:password -X GET \
    http://localhost:8001/admin/v1/timestamp
2013-05-15T10:34:38.932514-07:00
```

If such a request returns an HTTP response code other than 200 (OK), it is not safe to proceed with subsequent administrative requests.

By comparing the `last-startup` to the current timestamp, you can detect when the restart is completed. If the timestamp doesn't change after some reasonable time, you can conclude the restart was not successful. The following `bash` shell function performs this check:

```
#!/bin/bash
# ...
AUTH_CURL="curl -s -S --${AUTH_MODE} --user ${USER}:${PASS}"
N_RETRY=5
RETRY_INTERVAL=10
...
function restart_check {
  LAST_START=`$AUTH_CURL "http://$1:8001/admin/v1/timestamp"`
  for i in `seq 1 ${N_RETRY}`; do
    if [ "$2" == "$LAST_START" ] || [ "$LAST_START" == "" ]; then
      sleep ${RETRY_INTERVAL}
      LAST_START=`$AUTH_CURL "http://$1:8001/admin/v1/timestamp"`
    else
      return 0
    fi
  done
  echo "ERROR: Line $3: Failed to restart $1"
  exit 1
}
```

To use the function, capture the timestamp from an operation that causes a restart, and pass the timestamp, host name, and current line number to the function. The following example assumes only a single host is involved in the restart and uses the `sed` line editor to strip the timestamp from the `<restart/>` data returned by the request.

```
TIMESTAMP=`curl -s -S -X POST ... \
   http://${HOST}:8001/admin/v1/instance-admin \
   | sed 's%^.*<last-startup.*>\(.*\)</last-startup>.*$%\1%'`
if [ "$TIMESTAMP" == "" ]; then
  echo "ERROR: Failed to get restart timestamp." >&2
  exit 1
else
  restart_check $BOOTSTRAP_HOST $TIMESTAMP $LINENO
fi
```

Note: The `/admin/v1/timestamp` service requires digest authentication only after security is initialized, but the `restart_check` function shown here skips this distinction for simplicity and always passes authentication information.

An operation that causes multiple hosts to restart requires a more sophisticated check that iterates through all the `last-startup` host-id's and timestamps.

3.7 Controlling the Format of Input, Output, and Errors

This section describes REST Management API conventions for the format of input data, response data, and error details. The following topics are covered:

- Specifying Input Format
- Specifying Expected Output Format
- How Error Format is Determined

3.7.1 Specifying Input Format

Most methods of the REST Management API accept input as XML or JSON. Some methods accept URL-encoded form data (MIME type `application/x-www-form-urlencoded`). Use the HTTP Content-type request header to indicate the format of your input.

For details, see the *REST Management API Reference*.

3.7.2 Specifying Expected Output Format

Many methods can return data as XML or JSON. The monitoring GET methods, such as `GET:/manage/v2/clusters`, also support HTML.

The response data format is controlled through either the HTTP Accept header or a `format` request parameter (where available). When both the header and the parameter are present, the format parameter takes precedence.

For details, see the *REST Management API Reference*.

3.7.3 How Error Format is Determined

If a request results in an error, the body of the response includes error details. The MIME type of error details in the response is determined by the `format` request parameter (where supported), Accept header, or request Content-type header, in that order of precedence.

For example, if a request supplies XML input (request Content-type set to `application/xml`), but specifies JSON output using the `format` parameter, then error details are returned as JSON. If a request supplies JSON input, but no Accept header or `format` parameter, then error details are returned as JSON.

The default error detail format is XML.

3.8 Scripting Additional Administrative Tasks

Once you have initialized the hosts in your cluster, you can configure databases and App Servers using the Admin Interface, the XQuery Admin API, or the Management REST API. The REST Management API supports scripting of many administrative tasks, including the ones listed in the table below. For more details, see the *REST Management API Reference*.

Operation	REST Method
Restart or shutdown a cluster	`POST:/manage/v2/clusters/{id|name}`
Restart or shutdown a host	`POST:/manage/v2/hosts/{id|name}`
Remove a host from a cluster	`DELETE:/admin/v1/host-config`
Create a forest	`POST:/manage/v2/forests`
Enable or disable a forest	`PUT:/manage/v2/forests/{id|name}/properties`
Combine forests or migrate data to a new data directory	`PUT:/manage/v2/forests`
Delete a forest	`DELETE:/manage/v2/forests/{id|name}`
Change properties of a host, such as hostname or group	`PUT:/manage/v2/hosts/{id|name}/properties`
Create a database partition	`POST:/manage/v2/databases/{id|name}/partitions`
Resize, transfer, or migrate a partition	`PUT:/manage/v2/databases/{id|name}/partitions/{name}`

Operation	REST Method

Package database and App Server configurations for deployment on another host.	`POST:/manage/v2/packages`
	
	`POST:/manage/v2/packages/{pkgname}/databases/{name}`
	
	`POST:/manage/v2/packages/{pkgname}/servers/{name}`

Install packaged database and App Server configuration on a host.	`POST:/manage/v2/packages/{pkgname}/install`
Monitor real time usage and status of a cluster and its resources	`GET /manage/v2/`*`resource`* Where *resource* is one of: clusters, databases, forests, groups, hosts, or servers.
Manage historical usage of a cluster and its resources	`GET /manage/v2/`*`resource`*`?view=metrics` Where *resource* is one of: clusters, databases, forests, groups, hosts, or servers.

You can also use the Admin Interface and the XQuery Admin API to perform these and other operations. For details, see the following:

- “Server Maintenance Operations” on page 51
- *Administrator’s Guide*

4.0 Server Configuration Scripts

There are two basic approaches to creating and configuring server objects. One approach is to write a separate XQuery script for each new object, as shown in this chapter. The other approach is to write a more comprehensive XQuery program that creates all of the new server objects, as shown in the chapter. The approach you select will depend on your individual needs. The script-per-object approach is useful when making minor changes to an existing configuration. If your objective is to create a complete server configuration, you will probably want to write a complete configuration program like the sample configuration program.

The main topics in this chapter are:

- Creating and Configuring Forests and Databases
- Creating and Configuring Groups
- Creating and Configuring App Servers
- Creating and Configuring Roles and Users

4.1 Creating and Configuring Forests and Databases

The general information on forests and databases is provided in the Forests and Databases chapters in the *Administrator's Guide*.

This section shows individual XQuery scripts for tasks such as creating and configuring forests and databases. For more examples on how to use the Admin functions to modify your databases and monitor their operation, see "Database Maintenance Operations" on page 60.

The topics in this section are:

- Creating Forests and Databases
- Attaching Forests to Databases
- Adding a Database Field and Included Element
- Adding Indexes to a Database
- Creating a Scheduled Backup of a Database
- Creating and Configuring Databases in a Single Transaction
- Deleting a Forest and Database

Note: Running the examples in this section will modify your MarkLogic Server configuration.

4.1.1 Creating Forests and Databases

The following script creates two new forests, named `SampleDB-Forest` and `SampleModules-Forest` and two new databases, named `Sample-Database` and `Sample-Modules`. Note that the `$config` variable holds the progressive configurations, each of which is then passed as input to the next `admin` function, so that the return value held by the final `$config` variable passed to `admin:save-configuration` contains the configuration data for all of the new forests and databases.

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Get the configuration :)
let $config := admin:get-configuration()

(: Add new forests to the configuration :)
let $config := admin:forest-create(
  $config,
  "SampleDB-Forest",
  xdmp:host(), (
  ))

let $config := admin:forest-create(
  $config,
  "SampleModules-Forest",
  xdmp:host(),
  ())

(: Add new databases to the configuration :)
let $config := admin:database-create(
  $config,
  "Sample-Database",
  xdmp:database("Security"),
  xdmp:database("Schemas"))

let $config := admin:database-create(
  $config,
  "Sample-Modules",
  xdmp:database("Security"),
  xdmp:database("Schemas"))

(: Save the configuration :)
return admin:save-configuration($config)
```

4.1.2 Attaching Forests to Databases

The following script attaches the `SampleDB-Forest` forest to the `Sample-Database` and the `SampleModules-Forest` to the `Sample-Modules` database:

```
(: Now that the database and forest have been created, we can attach
the forest to the database. :)
```

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Get the configuration with the new forest and database :)
let $config := admin:get-configuration()

(: Attach the forest to the database :)
let $config := admin:database-attach-forest(
  $config,
  xdmp:database("Sample-Database"),
  xdmp:forest("SampleDB-Forest"))

let $config := admin:database-attach-forest(
  $config,
  xdmp:database("Sample-Modules"),
  xdmp:forest("SampleModules-Forest"))

(: Save the configuration :)
return admin:save-configuration($config)
```

4.1.3 Adding a Database Field and Included Element

As described in Overview of Fields in the *Administrator's Guide*, fields enable users to query portions of a database based on elements. If a field is configured with an included element, that element is indexed by the field so that the included element is also searched when searching for the field.

The following script adds the 'wiki-suggest' field to the Sample-Database.

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Get the configuration with the new database :)
let $config := admin:get-configuration()
let $dbid := xdmp:database("Sample-Database")
let $fieldspec := admin:database-field("wiki-suggest", fn:false())

return admin:save-configuration(
  admin:database-add-field($config, $dbid, $fieldspec))
```

This script adds `name` as an included element to the `wiki-suggest` field :

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Get the configuration with the new field :)
let $config := admin:get-configuration()
let $dbid := xdmp:database("Sample-Database")
let $fieldspec := admin:database-included-element(
```

```
  "http://marklogic.com/wikipedia",
  "name",
  1.0,
  "",
  "",
  "")

return admin:save-configuration(
    admin:database-add-field-included-element(
  $config,
  $dbid,
  "wiki-suggest",
  $fieldspec) )
```

4.1.4 Adding Indexes to a Database

As described in Understanding Range Indexes in the *Administrator's Guide*, you can create range indexes on elements or attributes of type xs:string to accelerate the performance of queries that sort by the string values.

The following script sets a range element index for the `name` element in the `Sample-Database` database:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $dbid := xdmp:database("Sample-Database")
let $rangespec := admin:database-range-element-index(
  "string",
  "http://marklogic.com/wikipedia",
  "name",
  "http://marklogic.com/collation/",
  fn:false() )

return admin:save-configuration(
  admin:database-add-range-element-index(
    $config,
    $dbid,
    $rangespec))
```

The following script sets a range element attribute index for the `year` attribute of the `nominee` element in the `Sample-Database` database:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $dbid := xdmp:database("Sample-Database")
let $rangespec := admin:database-range-element-attribute-index(
```

```
  "gYear",
  "http://marklogic.com/wikipedia",
  "nominee",
  "",
  "year",
  "",
  fn:false())

return admin:save-configuration(
  admin:database-add-range-element-attribute-index(
    $config,
    $dbid,
    $rangespec))
```

4.1.5 Creating a Scheduled Backup of a Database

The following script creates a weekly backup of the `Sample-Database` database and adds it to the configuration:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $database := xdmp:database("Sample-Database")
let $backup := admin:database-weekly-backup(
  "c:/backup-dir",
  "monday",
  xs:time("09:45:00"),
  10,
  true(),
  true(),
  true())

return
  admin:save-configuration(
    admin:database-add-backup($config, $database, $backup))
```

4.1.6 Creating and Configuring Databases in a Single Transaction

In this example, we create and configure the same databases as in the previous sections. Only the databases are created and configured in a single transaction. As described in “Creating and Configuring Objects in a Single Transaction” on page 12, we use the `admin:database-get-id` function to obtain the database IDs after creating the databases to configure the newly created databases in the same transaction. This example is abbreviated for the sake of simplicity and does not check for existing objects, as described in “Making Transactions Idempotent” on page 13.

Note: This example assumes the forests, `SampleDB-Forest` and `SampleModules-Forest`, have been already been created in a separate transaction.

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
         at "/MarkLogic/admin.xqy";

(: Get the configuration :)
let $config := admin:get-configuration()

(: Add new databases to the configuration :)

let $config := admin:database-create(
  $config,
  "Sample-Database",
  xdmp:database("Security"),
  xdmp:database("Schemas"))

let $config := admin:database-create(
  $config,
  "Sample-Modules",
  xdmp:database("Security"),
  xdmp:database("Schemas"))

(: Obtain the database IDs to configure the databases :)

let $Sample-Database := admin:database-get-id(
  $config,
  "Sample-Database")

let $Sample-Modules := admin:database-get-id(
  $config,
  "Sample-Modules")

(: Attach the forest to the database. :)

let $config := admin:database-attach-forest(
  $config,
  $Sample-Database,
  xdmp:forest("SampleDB-Forest"))

let $config := admin:database-attach-forest(
  $config,
  $Sample-Modules,
  xdmp:forest("SampleModules-Forest"))

(: Add a 'wiki-suggest' field to the Sample-Database :)

let $fieldspec := admin:database-field(
  "wiki-suggest",
  fn:false())

let $config := admin:database-add-field(
  $config,
  $Sample-Database,
  $fieldspec)
```

```
(: Add included elements to 'wiki-suggest' field :)

let $incfieldspec := admin:database-included-element(
  "http://marklogic.com/wikipedia",
  "name",
  1.0,
  "",
  "",
  "")

let $config := admin:database-add-field-included-element(
  $config,
  $Sample-Database,
  "wiki-suggest",
  $incfieldspec)

(: Add indexes to the Sample-Database :)

let $rangespec := admin:database-range-element-index(
  "string",
  "http://marklogic.com/wikipedia",
  "name",
  "http://marklogic.com/collation/",
  fn:false() )

let $config := admin:database-add-range-element-index(
    $config,
    $Sample-Database,
    $rangespec)

let $rangespec := admin:database-range-element-attribute-index(
  "gYear",
  "http://marklogic.com/wikipedia",
  "nominee",
  "",
  "year",
  "",
  fn:false())

let $config := admin:database-add-range-element-attribute-index(
  $config,
  $Sample-Database,
  $rangespec)

(: Configure a scheduled backup of the Sample-Database :)

let $backup := admin:database-weekly-backup(
  "c:/backup-dir",
  "monday",
  xs:time("09:45:00"),
  10,
  true(),
  true(),
  true())
```

```
let $config := admin:database-add-backup(
  $config,
  $Sample-Database,
  $backup)

return
  admin:save-configuration($config)
```

4.1.7 Deleting a Forest and Database

The following script deletes the forest and database created in “Creating and Configuring Databases in a Single Transaction” on page 39:

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Get the configuration :)
let $config := admin:get-configuration()

(: Delete the database from the configuration :)
let $config := admin:database-delete(
  $config,
  admin:database-get-id($config, "Sample-Database"))

let $config := admin:database-delete(
  $config,
  admin:database-get-id($config, "Sample-Modules"))

(: Save the configuration :)
return admin:save-configuration($config);

(: Now that the database has been deleted, we can delete the forest :)

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Get the configuration with the deleted database :)
let $config := admin:get-configuration()

(: Delete the forest from the configuration :)
let $config := admin:forest-delete(
  $config,
  admin:forest-get-id($config, "SampleDB-Forest"),
  fn:true())

let $config := admin:forest-delete(
  $config,
  admin:forest-get-id($config, "SampleModules-Forest"),
  fn:true())
```

```
(: Save the configuration :)
return admin:save-configuration($config)
```

4.2 Creating and Configuring Groups

The general information on groups is provided in the Groups chapter in the *Administrator's Guide*.

This section shows individual XQuery scripts that create and configure groups. See "Group Maintenance Operations" on page 51 for more examples on how to use the Admin functions to modify your groups and monitor their operation.

The topics in this section are:

- Creating a Group
- Enabling Auditing on a Group
- Creating a New Namespace for a Group
- Creating and Configuring a Group in a Single Transaction
- Deleting a Group

4.2.1 Creating a Group

The following script creates a new group, named `Sample`:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

return
  admin:save-configuration(
    admin:group-create($config, "Sample"))
```

4.2.2 Enabling Auditing on a Group

The following script enables auditing of `user-configuration-change` and `user-role-addition` events on the `Sample` group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Sample")
```

```
(: Set user-configuration-change and user-role-addition events to be
audited in the "Sample" group. :)

let $config :=
  admin:group-enable-audit-event-type(
    $config,
    $groupid,
    ("user-configuration-change", "user-role-addition"))

(: Enable auditing for the "Sample" group. :)

return
  admin:save-configuration(
    admin:group-set-audit-enabled($config, $groupid, fn:true()))
```

4.2.3 Creating a New Namespace for a Group

The following script creates a new namespace, named `myprefix`, and adds it to the `Sample` group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Sample")

return
  admin:save-configuration(
    admin:group-add-namespace(
      $config,
      $groupid,
      admin:group-namespace("myprefix", "http://myuri/namespace")))
```

4.2.4 Creating and Configuring a Group in a Single Transaction

In this example, we create and configure the same group as in the previous sections. Only the group is created and configured in a single transaction. As described in "Creating and Configuring Objects in a Single Transaction" on page 12, we use the `admin:group-get-id` function to obtain the group ID after creating the group to configure the newly created group in the same transaction. This example is abbreviated for the sake of simplicity and does not check for existing objects, as described in "Making Transactions Idempotent" on page 13.

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

(: Create the "Sample" group :)
let $config := admin:group-create($config, "Sample")
```

```
(: Obtain the group ID to configure the group :)
let $SampleGroup := admin:group-get-id($config, "Sample")

(: Set user-configuration-change and user-role-addition events to be
   audited in the "Sample" group. :)
let $config := admin:group-enable-audit-event-type(
  $config,
  $SampleGroup,
  ("user-configuration-change", "user-role-addition"))

(: Enable auditing for the "Sample" group. :)
let $config := admin:group-set-audit-enabled(
  $config,
  $SampleGroup,
  fn:true())

(: Add a namespace to the "Sample" group. :)
let $config := admin:group-add-namespace(
  $config,
  $SampleGroup,
  admin:group-namespace("myprefix", "http://myuri/namespace"))

return admin:save-configuration($config)
```

4.2.5 Deleting a Group

The following script deletes the group created in "Creating and Configuring a Group in a Single Transaction" on page 44:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

return admin:save-configuration(
  admin:group-delete(
    $config,
    admin:group-get-id($config, "Sample")))
```

4.3 Creating and Configuring App Servers

The general information on App Servers is provided in the HTTP Servers, XDBC Servers, and WebDAV Servers chapters in the *Administrator's Guide*.

This section shows individual XQuery scripts for tasks such as creating and configuring App Servers. See "App Server Maintenance Operations" on page 55 for more examples on how to use the Admin functions to modify your App Servers and monitor their operation.

This section describes:

- Creating an App Server

4.3.1 Creating an App Server

The following script creates a new HTTP server in the `Sample` group, named `Sample-Server`, at port `8016`. The `application/` directory is the root, the `Sample-Database` is the content database and `Sample-Modules` is the modules database:

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Get the configuration :)
let $config := admin:get-configuration()

(: Get the group under which to create the App Server :)
let $groupid := admin:group-get-id($config, "Sample")

(: Add the new App Server to the configuration :)
let $server := admin:http-server-create(
  $config,
  $groupid,
  "Sample-Server",
  "application/",
  8016,
  admin:database-get-id($config, "Sample-Modules"),
  admin:database-get-id($config, "Sample-Database"))

(: Save the configuration :)
return admin:save-configuration($server);
```

4.3.2 Setting a URL Rewriter on an App Server

The following script sets the `rewriter.xqy` module to rewrite the URL from clients to an internal URL:

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Get the configuration containing the new App Server :)
let $config := admin:get-configuration()
```

```
(: Get the group for the App Server :)
let $groupid := admin:group-get-id($config, "Sample")

(: Set the URL rewriter :)
let $urlrewriter := admin:appserver-set-url-rewriter(
  $config,
  admin:appserver-get-id($config, $groupid, "Sample-Server"),
  "rewriter.xqy")

(: Save the configuration :)
return admin:save-configuration($urlrewriter)
```

4.3.3 Setting the Concurrent Request Limit on an App Server

The following script sets the Concurrent Request Limit for the “Sample-Server” App Server to 15:

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Sample")

return
  admin:save-configuration(
    admin:appserver-set-concurrent-request-limit(
    $config,
    admin:appserver-get-id($config, $groupid, "Sample-Server"),
    5))
```

4.3.4 Enabling Display Last-Login on an App Server

The following script enables Display Last Login on the “Sample-Server” App Server, using the “Last-Login” database:

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Sample")
let $config2 := admin:appserver-set-last-login(
  $config,
  admin:appserver-get-id($config, $groupid, "Sample-Server"),
  xdmp:database("Last-Login"))
```

```
return admin:save-configuration(
  admin:appserver-set-display-last-login(
    $config2,
    admin:appserver-get-id($config, $groupid, "Sample-Server"),
    fn:true()))
```

4.3.5 Creating and Configuring an App Server in a Single Transaction

In this example, we create and configure the same App Server as in the previous sections. Only the App Server is created and configured in a single transaction. As described in “Creating and Configuring Objects in a Single Transaction” on page 12, we use the `admin:appserver-get-id` function to obtain the App Server ID after creating the App Server to configure the newly created App Server in the same transaction. This example is abbreviated for the sake of simplicity and does not check for existing objects, as described in “Making Transactions Idempotent” on page 13.

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Get the configuration :)
let $config := admin:get-configuration()

(: Get the group under which to create the App Server :)
let $groupid := admin:group-get-id($config, "Sample")

(: Add the new App Server to the configuration :)
let $config := admin:http-server-create(
  $config,
  $groupid,
  "Sample-Server",
  "application/",
  8016,
  admin:database-get-id($config, "Sample-Modules"),
  admin:database-get-id($config, "Sample-Database"))

let $Sample-Server := admin:appserver-get-id(
  $config,
  $groupid,
  "Sample-Server")

(: Set the URL rewriter :)
let $config := admin:appserver-set-url-rewriter(
  $config,
  $Sample-Server,
  "rewriter.xqy")

let $config := admin:appserver-set-concurrent-request-limit(
  $config,
  $Sample-Server,
  15)
```

```
let $config := admin:appserver-set-last-login(
  $config,
  $Sample-Server,
  xdmp:database("Last-Login"))

let $config := admin:appserver-set-display-last-login(
  $config,
  $Sample-Server,
  fn:true())

return admin:save-configuration($config)
```

4.3.6 Deleting an App Server

The following script deletes the HTTP server created in “Creating and Configuring an App Server in a Single Transaction” on page 48:

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Get the configuration :)
let $config := admin:get-configuration()

(: Get the group from which to delete the App Server :)
let $groupid := admin:group-get-id($config, "Default")

(: Delete the App Server from the configuration :)
let $config := admin:appserver-delete(
  $config,
  admin:appserver-get-id($config, $groupid, "Sample-Server"))

(: Save the configuration :)
return admin:save-configuration($config)
```

4.4 Creating and Configuring Roles and Users

The general information on users and roles is provided in the Security Administration chapter in the *Administrator's Guide*.

This section shows an individual XQuery script that creates a new role, named `Temporary`, and two new users, named `Tom` and `Sue`. The new role is given the default collection, `testDocument`, and is assigned the privilege, `unprotected-collections`. Jim is given the default permission, `security(read)`. See “User Maintenance Operations” on page 70 for more examples on how to use the Security library functions to modify security objects and monitor their operation.

Note: Security objects must be created in the Security database. See “Executing Queries in Select Databases” on page 17 for techniques on how to execute queries in a database other than the one set for your App Server.

```
(: run this against the Security database :)

xquery version "1.0-ml";
import module namespace sec="http://marklogic.com/xdmp/security"
  at "/MarkLogic/security.xqy";

(: Create new role :)

sec:create-role(
  "Temporary",
  "Temporary worker access",
  ("filesystem-access"),
  (),
  ("testDocument"));

(: Now that the role is created, we can assign it a new privilege and
assign the role to new users :)

xquery version "1.0-ml";
import module namespace sec="http://marklogic.com/xdmp/security"
  at "/MarkLogic/security.xqy";

(: Add the 'Temporary' role to the list of roles with
'unprotected-collections' privilege :)

sec:privilege-add-roles(
  "http://marklogic.com/xdmp/privileges/unprotected-collections",
  "execute",
  ("Temporary")),

(: Create two new users with the role, 'Temporary'. :)

sec:create-user(
  "Jim",
  "Jim the temp",
  "newtemp",
  "Temporary",
  (xdmp:permission("security", "read")),
  ()),

sec:create-user(
  "Sue",
  "Sue the temp",
  "newtemp",
  "Temporary",
  (),
  ())
```

5.0 Server Maintenance Operations

This chapter describes how to use the Admin API to automate some of the operations you might want to perform on an existing MarkLogic Server configuration.

The main topics in this chapter are:

5.1 Group Maintenance Operations

The operations for creating and deleting groups are described in “Creating and Configuring Groups” on page 43. This section describes how to use the Admin API to automate some of the operations you might want to perform on an existing group.

The topics in this section are:

5.1.1 Enabling Auditing on a Group

The following script enables auditing of `user-configuration-change` and `user-role-addition` events on the “Default” group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Default")

(: Set user-configuration-change and user-role-addition events to be
audited in the "Default" group. :)

let $config := admin:group-enable-audit-event-type(
  $config,
  $groupid,
  ("user-configuration-change","user-role-addition"))

(: Enable auditing for the "Default" group. :)

return admin:save-configuration(
  admin:group-set-audit-enabled($config, $groupid, fn:true()))
```

5.1.2 Disabling Auditing on a Group

The following script disables auditing of all events on the "Default" group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Default")

return admin:save-configuration(
  admin:group-set-audit-enabled($config, $groupid, fn:false()))
```

5.1.3 Removing Events to be Audited on a Group

The following script disables auditing of `user-configuration-change` and `user-role-addition` events on the "Default" group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Default")

return admin:save-configuration(
  admin:group-disable-audit-event-type(
    $config,
    $groupid,
    ("user-configuration-change","user-role-addition")))
```

5.1.4 Adding a Namespace to a Group

The following script creates a new namespace, named “myprefix,” and adds it to the “Default” group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Default")

return admin:save-configuration(
  admin:group-add-namespace(
    $config,
    $groupid,
    admin:group-namespace("myprefix", "http://myuri/namespace")))
```

5.1.5 Returning the Namespace Settings on a Group

The following script returns the namespaces set for the “Default” group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Default")

return admin:group-get-namespaces($config, $groupid)
```

5.1.6 Deleting a Namespace from a Group

The following script deletes the “myprefix” namespace from the “Default” group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

declare namespace group = "http://marklogic.com/xdmp/group";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Default")

return admin:save-configuration(
  admin:group-delete-namespace(
    $config,
    $groupid,
    admin:group-get-namespaces($config, $groupid)
      [group:prefix eq "myprefix"]))
```

5.1.7 Returning the System Log Settings

The following script returns the current system log settings for the “Default” group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Default")

return
  admin:group-get-system-log-level($config, $groupid)

return (
  fn:concat("Log Level Setting: ",
    admin:group-get-system-log-level($config, $groupid)),
  fn:concat("Number of Log Files Kept: ",
    admin:group-get-keep-log-files($config, $groupid)),
  fn:concat("Log File Rotation Frequency: ",
    admin:group-get-rotate-log-files($config, $groupid)))
```

5.1.8 Resetting the System Log Settings

The following script resets the system log settings for the “Default” group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

let $groupid := admin:group-get-id(
  $config,
  "Default")

let $config := admin:group-set-system-log-level(
  $config,
  $groupid,
  "debug")

let $config := admin:group-set-keep-log-files(
  $config,
  $groupid,
  3)

return admin:save-configuration(
  admin:group-set-rotate-log-files(
    $config,
    $groupid,
    "friday"))
```

5.1.9 Creating a New Hourly Task

The following script creates an hourly scheduled task to invoke the `Scheduler_test.xqy` module every two hours and adds it to the "Default" group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

let $task := admin:group-hourly-scheduled-task(
  "Scheduler_test.xqy",
  "/Docs",
  2,
  30,
  xdmp:database("Sample-Database"),
  0,
  xdmp:user("Jim"),
  0)

let $config := admin:group-add-scheduled-task(
  $config,
  admin:group-get-id($config, "Default"),
  $task)

return admin:save-configuration($config)
```

5.1.10 Deleting all Scheduled Tasks from a Group

The following script deletes all of the scheduled tasks in the "Default" group:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $group := admin:group-get-id($config, "Default")
let $tasks := admin:group-get-scheduled-tasks($config, $group)

return admin:group-delete-scheduled-task($config, $group, $tasks)
```

5.2 App Server Maintenance Operations

The operations for creating and deleting App Servers are described in “Creating and Configuring App Servers” on page 45. This section describes how to use the Admin API to automate some of the operations you might want to perform on an existing App Server.

The topics are:

- Modifying the App Server Root for an HTTP App Server

- Changing the App Server Root and Cloning the Changed App Server
- Enabling SSL on an App Server
- Generating a Certificate Request
- Importing a Signed Certificate into the Database

5.2.1 Modifying the App Server Root for an HTTP App Server

The following example modifies an App Server configuration by changing its root from the relative path `myRoot` to the absolute path `/space/myRoot`.

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Default")

let $appserverid := admin:appserver-get-id(
  $config,
  $groupid,
  "Sample-Server")

let $config := admin:appserver-set-root(
  $config,
  $appserverid,
  "/space/myRoot")

return admin:save-configuration($config)
```

5.2.2 Changing the App Server Root and Cloning the Changed App Server

The following example does the same thing as the previous example (modifies the HTTP App Server root), and then it also takes that modified configuration and creates another App Server with the same settings but a different name.

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
    at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $groupid := admin:group-get-id($config, "Default")

let $appserverid := admin:appserver-get-id(
  $config,
  $groupid,
  "Sample-Server")

let $config := admin:appserver-set-root(
  $config,
  $appserverid,
  "/space/myRoot")

let $config := admin:appserver-copy(
  $config,
  $appserverid,
  (),
  "newHTTPServer",
  9021)

return admin:save-configuration($config)
```

This will result in both changes to the configuration, the change in root to `Sample-Server` and the `newHTTPServer` being created.

5.2.3 Enabling SSL on an App Server

The Configuring SSL on App Servers chapter in the *Security Guide* describes how to use the Admin UI to enable SSL on an App Server. The following sections describe how to enable SSL on an App Server using the Admin and PKI APIs:

- Creating a Certificate Template
- Generating a Certificate Request
- Importing a Signed Certificate into the Database
- Setting a Certificate Template on an App Server

5.2.3.1 Creating a Certificate Template

The following script creates a new certificate template, named `newTemplate`, and inserts it into the Security database:

```
xquery version "1.0-ml";
import module namespace pki = "http://marklogic.com/xdmp/pki"
  at "/MarkLogic/pki.xqy";

declare namespace x509 = "http://marklogic.com/xdmp/x509";
declare namespace ssl = "http://marklogic.com/xdmp/ssl";

let $x509 :=
  <x509:req>
    <x509:version>2</x509:version>
    <x509:subject>
      <x509:countryName>US</x509:countryName>
      <x509:stateOrProvinceName>CA</x509:stateOrProvinceName>
      <x509:localityName>San Carlos</x509:localityName>
      <x509:organizationName>MarkLogic</x509:organizationName>
      <x509:organizationalUnitName>
          Engineering
      </x509:organizationalUnitName>
      <x509:commonName>my.host.com</x509:commonName>
      <x509:emailAddress>user@marklogic.com</x509:emailAddress>
    </x509:subject>
    <x509:v3ext>
      <x509:nsCertType critical="false">SSL Server</x509:nsCertType>
      <x509:subjectAltName>
          DNS:marklogic.com, IP:127.0.0.1
      </x509:subjectAltName>
    </x509:v3ext>
  </x509:req>

let $options :=
  <pki:key-options xmlns="ssl:options">
    <key-length>2048</key-length>
  </pki:key-options>

return pki:insert-template(
  pki:create-template(
    "newTemplate",
    "Creating a new template",
    "rsa",
    $options,
    $x509))
```

5.2.3.2 Generating a Certificate Request

The following script generates a certificate request from the certificate template created in “Enabling SSL on an App Server” on page 57:

```
xquery version "1.0-ml";
import module namespace pki = "http://marklogic.com/xdmp/pki"
  at "/MarkLogic/pki.xqy";

let $tid := pki:template-get-id(
  pki:get-template-by-name("newTemplate"))

return pki:generate-certificate-request(
  $tid,
  (),
  "marklogic.com",
  "127.0.0.1")
```

5.2.3.3 Importing a Signed Certificate into the Database

The following script imports the PEM-encoded signed certificate from the Sample_cert.cer file into the Security database:

```
xquery version "1.0-ml";
import module namespace pki = "http://marklogic.com/xdmp/pki"
  at "/MarkLogic/pki.xqy";

pki:insert-signed-certificates(
  xdmp:document-get(
    "c:\SignedCertificates\Sample_cert.cer",
    <options xmlns="xdmp:document-get">
      <format>text</format>
    </options>))
```

5.2.3.4 Setting a Certificate Template on an App Server

The following script sets the PEM-encoded signed certificate in the Security database on the Sample-Server App Server:

```
xquery version "1.0-ml";
import module namespace pki = "http://marklogic.com/xdmp/pki"
  at "/MarkLogic/pki.xqy";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

let $appServer := admin:appserver-get-id(
  $config,
  admin:group-get-id($config, "Default"),
  "Sample-Server")

let $tid := pki:template-get-id(pki:get-template-by-name("mycert"))

return admin:save-configuration(
  admin:appserver-set-ssl-certificate-template(
    $config,
```

```
    $appServer,
    $tid))
```

5.3 Database Maintenance Operations

The operations for creating and deleting forests and databases are described in “Creating and Configuring Forests and Databases” on page 35. This section describes how to use the Admin API to automate some of the operations you might want to perform on an existing database and/or forest.

The topics in this section are:

- Creating a Database by Cloning an Existing Database Configuration
- Returning the Size of the Forests in a Database
- Deleting Element and Attribute Range Indexes
- Adding a Fragment Root to a Database
- Returning the Fragment Roots Set in a Database
- Deleting a Fragment Root from a Database
- Merging the Forests in a Database
- Scheduling Forest Backups
- Restoring from a Full Backup
- Scheduling Forest Backups
- Alerting the Administrator if the Forest Grows Beyond its Maximum Allowable Size
- Rotating Forest Update Types

5.3.1 Creating a Database by Cloning an Existing Database Configuration

The following example creates a new database with the exact same setup (including index settings, fragmentation, range indexes, and so on) as an existing database. It uses the `admin:database-copy` function to clone the database.

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
    at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $config := admin:database-copy(
  $config,
  xdmp:database("myOldDatabase"),
  "myNewDatabase")

return admin:save-configuration($config)
```

After running this XQuery program, a new database configuration named `myNewDatabase` is created with the same settings as the database named `myOldDatabase`. Note that this database will not have any forests attached to it, as forests can only be attached to a single database.

5.3.2 Returning the Size of the Forests in a Database

The following script returns the size of all of the forests in the "Sample-Database" database:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

declare namespace forest = "http://marklogic.com/xdmp/status/forest";

(: Get all of the forests in the "Sample-Database database. :)
for $forests in xdmp:forest-status(
  xdmp:database-forests(xdmp:database("Sample-Database")))

(: Get the remaining disk space for each forest device. :)
  let $space := $forests//forest:device-space

(: Get the name of each forest. :)
  let $f_name := $forests//forest:forest-name

(: The size of a forest is the sum of its stand sizes. :)
  for $stand in $forests//forest:stands
    let $f_size := fn:sum($stand/forest:stand/forest:disk-size)

(: Return the name and size for each forest and remaining
   disk space. :)
return fn:concat(
  "Forest Name: ",
  fn:string($f_name),
  "  Forest Size: ",
  fn:string($f_size),
  "  Disk Space Left: ",
  $space)
```

5.3.3 Deleting Element and Attribute Range Indexes

The following script deletes the range element index and the range element attribute index created in the previous two examples from the "Sample-Database" database:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $dbid := xdmp:database("Sample-Database")

let $elem-rangespec := admin:database-range-element-index(
  "date",
```

```
  "/myco/employees",
  "birthday",
  "",
  fn:false() )

let $elem-attr-rangespec :=
  admin:database-range-element-attribute-index(
    "date",
    "/myco/employees",
    "Personal",
    "",
    "birthday hire-date",
    "",
    fn:false())

let $config2 := admin:database-delete-range-element-index(
  $config,
  $dbid,
  $elem-rangespec)

return admin:save-configuration(
  admin:database-delete-range-element-attribute-index(
    $config2,
    $dbid,
    $elem-attr-rangespec))
```

5.3.4 Adding a Fragment Root to a Database

The following script adds a fragment root specification for the “TITLE” element to the “Sample-Database” database:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $dbid := xdmp:database("Sample-Database")
let $fragspec := admin:database-fragment-root(
  "/shakespeare/plays",
  "TITLE")

return   admin:save-configuration(
  admin:database-add-fragment-root(
    $config,
    $dbid,
    $fragspec))
```

5.3.5 Returning the Fragment Roots Set in a Database

The following script returns the fragment root specifications set in the “Sample-Database” database:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

return admin:database-get-fragment-roots(
  $config,
  xdmp:database("Sample-Database"))
```

5.3.6 Deleting a Fragment Root from a Database

The following script deletes the fragment root specification for the “TITLE” element from the “Sample-Database” database:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
let $dbid := xdmp:database("Sample-Database")
let $fragspec := admin:database-fragment-root(
  "/shakespeare/plays",
  "TITLE")

return admin:save-configuration(
  admin:database-delete-fragment-root($config, $dbid, $fragspec))
```

5.3.7 Merging the Forests in a Database

The following script merges four forests in the database with the specification to not leave any single stand larger than 50MB. For example, if the forest size is 180 MB, this script would merge the content into four stands:

```
xquery version "1.0-ml";

xdmp:merge(
  <options xmlns="xdmp:merge">
    <merge-max-size>50</merge-max-size>
    <forests>
      <forest>{xdmp:forest("myforest1")}</forest>
      <forest>{xdmp:forest("myforest2")}</forest>
      <forest>{xdmp:forest("myforest3")}</forest>
      <forest>{xdmp:forest("myforest4")}</forest>
    </forests>
  </options>)
```

5.3.8 Scheduling Forest Backups

The following script establishes a backup schedule for all of the forests in the “Sample-Database” database:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Get the configuration :)
let $config := admin:get-configuration()

(: Set up the backup elements. :)
let $backup := admin:forest-weekly-backup(
  "/backup-dir",
  "friday",
  xs:time("23:00:00"))

(: Get all of the forests in the "Sample-Database" database. :)
for $forest in admin:database-get-attached-forests(
  $config,
  xdmp:database("Sample-Database"))

(: Add the backup elements to each forest configuration. :)
return admin:forest-add-backup(
  $config,
  $forest,
  $backup)
```

5.3.9 Alerting the Administrator if the Forest Grows Beyond its Maximum Allowable Size

The following script checks the amount of disk space used by the forests in the "Sample-Database" database against the available disk space on the forest devices. If the size of a forest surpasses its maximum allowable size, an event is logged and an email is sent to `urgent@mycompany.com`. Such a script could be executed periodically using the scheduling features described in Scheduling Tasks in the *Administrator's Guide*.

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

declare namespace forest = "http://marklogic.com/xdmp/status/forest";

(: Get all of the forests in the "Sample-Database" database. :)
   for $forests in xdmp:forest-status(
  xdmp:database-forests(xdmp:database("Sample-Database")))

(: Get the remaining disk space for each forest device. :)
  let $space := $forests//forest:device-space

(: Get the name of each forest. :)
  let $f_name := $forests//forest:forest-name

(: The size of a forest is the sum of its stand sizes. :)
  for $stand in $forests//forest:stands
     let $f_size := fn:sum($stand/forest:stand/forest:disk-size)
```

```
(: The maximum size of the forest is calculated by multiplying the
   size of the forest by 3 and comparing that value against the
   available disk space - 1000 MB. If the forest grows beyond its
   maximum size, log the event and send an email alert to the
   administrator. :)

  return

    if (($f_size * 3) > ($space - 1000))

    then (xdmp:log(
      fn:concat($f_name, " forest space check status: Failed"),
      "emergency"),
      xdmp:email(
         <em:Message
         xmlns:em="URN:ietf:params:email-xml:"
         xmlns:rf="URN:ietf:params:rfc822:">
           <rf:subject>Forest Space Check Failure</rf:subject>
           <rf:from>
             <em:Address>
               <em:name>MarkLogic Server</em:name>
               <em:adrs>no_return@mycompany.com</em:adrs>
             </em:Address>
           </rf:from>
           <rf:to>
             <em:Address>
               <em:name>System Administrator</em:name>
               <em:adrs>urgent@mycompany.com</em:adrs>
             </em:Address>
           </rf:to>
           <em:content xml:space="preserve">
             {fn:concat($f_name, " forest space check status: Failed")}
           </em:content>
         </em:Message>))

    else (xdmp:log(
         fn:concat($f_name, " forest space check status: Passed")))
```

5.3.10 Rotating Forest Update Types

As described in Making a Forest Delete-Only, in the *Administrator's Guide*, there may be circumstances in which you have multiple forests in a database and you want to manage which forests change. The following script checks the status of each forest's update type in the database, "Sample-Database," and changes any forest with an update type of `all` to `delete-only` and any forest with an update type of `delete-only` to `all`. Such a script could be executed periodically as a scheduled task.

Warning Applications that use a database containing `delete-only` forests must specify an updateable forest in the `xdmp:document-insert` function. Otherwise an insert to a document on a `delete-only` forest returns an error.

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

declare namespace forest = "http://marklogic.com/xdmp/status/forest";

(: Get the configuration :)
let $config := admin:get-configuration()

(: Update the configuration :)
let $new-config :=

(: Get all of the forests in the "Sample-Database" database. :)
for $forests in xdmp:forest-status(
  xdmp:database-forests(xdmp:database("Sample-Database")))

(: Get the id of each forest. :)
  let $f_id := $forests//forest:forest-id

(: Get the name of each forest. :)
  let $f_name := $forests//forest:forest-name

(: Get the updates allowed status of each forest. :)
  let $f_updates := $forests//forest:updates-allowed

(: Reset the update type for the 'delete-only' forests to 'all' and the
'all' forests to 'delete-only'. Add a log message for each update. :)

return
  if ($f_updates eq "all")
    then (
      xdmp:log(fn:concat(
         "Setting ",
         $f_name,
         " forest with updates set to ",
         $f_updates,
         " to delete-only")),
      xdmp:set(
         $config,
         admin:forest-set-updates-allowed(
            $config,
            $f_id,
            "delete-only")) )

    else (
      xdmp:log(fn:concat(
         "Setting ",
         $f_name,
         " forest with updates set to ",
         $f_updates,
         " forest to 'all'")),

      xdmp:set($config, admin:forest-set-updates-allowed(
         $config,
```

```
            $f_id,
            "all"))
        )

    return admin:save-configuration($config)
```

The example script below does the same thing as the above script using a different approach. The script above creates and sets the configuration once for each forest. The script below uses a function that returns a single configuration for all of the forests in the database. Though the script below contains more complex logic, it represents the type of script you would want to use in a production environment:

```
xquery version "1.0-ml";

declare namespace forest = "http://marklogic.com/xdmp/status/forest";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

(: Function accepts the empty configuration sequence and list of
forests and returns the final configuration. :)

declare function local:change-update-types(
  $config as element(configuration)*,
  $forests as xs:unsignedLong*) as element(configuration)
{

(: The first time through the loop, get the configuration. After that,
continue to build on the modified configuration. :)
if (fn:empty($config))
   then xdmp:set($config, admin:get-configuration())
   else (),

(: Reset the update type for the 'delete-only' forests to 'all' and
   the 'all' forests to 'delete-only'. When complete, return the
   configuration. :)

(: Determine whether there are remaining forests. If not, return
   the final configuration. :)
if (fn:count($forests) lt 1)
   then ($config)

(: Convert the update type for the next forest in the sequence. Add a
log message for each conversion. :)
   else (
    let $f_updates :=
           xdmp:forest-status($forests[1])//forest:updates-allowed
    let $name := xdmp:forest-name($forests[1])
    return (
       if ($f_updates eq "all")

          then (
             xdmp:log(fn:concat(
```

```
                  "Setting ",
                  $name,
                  " forest with updates set to ",
                  $f_updates,
                  " to delete-only")),

               xdmp:set($config, admin:forest-set-updates-allowed(
                  $config,
                  $forests[1],
                  "delete-only")))

            else (
               xdmp:log(fn:concat(
                  "Setting ",
                  $name,
                  " forest with updates set to ",
                  $f_updates, " forest to 'all'")),

               xdmp:set($config, admin:forest-set-updates-allowed(
                  $config,
                  $forests[1],
                  "all"))),

(: Function calls itself for each remaining forest in the sequence. :)

    local:change-update-types($config, $forests[2 to last()])
    )
  )
};

(: Main :)

(: Define $config as an empty sequence. The function uses this as a
flag to determine whether or not to get the initial configuration. :)
let $config := ()

(: Obtain the id of each forest in the database. :)
let $forests :=
  xdmp:database-forests(xdmp:database("Sample-Database"))

(: Call the change-update-types function and set the returned
configuration. :)
return admin:save-configuration(
  local:change-update-types($config, $forests))
```

5.4 Host Maintenance Operations

This section describes how to use the Admin API to automate some of the operations you might want to perform on an existing host.

The topics in this section are:

- Returning the Status of the Host

5.4.1 Returning the Status of the Host

The following script returns the current status of the local host:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

return xdmp:host-status(
  admin:host-get-id($config, xdmp:host-name()))
```

5.4.2 Returning the Time Host was Last Started

The following script returns the time the local host was last started:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

declare namespace host = "http://marklogic.com/xdmp/status/host";

let $config := admin:get-configuration()

for $i in (xdmp:host-status(
  admin:host-get-id(
  $config,
  xdmp:host-name()))//host:last-startup
return
  fn:string($i)
```

5.4.3 Restarting MarkLogic Server on all Hosts in the Cluster

The following script restarts MarkLogic Server on all of the hosts in the cluster that contains the host invoking the script (including the invoking host):

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

declare namespace host="http://marklogic.com/xdmp/status/host";

let $hostids := for $id in xdmp:host-status(xdmp:host())
  /host:hosts//host:host/host:host-id
    return fn:data($id)

return admin:restart-hosts($hostids)
```

5.5 User Maintenance Operations

This section describes the user and role maintenance operations that make use of the functions in the `security.xqy` library module. All calls to functions in the `security.xqy` library module must be executed against the Security database. For information on how to execute queries to databases other than the one set for your App Server, see “Executing Queries in Select Databases” on page 17.

The topics in this section are:

5.5.1 Removing all Users with a Specific Role

The following script removes all users assigned the role, `Temporary`:

```
(: run this against the Security database :)

xquery version "1.0-ml";
import module "http://marklogic.com/xdmp/security"
  at "/MarkLogic/security.xqy";

for $user in fn:data(//sec:user-name)
  return
    if (fn:matches(sec:user-get-roles($user), "Temporary"))
       then (fn:concat("Removed:  ", $user), sec:remove-user($user))
       else ()
```

5.5.2 Removing a Specific Role, if Present

The following script removes the role, `Temporary`, if present:

```
xquery version "1.0-ml";
import module "http://marklogic.com/xdmp/security"
  at "/MarkLogic/security.xqy";

for $role in fn:data(//sec:role-name)
  return
    if (fn:matches($role, "Temporary"))
      then (fn:concat("Removed:  ", $role),
            sec:remove-role("Temporary"))
      else ()

return admin:save-configuration(
  admin:database-add-range-element-attribute-index(
    $config,
    $dbid,
    $rangespec))
```

5.5.3 Retrieving the Last-Login Information

The `xdmp:user-last-login` function returns an XML node with information about the last successful login, last unsuccessful login, and the number of unsuccessful login attempts for a user. If there is no last-login database configured, then the function returns the empty sequence.

The following is a very simple program that demonstrates how to use this information to add a message to your application. It uses the display-last-login field to determine whether to display anything.

```
xquery version "1.0-ml";
declare namespace ll="http://marklogic.com/xdmp/last-login";

let $last := xdmp:user-last-login()
return
( if ($last/ll:display-last-login/text() eq "true")
    then ( fn:concat("You are logged in as the user '",
       xdmp:get-current-user(),
       "' (", fn:data($last/ll:user-id), ") ",
       "who last successfully logged in on ",
       $last/ll:last-successful-login, ".") )
    else ()  )
```

This script returns output similar to the following:

```
You are logged in as the user 'Jim' (8936413450950930 63) who last
successfully logged in on 2008-07-15T16:13:54-07:00.
```

5.6 Backing Up and Restoring

This section provides examples using the Admin API for backup and restore tasks using full and incremental backups. This section includes the following parts:

- Full Backup of a Database
- Incremental Backup of a Database
- Restoring from a Full Backup
- Restoring from an Incremental Backup
- Validating an Incremental Backup

5.6.1 Full Backup of a Database

The following script immediately does a full backup all of the forests in the “Sample-Database” database to the `/backup-dir` directory:

```
xquery version "1.0-ml";
```

```
xdmp:database-backup(
  xdmp:database-forests(xdmp:database("Sample-Database")),
  "c:/backup-dir")

=>

437302857479804813287
```

5.6.2 Incremental Backup of a Database

The following script does an immediate incremental backup of two forests (`11183608861595735720` and `898513504988507762)` to the `/backups/Data` directory:

```
xquery version "1.0-ml";

xdmp:database-incremental-backup(
  (11183608861595735720,898513504988507762),
  "/backups/Data")

=>

33030877979801813489
```

5.6.3 Restoring from a Full Backup

The following script restores the “Sample-Database” database from the backup taken on 2014-09-15:

```
xquery version "1.0-ml";

xdmp:database-restore(
  xdmp:database-forests(xdmp:database("Sample-Database")),
  "c:/fullbackup-dir",
xs:dateTime("2014-09-15T11:30:51.33885-05:00"),fn:false()

=>

33030877979801813489
```

5.6.4 Restoring from an Incremental Backup

This script restores a database from the incremental backup done on 2014-09-17.

```
xquery version "1.0-ml";

xdmp:database-restore(
xdmp:database-forests(xdmp:database("Documents")),
"c:/test-backup", xs:dateTime("2014-09-17T11:35:51.33882-07:00"),
fn:false()," ",fn:true(), "c:/test-backup")

=>
```

```
3504087797980181448 9
```

5.6.5 Validating an Incremental Backup

This script validates that the specified list of forests (`11183608861595735720` and `898513504988507762`) can be backed up to the backup data directory (`/backups/Data2`):

```
xquery version "1.0-ml";

xdmp:database-incremental-backup-validate((11183608861595735720,898513
504988507762),
  "/backups/Data", fn:false(), "/backups/Data2", fn:true())

=>

<bp:backup-plan
   xsi:schemaLocation="http://marklogic.com/xdmp/backup-plan
backup-plan.xsd"
   xmlns:bp="http://marklogic.com/xdmp/backup-plan"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance";>
  <bp:forest>
    <bp:forest-name>Documents</bp:forest-name>
    <bp:forest-id>9157142760003704384</bp:forest-id>
    <bp:forest-status>okay</bp:forest-status>
    <bp:directory-path>/backups/Data</bp:directory-path>
    <bp:directory-status>okay</bp:directory-status>
    <bp:action>incremental-backup</bp:action>
    <bp:incremental-backup>true</bp:incremental-backup>
    <bp:incremental-backup-path>/backups/Data2</bp:incremental-backup-
path>
    <bp:journal-archiving>true</bp:journal-archiving>
    <bp:journal-archive-path>/tmp</bp:journal-archive-path>
    <bp:journal-archive-path-status>okay</bp:journal-archive-path-stat
us>
    <bp:journal-archive-lag-limit>15</bp:journal-archive-lag-limit>
    <bp:journal-archive-lag-limit-status>okay</bp:journal-archive-lag-
limit-status>
  </bp:forest>
</bp:backup-plan>
```

6.0 Scripting Content Processing Framework (CPF) Configuration

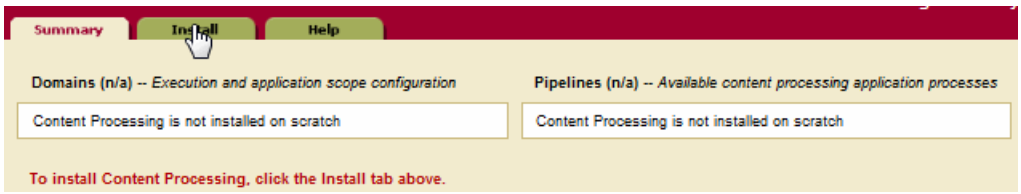
The MarkLogic Server Content Processing Framework (CPF) is described in detail in the *Content Processing Framework Guide*. This chapter describes how to use the CPF API to programmatically configure CPF. The main topics in this chapter are:

- General Procedure for Configuring CPF
- Creating CPF Pipelines
- Inserting Existing CPF Pipelines
- Creating a CPF Domain
- Configuring a CPF Restart Trigger

Note: All queries must be executed on the database that stores your triggers. Though MarkLogic Server provides a preconfigured Triggers database that contains the out-of-the box triggers, the examples in this chapter assume you are configuring your own triggers database. If you decide to use the triggers from the preconfigured Triggers database, you only need to create your domain, as described in “Creating a CPF Domain” on page 77.

6.1 General Procedure for Configuring CPF

When using the Admin Interface, you can select the Install tab in the Content Processing Summary page to “install” CPF:



The term “install” is a bit of a misnomer. What really happens is that MarkLogic Server installs the out-of-the-box pipelines, creates a restart trigger, creates a default domain, and assigns some default pipelines to the default domain. When doing this, the Admin Interface makes certain assumptions about how to configure CPF. One of the reasons for using the CPF API to configure CPF is that you can control which pipelines are installed and configured for a domain, as well as the restart trigger user, permissions, and evaluation context.

This section describes the general procedure for configuring CPF on a triggers database. The general steps are:

- Create CPF pipelines, as described in Creating CPF Pipelines. If you are configuring CPF with existing pipelines, insert them into the triggers database, as described in Inserting Existing CPF Pipelines.

- Create CPF domains, as described in Creating a CPF Domain.
- Configure a CPF restart trigger, as described in Configuring a CPF Restart Trigger.

6.2 Creating CPF Pipelines

CPF Pipelines are described in detail in Understanding and Using Pipelines in the *Content Processing Framework Guide*. This section describes how to use the CPF API to create a Status Change Handling pipeline, which is required for most CPF operations.

The following query is executed against the triggers database used by the content database.

```
xquery version "1.0-ml";

import module namespace dom = "http://marklogic.com/cpf/domains"
  at "/MarkLogic/cpf/domains.xqy";

import module namespace p = "http://marklogic.com/cpf/pipelines"
  at "/MarkLogic/cpf/pipelines.xqy";

let $success := xs:anyURI("http://marklogic.com/states/replicated")
let $failure := xs:anyURI("http://marklogic.com/states/error")

return (

(: Create the Status Change Handling Pipeline :)

p:create(
  "Status Change Handling",
  "Status Change Handling Pipeline",
  p:action("/MarkLogic/cpf/actions/success-action.xqy", (), ()),
  p:action("/MarkLogic/cpf/actions/failure-action.xqy", (), ()),

  (p:status-transition(
    "created",
    "New document entering the system: kick it into the appropriate
initial state. If is has an initial state, go to that state. If it
doesn't, go to the standard initial state and set the initial
timestamp. ",
    xs:anyURI("http://marklogic.com/states/initial"),
    (),
    100,
    p:action("/MarkLogic/cpf/actions/set-updated-action.xqy", (), ()),
      (p:execute(
         p:condition(
           "/MarkLogic/cpf/actions/renamed-links-condition.xqy",
           (),
           () ),
         p:action(
           "/MarkLogic/cpf/actions/link-rename-action.xqy",
           (),
           () ),
        () ),
```

```
        p:execute(
           p:condition(
             "/MarkLogic/cpf/actions/existing-state-condition.xqy",
             (),
             () ),
           p:action(
             "/MarkLogic/cpf/actions/touch-state-action.xqy",
             (),
             () ),
           () )
      )
    ),

    p:status-transition(
      "deleted",
      "Clean up dangling links and dependent documents from deleted
  documents. ",
      (),
      (),
      100,
      p:action(
        "/MarkLogic/cpf/actions/link-coherency-action.xqy",
        (),
        () ),
      ()
    ),

    p:status-transition(
      "updated",
      "Update the document time stamp and shift to the updated state. ",
      xs:anyURI("http://marklogic.com/states/updated"),
      (),
      100,
      p:action("/MarkLogic/cpf/actions/set-updated-action.xqy", (), ()),
      ()
    ) ),
    ()
  ) )
```

6.3 Inserting Existing CPF Pipelines

If you have pipeline configuration in the form of an XML file, then you can use the `p:insert` function to insert the pipeline into a triggers database. For example, the pipelines shipped with MarkLogic Server are located in the `/MarkLogic/Installer` directory. This section describes how to use the `p:insert` function to insert the Flexible Replication and the Status Change Handling pipelines into a triggers database.

Note: The Flexible Replication and the Status Change Handling pipelines are the two pipelines required to configure flexible replication. They must be inserted into a triggers database and assigned to a domain before using the `flexrep` API functions

to configure flexible replication, as described in "Scripting Flexible Replication Configuration" on page 79.

The following query is executed against the triggers database used by the content database.

```
xquery version "1.0-ml";

import module namespace dom = "http://marklogic.com/cpf/domains"
  at "/MarkLogic/cpf/domains.xqy";

import module namespace p = "http://marklogic.com/cpf/pipelines"
  at "/MarkLogic/cpf/pipelines.xqy";

let $flexrep-pipeline :=
  xdmp:document-get("Installer/flexrep/flexrep-pipeline.xml")

let $status-pipeline :=
  xdmp:document-get("Installer/cpf/status-pipeline.xml")

return (
  p:insert($flexrep-pipeline),
  p:insert($status-pipeline) )
```

6.4 Creating a CPF Domain

CPF Domains are described in detail in Understanding and Using Domains in the *Content Processing Framework Guide*. This section describes how to create a new CPF domain. If you have already created the pipelines to be used by the domain, then you can specify them in your `dom:create` function. Otherwise you can add the pipelines to the domain by means of the `dom:add-pipeline` or `dom:set-pipelines` function.

The following query creates a domain named Replication Content. The scope of the domain is the root directory of the content database that uses the domian. The evaluation context is the root directory of the Modules database. The pipelines assigned to the domain are Flexible Replication and the Status Change Handling. The domain can be read and executed by the user, `app-user`. This query is executed against the triggers database used by the content database.

```
xquery version "1.0-ml";

import module namespace dom = "http://marklogic.com/cpf/domains"
  at "/MarkLogic/cpf/domains.xqy";

import module namespace p = "http://marklogic.com/cpf/pipelines"
  at "/MarkLogic/cpf/pipelines.xqy";

dom:create(
    "Replicated Content",
    "Handle replicated documents",
    dom:domain-scope(
       "directory",
       "/",
```

```
        "infinity"),
      dom:evaluation-context(
        xdmp:database("Modules"),
        "/" ),
      (p:get("Status Change Handling")/p:pipeline-id,
       p:get("Flexible Replication")/p:pipeline-id),
      (xdmp:permission('app-user', 'read'),
       xdmp:permission('app-user', 'execute') )
)
```

6.5 Configuring a CPF Restart Trigger

CPF is designed so that, if the server or database goes offline, it will pick up where it left off. In order to resume from where it left off, CPF needs to have a restart trigger configured on the triggers database used by the content database. There is only one restart trigger for each triggers database.

After you have created your piplelines and domains, call the `dom:configuration-create` function to configure your database with a restart trigger. Only do this once, as there is only one restart trigger per triggers database. The restart trigger needs to associated with a particular user, an evaluation context, and a default domain. Unlike other CPF triggers that obtain their evaluation context from a domain, the restart trigger obtains its execution context from the CPF configuration. All the restarted actions are executed as the restart-user. The restart user should have the `cpf-restart` role, as well as all of the permissions and privileges that normal users have on the documents.

The following query configures a restart trigger. The restart user is `CPFuser`, the default domain is Replicated Content, and the evaluation context is the root directory of the Modules database. This query is executed against the triggers database used by the content database.

```
xquery version "1.0-ml";

import module namespace dom = "http://marklogic.com/cpf/domains"
  at "/MarkLogic/cpf/domains.xqy";
(: only create a single restart trigger per triggers database as
   it applies to all domains :)

dom:configuration-create(
  "CPFuser",
  dom:evaluation-context( xdmp:database("Modules"), "/" ),
  fn:data(dom:get("Replicated Content")/dom:domain-id),
  (xdmp:permission('app-user', 'read'),
   xdmp:permission('app-user', 'execute') ) )
```

7.0 Scripting Flexible Replication Configuration

This chapter describes how to use the XQuery and REST APIs to configure flexible replication. For details on flexible replication, see the *Flexible Replication Guide*. The procedures are:

7.1 Configuring Push Replication using the XQuery API

This section describes how to use the XQuery API to configure a Master database to push replication updates to the Replica database. The procedures are:

7.1.1 Preliminary Configuration Procedures

The procedures described in this chapter assume you have done the following configuration on MarkLogic Server:

1. Create three new forests:

   - `MasterForest`
   - `MyTriggersForest`
   - `ReplicaForest`

   The MasterForest and MyTriggersForest must be on the same server. For details on how to create forests, see “Creating Forests and Databases” on page 36.

2. Create three new databases:

   - `Master`
   - `MyTriggers`
   - `Replica`

   The Master and MyTriggers databases must be on the same server. For details on how to create databases, see “Creating Forests and Databases” on page 36.

3. Attach the forests to the databases:

   - `MasterForest` to `Master`
   - `MyTriggersForest` to `MyTriggers`
   - `ReplicaForest` to `Replica`

   For details on how to attach forests to databases, see "Attaching Forests to Databases" on page 36.

4. Configure the Master database to use the MyTriggers database as its Triggers Database:

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

let $config := admin:database-set-triggers-database(
  $config, xdmp:database("Master"),
  xdmp:database("MyTriggers"))

return admin:save-configuration($config)
```

5. For the MyTriggers database:

   - Insert a Flexible Replication pipeline and a Status Change Handling pipeline, as described in "Inserting Existing CPF Pipelines" on page 76.
   - Create a CPF domain, named Replicated Content, that uses the Flexible Replication and Status Change Handling pipelines, as described in "Creating a CPF Domain" on page 77.

6. Create two HTTP App Servers with the following settings

Server Name	Root	Port	Database
Master-flexrep	FlexRep	8010	Master
Replica-flexrep	FlexRep	8011	Replica

   For details on how to use the Admin API to create App Servers, see "Creating an App Server" on page 46.

7.1.2 Configuring the Master Database

The `flexrep:configure-database` function creates the indexes needed by the Master database for CPF based replication.

```
xquery version "1.0-ml";

import module namespace flexrep =
  "http://marklogic.com/xdmp/flexible-replication"
  at "/MarkLogic/flexrep.xqy";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

let $config := flexrep:configure-database(
  $config,
  xdmp:database("Master"))

return admin:save-configuration($config)
```

7.1.3 Creating a Replication Configuration Element

Most of the Flexible Replication API functions require a replication configuration element for each replicated domain. You create a replication configuration element by calling the `flexrep:configuration-create` function and insert it into the database by calling the `flexrep:configuration-insert` function.

The following query creates a new replication configuration element for the Replication Content domain and inserts it into the database. This query is executed against the Master database, so an `xdmp:eval` function is used to obtain the domain ID from the MyTriggers database.

```
xquery version "1.0-ml";

import module namespace flexrep =
  "http://marklogic.com/xdmp/flexible-replication"
    at "/MarkLogic/flexrep.xqy";

(: Obtain the id of the replicated CPF domain from the
   Triggers database. :)

let $domain:= xdmp:eval(
    'xquery version "1.0-ml";
    import module namespace dom = "http://marklogic.com/cpf/domains"
      at "/MarkLogic/cpf/domains.xqy";
    fn:data(dom:get( "Replicated Content" )//dom:domain-id)',
    (),
    <options xmlns="xdmp:eval">
      <database>{xdmp:database("MyTriggers")}</database>
    </options>)

(: Create a replication configuration for the Replicated
   Content domain. :)

let $cfg := flexrep:configuration-create($domain)
```

```
(: Insert the replication configuration element into the database. :)

return flexrep:configuration-insert($cfg)
```

7.1.4 Creating a Replication Target

This section describes how to use the `flexrep:target-create` function to create a replication target. The following query is executed against the Master database, so an `xdmp:eval` function is used to obtain the domain id from the MyTriggers database.

```
xquery version "1.0-ml";

import module namespace flexrep =
  "http://marklogic.com/xdmp/flexible-replication"
    at "/MarkLogic/flexrep.xqy";

(: Obtain the id of the replicated CPF domain from the
   Triggers database. :)

let $domain:= xdmp:eval(
    'xquery version "1.0-ml";
    import module namespace dom = "http://marklogic.com/cpf/domains"
      at "/MarkLogic/cpf/domains.xqy";
    fn:data(dom:get( "Replicated Content" )//dom:domain-id)',
    (),
    <options xmlns="xdmp:eval">
      <database>{xdmp:database("MyTriggers")}</database>
    </options>)

(: Obtain the replication configuration. :)

let $cfg := flexrep:configuration-get($domain, fn:true())

(: Specify the HTTP options for the replication target. :)

let $http-options :=
  <flexrep:http-options
    xmlns:flexrep="http://marklogic.com/xdmp/flexible-replication">
    <http:authentication xmlns:http="xdmp:http">
      <http:username>admin</http:username>
      <http:password>admin</http:password>
    </http:authentication>
    <http:client-cert xmlns:http="xdmp:http"/>
    <http:client-key xmlns:http="xdmp:http"/>
    <http:pass-phrase xmlns:http="xdmp:http"/>
  </flexrep:http-options>

(: Create the replication target. :)

let $cfg := flexrep:target-create(
  $cfg,
  "Replica",
  "http://localhost:8011/",
```

```
  60,
  300,
  10,
  fn:true(),
  $http-options,
  fn:false(),
  (),
  () )

(: Insert the changes to the replication configuration. :)

return flexrep:configuration-insert($cfg)
```

7.1.5 Creating a Push Replication Scheduled Task

This section describes how to use the scheduler functions to create a scheduled replication push task. The following query is executed against the Master database.

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

(: Define a "minutely" scheduled task to push replication
   each minute. :)

let $task := admin:group-minutely-scheduled-task(
  "/MarkLogic/flexrep/tasks/push.xqy",
  "Modules",
  1,
  xdmp:database("Master"),
  0,
  xdmp:user("admin"),
  admin:host-get-id($config, xdmp:host-name())
)

(: Add the scheduled task to the Default group. :)

let $config:= admin:group-add-scheduled-task(
  $config,
  admin:group-get-id($config, "Default"),
  $task)

return admin:save-configuration($config)
```

7.2 Configuring Pull Replication using the XQuery API

This section describes how to use the XQuery API to configure a Replica database to retrieve replication updates from the Master database. The procedures are:

- Disabling Push Replication on the Master Database

- Creating a Pull Replication Configuration
- Creating a Pull Replication Scheduled Task

7.2.1 Disabling Push Replication on the Master Database

Before configuring Pull Replication on the Replica database, you must disable Push Replication on the Master database. The following query is executed against the Master database.

```
xquery version "1.0-ml";

import module namespace flexrep =
  "http://marklogic.com/xdmp/flexible-replication"
    at "/MarkLogic/flexrep.xqy";

(: Obtain the id of the replicated CPF domain from the
   Triggers database. :)

let $domain:= xdmp:eval(
    'xquery version "1.0-ml";
    import module namespace dom = "http://marklogic.com/cpf/domains"
      at "/MarkLogic/cpf/domains.xqy";
    fn:data(dom:get( "Replicated Content" )//dom:domain-id)',
    (),
    <options xmlns="xdmp:eval">
      <database>{xdmp:database("MyTriggers")}</database>
    </options>)

(: Obtain the replication configuration. :)
let $cfg := flexrep:configuration-get($domain, fn:true())

(: Obtain the replication target id. :)
let $target-id := flexrep:configuration-target-get-id(
  $cfg,
  "Replica")

(: Disable Push Replication on the replication target. :)
let $cfg := flexrep:configuration-target-set-enabled(
  $cfg,
  $target-id,
  fn:false())

(: Insert the replication configuration element into the database. :)
return flexrep:configuration-insert($cfg)
```

7.2.2 Creating a Pull Replication Configuration

Pull Replication requires a pull replication configuration element for each replicated domain. You create a pull replication configuration element by calling the `flexrep:pull-create` function and insert it into the database by calling the `flexrep:pull-insert` function.

The following query is executed against the Replica database, which is most likely running on a different server than the Master database. As a consequence, you will need to first obtain the domain ID from the master database and the target ID from the master's triggers database.

```
xquery version "1.0-ml";

import module namespace flexrep =
  "http://marklogic.com/xdmp/flexible-replication"
    at "/MarkLogic/flexrep.xqy";

(: Specify the id of the replicated CPF domain obtained from the
   Master's Triggers database. :)

let $domain:= 9535475951259984368

(: Specify the id of the replication target obtained from the
   Master database. :)

let $target-id := 18130470845627037840

(: Specify the HTTP options for the replication target. :)

let $http-options :=
  <flexrep:http-options
    xmlns:flexrep="http://marklogic.com/xdmp/flexible-replication">
    <http:authentication xmlns:http="xdmp:http">
      <http:username>admin</http:username>
      <http:password>admin</http:password>
    </http:authentication>
    <http:client-cert xmlns:http="xdmp:http"/>
    <http:client-key xmlns:http="xdmp:http"/>
    <http:pass-phrase xmlns:http="xdmp:http"/>
  </flexrep:http-options>

let $pullconfig := flexrep:pull-create(
  "Master",
  $domain,
  $target-id,
  "http://localhost:8010/",
  $http-options)

(: Insert the pull configuration into the Replica database. :)

return flexrep:pull-insert($pullconfig)
```

7.2.3 Creating a Pull Replication Scheduled Task

This section describes how to use the scheduler functions to create a scheduled replication pull task. The following query is executed against the Replica database.

```
xquery version "1.0-ml";
```

```
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()

(: Define a "minutely" scheduled task to pull updates from the
   Master database each minute. :)

let $task := admin:group-minutely-scheduled-task(
  "/MarkLogic/flexrep/tasks/pull.xqy",
  "Modules",
  1,
  xdmp:database("Replica"),
  0,
  xdmp:user("admin"),
  admin:host-get-id($config, xdmp:host-name())
)

(: Add the scheduled task to the Default group. :)

let $config:= admin:group-add-scheduled-task(
  $config,
  admin:group-get-id($config, "Default"),
  $task)

return admin:save-configuration($config)
```

7.3 Configuring Push Replication using the REST API

This section describes how to use the REST API to configure a Master database to push replication updates to the Replica database. The procedures are:

- Preliminary Configuration Procedures
- Installing and Configuring CPF
- Creating a Replication Configuration Element
- Creating a Replication Target
- Creating a Push Replication Scheduled Task

7.3.1 Preliminary Configuration Procedures

This section describes how to create the databases, forests and App Servers to be used for Flexible Replication.

1. Use `POST:/manage/v2/databases` to create a Master, Replica, and Triggers database:

```
curl -v -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d '{"database-name":"Master"}' \
http://localhost:8002/manage/v2/databases
```

```
curl -v -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d '{"database-name":"MyTriggers"}' \
http://localhost:8002/manage/v2/databases

curl -v -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d '{"database-name":"Replica"}' \
http://localhost:8002/manage/v2/databases
```

2. Use `POST:/manage/v2/forests` to create forests for the databases and attach them to the databases.

   Note: The `host` property must specify the name of the host, not `localhost`. In this example, the name of the host is `myhost`.

```
curl --anyauth --user admin:admin -X POST \
-d '{"forest-name": "MasterForest",
     "host": "myhost.marklogic.com",
     "database": "Master"}' \
-i -H "Content-type: application/json" \
http://localhost:8002/manage/v2/forests

curl --anyauth --user admin:admin -X POST \
-d '{"forest-name":"MyTriggersForest",
     "host":"myhost.marklogic.com",
     "database":"MyTriggers"}' \
-i -H "Content-type: application/json" \
http://localhost:8002/manage/v2/forests

curl --anyauth --user admin:admin -X POST \
-d '{"forest-name":"ReplicaForest",
     "host":"myhost.marklogic.com",
     "database":"Replica"}' \
-i -H "Content-type: application/json" \
http://localhost:8002/manage/v2/forests
```

3. Use `PUT:/manage/v2/databases/{id|name}/properties` to configure the Master database to use the MyTriggers database:

```
curl -v -X PUT --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d '{"triggers-database" : "MyTriggers"}' \
http://localhost:8002/manage/v2/databases/Master/properties
```

4. Use `POST:/manage/v2/servers` to create the Master and Replica App Servers, configured with their respective databases.

```
curl -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d '{
   "server-name":"Master-flexrep",
```

```
   "server-type":"http",
   "group-name":"Default",
   "root":"FlexRep",
   "port":8010,
   "content-database":"Master"
   }' \
http://localhost:8002/manage/v2/servers?group-id=Default

curl -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d '{
   "server-name":"Replica-flexrep",
   "server-type":"http",
   "group-name":"Default",
   "root":"FlexRep",
   "port":8011,
   "content-database":"Replica"
   }' \
http://localhost:8002/manage/v2/servers?group-id=Default
```

7.3.2 Installing and Configuring CPF

This section describes how to install CPF and create a domain with the pipelines required for Flexible Replication.

1. Use `POST:/manage/v2/databases/{id|name}/pipelines` to load the Flexible Replication Pipleines

```
curl -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d'{"operation": "load-default-cpf-pipelines"}' \
http://localhost:8002/manage/v2/databases/MyTriggers/pipelines
```

2. Use `POST:/manage/v2/databases/{id|name}/domains` to create a domain

```
curl -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d '{
   "domain-name": "Replicated Content",
   "description": "Flexible Replication Domain",
   "scope": "directory",
   "uri": "/",
   "depth": "infinity",
   "eval-module": "Modules",
   "eval-root": "/",
   "pipeline": ["Flexible Replication","Status Change Handling"]
   }' \
http://localhost:8002/manage/v2/databases/MyTriggers/domains
```

3. Use `POST:/manage/v2/databases/{id|name}/cpf-configs` to install CPF:

```
curl -X POST  --anyauth --user admin:admin --header
"Content-Type:application/json" \
-d '{
   "domain-name": "Replicated Content",
   "restart-user-name": "admin",
   "eval-module": "Modules",
   "eval-root": "/",
   "conversion-enabled": true,
   "permission": [{
       "role-name": "app-user",
       "capability": "read"
       }]
   }' \
http://localhost:8002/manage/v2/databases/MyTriggers/cpf-configs?forma
t=json
```

7.3.3 Creating a Replication Configuration Element

Use `POST:/manage/v2/databases/{id|name}/flexrep/configs` to configure the Replicated Content domain to be used for Flexible Replication:

```
curl -v -X POST  --anyauth --user admin:admin --header
"Content-Type:application/json" \
-d '{"domain-name": "Replicated Content"}' \
http://localhost:8002/manage/v2/databases/Master/flexrep/configs
```

7.3.4 Creating a Replication Target

Use `POST:/manage/v2/databases/{id|name}/flexrep/configs/{id|name}/targets` to create a flexible replication target on Master-flexrep which defines url to Replica-flexrep

```
curl -v -X POST  --anyauth --user admin:admin --header
"Content-Type:application/json" \
-d '{
   "target-name": "replica",
   "url": [ "http://localhost:8011/" ],
   "retry-seconds-min": 60,
   "immediate-push": true,
   "retry-seconds-max": 300,
   "documents-per-batch": 10,
   "enabled": true,
   "replicate-cpf": false,
   "http-options": {
       "username": "admin",
       "password": "admin",
       "client-cert": "",
       "client-key": "",
       "client-pass-phrase": ""},
   "filter-module": "",
   "filter-option": [""]
   }' \
http://localhost:8002/manage/v2/databases/Master/flexrep/configs/Repli
cated%20Content/targets?format=json
```

7.3.5 Creating a Push Replication Scheduled Task

Use `POST:/manage/v2/tasks` to create a flexrep scheduled task:

```
curl -v -X POST  --anyauth --user admin:admin --header
"Content-Type:application/json" \
-d '{
   "task-enabled":true,
   "task-path":"/MarkLogic/flexrep/tasks/push-local-forests.xqy",
   "task-root":"Modules/",
   "task-type":"minutely",
   "task-period":1,
   "task-timestamp":"2014-11-24T13:44:53.143178-08:00",
   "task-database":"Master",
   "task-modules":"",
   "task-user":"admin",
   "task-priority":"higher"}' \
http://localhost:8002/manage/v2/tasks?group-id=Default
```

7.4 Configuring Query-based Replication using the REST API

This section describes how to use the REST API to configure a Master database for Query-based Flexible Replication. In this example, the master database is configured to push replication updates to a database named `push` and a database, named `pull`, to pull replicated updates from the master database. Alerting is used to control what documents are are pushed and pulled, based on their content and which users have read permission on the documents.

Query-based Flexible Replication is described in Configuring Alerting With Flexible Replication in the *Flexible Replication Guide*.

The procedures are:

- Preliminary Configuration Procedures
- Installing and Configuring CPF
- Configuring the Master Database and Creating App Servers
- Creating Users
- Configuring Alerting
- Creating a Replication Configuration Element
- Creating Replication Targets
- Creating Alerting Rules
- Configuring Pull Replication
- Creating a Push and Pull Replication Scheduled Task
- Inserting Some Documents to Test

7.4.1 Preliminary Configuration Procedures

1. Use `POST:/manage/v2/databases` to create the following databases:

   master - the flexrep master database

   master-triggers - the triggers database for 'master'

   master-modules - the modules database for master's triggers

   push - the target database for push replication

   pull - the target database for pull replication

```
curl -v -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
  -d '{"database-name":"master"}'
http://localhost:8002/manage/v2/databases

curl -v -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
  -d '{"database-name":"master-triggers"}'
http://localhost:8002/manage/v2/databases

curl -v -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
  -d '{"database-name":"master-modules"}'
http://localhost:8002/manage/v2/databases

curl -v -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
  -d '{"database-name":"push"}'
http://localhost:8002/manage/v2/databases

curl -v -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
  -d '{"database-name":"pull"}'
http://localhost:8002/manage/v2/databases
```

2. Use `POST:/manage/v2/forests` to create forests for the databases and attach them to the databases.

   Note: The `host` property must specify the name of the host, not `localhost`. In this example, the name of the host is `myhost`.

```
curl --anyauth --user admin:admin -X POST \
-d '{"forest-name": "master",
     "host": "myhost.marklogic.com",
     "database": "master"}' \
-i -H "Content-type: application/json" \
http://localhost:8002/manage/v2/forests
```

```
curl --anyauth --user admin:admin -X POST \
-d '{"forest-name":"master-triggers",
     "host":"myhost.marklogic.com",
     "database":"master-triggers"}' \
-i -H "Content-type: application/json" \
http://localhost:8002/manage/v2/forests

curl --anyauth --user admin:admin -X POST \
-d '{"forest-name":"master-modules",
     "host":"myhost.marklogic.com",
     "database":"master-modules"}' \
-i -H "Content-type: application/json" \
http://localhost:8002/manage/v2/forests

curl --anyauth --user admin:admin -X POST \
-d '{"forest-name":"push",
     "host":"myhost.marklogic.com",
     "database":"push"}' \
-i -H "Content-type: application/json" \
http://localhost:8002/manage/v2/forests

curl --anyauth --user admin:admin -X POST \
-d '{"forest-name":"pull",
     "host":"myhost.marklogic.com",
     "database":"pull"}' \
-i -H "Content-type: application/json" \
http://localhost:8002/manage/v2/forests
```

3. Use `PUT:/manage/v2/databases/{id|name}/properties` to set `master-triggers` as the triggers database

```
curl -v -X PUT --anyauth -u admin:admin \
--header "Content-Type:application/json" \
-d '{"triggers-database" : "master-triggers"}' \
http://localhost:8002/manage/v2/databases/master/properties
```

7.4.2 Installing and Configuring CPF

This section describes how to install CPF and create a domain with the pipelines required for Flexible Replication.

1. Use `POST:/manage/v2/databases/{id|name}/pipelines` to load the Flexible Replication Pipleines into the `master-triggers` database.

```
curl -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d'{"operation": "load-default-cpf-pipelines"}' \
http://localhost:8002/manage/v2/databases/master-triggers/pipelines
```

2. Use `POST:/manage/v2/databases/{id|name}/domains` to create a domain with the `Flexible Replication` and `Status Change Handling` pipelines.

```
curl -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d '{
     "domain-name": "my-cpf-domain",
     "description": "My domain of documents",
     "scope": "directory",
     "uri": "/",
     "depth": "infinity",
     "eval-module": "master-modules",
     "eval-root": "/",
     "pipeline": ["Flexible Replication","Status Change Handling"]
   }' \
http://localhost:8002/manage/v2/databases/master-triggers/domains
```

3. Use `POST:/manage/v2/databases/{id|name}/cpf-configs` to install CPF:

```
curl -X POST  --anyauth --user admin:admin --header
"Content-Type:application/json" \
-d '{
    "domain-name": "my-cpf-domain",
    "restart-user-name": "admin",
    "eval-module": "master-modules",
    "eval-root": "/",
    "conversion-enabled": true,
    "permission": [{
        "role-name": "admin",
        "capability": "execute"
       }]
   }' \
http://localhost:8002/manage/v2/databases/master-triggers/cpf-configs?
format=json
```

7.4.3 Configuring the Master Database and Creating App Servers

1. Use `PUT:/manage/v2/databases/{id|name}/properties` to configure the range indexes needed for flexible replication.

   Note: The range indexes will replace any existing range indexes, so if the application needs additional range indexes you'll need to include them here as well.

```
curl -X PUT  --anyauth --user admin:admin --header
"Content-Type:application/json" \
-d '{"word-positions": true,
     "element-word-positions": true,
     "range-element-index":
    [ { "scalar-type": "dateTime",
        "namespace-uri": "http://marklogic.com/cpf",
        "localname": "last-updated",
        "collation": "",
        "range-value-positions": false,
        "invalid-values": "reject"
      },
```

```
      { "scalar-type": "dateTime",
        "namespace-uri":
"http://marklogic.com/xdmp/flexible-replication",
        "localname": "last-success",
        "collation": "",
        "range-value-positions": true,
        "invalid-values": "reject"
      },
      { "scalar-type": "dateTime",
        "namespace-uri":
"http://marklogic.com/xdmp/flexible-replication",
        "localname": "next-try",
        "collation": "",
        "range-value-positions": true,
        "invalid-values": "reject"
      },
      { "scalar-type": "unsignedLong",
        "namespace-uri":
"http://marklogic.com/xdmp/flexible-replication",
        "localname": "target-id",
        "collation": "",
        "range-value-positions": true,
        "invalid-values": "reject"
      },
      { "scalar-type": "unsignedLong",
        "namespace-uri":
"http://marklogic.com/xdmp/flexible-replication",
        "localname": "domain-id",
        "collation": "",
        "range-value-positions": true,
        "invalid-values": "reject"
      },
      {"scalar-type": "dateTime",
        "namespace-uri":
"http://marklogic.com/xdmp/flexible-replication",
        "localname": "received-at",
        "collation": "",
        "range-value-positions": true,
        "invalid-values": "reject"
      }]}' \
http://localhost:8002/manage/v2/databases/master/properties
```

2. Use `POST:/manage/v2/servers` to create a `master-flexrep` App Server to service the `master` database. The `master-flexrep` App Server is also used to service the alerting rules, as described in “Creating Alerting Rules” on page 98.

```
curl -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d '{
   "server-name":"master-flexrep",
   "server-type":"http",
   "group-name":"Default",
   "root":"FlexRep",
```

```
   "port":8010,
   "content-database":"master",
   "log-errors": true,
   "default-error-format": "compatible",
   "error-handler": "/error-handler.xqy",
   "url-rewriter": "/rewriter.xml",
   "rewrite-resolves-globally": false
   }' \
http://localhost:8002/manage/v2/servers?group-id=Default
```

3. Create a `push-flexrep` App Server for the replica database to receive push replicated content. Each target cluster would need to have one of these configured for inbound replication.

```
curl -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d '{
   "server-name":"push-flexrep",
   "server-type":"http",
   "group-name":"Default",
   "root":"FlexRep",
   "port":8011,
   "content-database":"push",
   "log-errors": true,
   "default-error-format": "compatible",
   "error-handler": "/error-handler.xqy",
   "url-rewriter": "/rewriter.xml",
   "rewrite-resolves-globally": false
   }' \
http://localhost:8002/manage/v2/servers?group-id=Default
```

4. Create a `pull-flexrep` App Server for the replica database to receive pulled replicated content. This would be used for status information on the replica database.

```
curl -X POST  --anyauth -u admin:admin --header
"Content-Type:application/json" \
-d '{
   "server-name":"pull-flexrep",
   "server-type":"http",
   "group-name":"Default",
   "root":"FlexRep",
   "port":8012,
   "content-database":"pull",
   "log-errors": true,
   "default-error-format": "compatible",
   "error-handler": "/error-handler.xqy",
   "url-rewriter": "/rewriter.xml",
   "rewrite-resolves-globally": false
   }' \
http://localhost:8002/manage/v2/servers?group-id=Default
```

7.4.4 Creating Users

Use `POST:/manage/v2/users` to create `user1` for the query-based push target and `user2` for the query-based pull target. Create `flexrep-admin-user` for admin access to the `master-flexrep` App Server.

```
curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{"user-name": "user1",
     "password": "password1",
     "description": "first user",
     "role": [ "flexrep-user", "alert-user" ]
    }' \
http://localhost:8002/manage/v2/users

curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{"user-name": "user2",
     "password": "password2",
     "description": "second user",
     "role": [ "flexrep-user", "alert-user" ]
    }' \
http://localhost:8002/manage/v2/users

curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{"user-name": "flexrep-admin-user",
     "password": "flexrep-admin-password",
     "description": "FlexRep administrative user",
     "role": [ "flexrep-admin", "flexrep-user" ]
    }' \
http://localhost:8002/manage/v2/users
```

7.4.5 Configuring Alerting

Use `POST:/manage/v2/databases/{id|name}/alert/configs` to create the alerting configuration for the master database.

```
curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{
  "uri": "http://acme.com/alerting",
  "name": "qbfr",
  "description": "alerting rules for query-based flexrep",
  "trigger": [],
  "domain": [],
  "action": [],
  "option": []
   }' \
http://localhost:8002/manage/v2/databases/master/alert/configs
```

Use `POST:/manage/v2/databases/{id|name}/alert/actions` to create the alert action and apply it to the alert configuration.

```
curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{
  "name": "log",
  "description": "QBFR log action",
  "module": "/log.xqy",
  "module-db": "master-modules",
  "module-root": "/",
  "option": []
   }' \
http://localhost:8002/manage/v2/databases/master/alert/actions?uri=htt
p://acme.com/alerting
```

7.4.6 Creating a Replication Configuration Element

Use `POST:/manage/v2/databases/{id|name}/flexrep/configs` to configure the `my-cpf-domain` domain to be used for Flexible Replication:

```
curl -v -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{"domain-name": "my-cpf-domain",
     "alerting-uri": "http://acme.com/alerting"}' \
http://localhost:8002/manage/v2/databases/master/flexrep/configs
```

7.4.7 Creating Replication Targets

Use `POST:/manage/v2/databases/{id|name}/flexrep/configs/{id|name}/targets` to create flexible replication targets for the `push` and `pull` databases on `master` database.

> **Note:** The `user1` user is associated with `user1-target` to ensure only documents intended for `user1` are replicated to that target. The same is done for `user2` and `user1-target`. The alerting rules described in “Creating Alerting Rules” on page 98 control which documents are replicated to which target.

1. Create the push target, `user1-target`. The `user` setting is required for a query-based target.

```
curl -X POST  --anyauth --user admin:admin \
-H "Content-Type: application/json" \
-d '{"target-name": "user1-target",
     "url": [ "http://localhost:8011/" ],
     "documents-per-batch": 10,
     "http-options": {
        "username": "admin",
        "password": "admin"},
     "user": "user1"}' \
http://localhost:8002/manage/v2/databases/master/flexrep/configs/my-cp
f-domain/targets
```

2. Create the pull target, `user2-target`. The `user` setting is required for a query-based target

```
curl -X POST  --anyauth --user admin:admin \
-H "Content-Type: application/json" \
-d '{"target-name": "user2-target",
     "url": [],
     "documents-per-batch": 10,
     "user": "user2"}' \
http://localhost:8002/manage/v2/databases/master/flexrep/configs/my-cp
f-domain/targets
```

7.4.8 Creating Alerting Rules

Use `POST:/v1/domains/{domain-id-or-default-domain-name}/targets/{id|name}/rules` to create the alerting rules for the two users, via the `master-flexrep` App Server (port 8010) on the master database.

1. The first two rules match documents readable by `user1` and contain either the word `apple` or `orange`. In this way, only documents with these attributes will be pushed to the `push` replica database.

```
curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{ "name": "apple",
    "query": [
      {"wordQuery":{"text":["apple"],
      "user-name": "user1",
       "options":["lang=en"]}} ],
       "action-name": "log" }' \
http://localhost:8002/manage/v2/databases/master/alert/actions/log/rul
es?uri=http://acme.com/alerting
curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{ "name": "orange",
    "query": [
      {"wordQuery":{"text":["orange"],
      "user-name": "user2",
       "options":["lang=en"]}} ],
       "action-name": "log" }' \
http://localhost:8002/manage/v2/databases/master/alert/actions/log/rul
es?uri=http://acme.com/alerting
```

2. The second two rules match documents readable by `user2` and contain either the word `apple` or `banana`. In this way, only documents with these attributes will be pulled into the `pull` replica database.

```
curl -X POST  --anyauth --user admin:admin \

--header "Content-Type:application/json" \

-d '{ "name": "apple",

      "query": [
```

```
          {"wordQuery":{"text":["apple"],

          "user-name": "user1",

           "options":["lang=en"]}} ],

           "action-name": "log" }' \

http://localhost:8002/manage/v2/databases/master/alert/actions/log/rul
es?uri=http://acme.com/alerting

curl -X POST  --anyauth --user admin:admin \

--header "Content-Type:application/json" \

-d '{ "name": "orange",

      "query": [

         {"wordQuery":{"text":["orange"],

         "user-name": "user2",

          "options":["lang=en"]}} ],

      "action-name": "log" }' \

http://localhost:8002/manage/v2/databases/master/alert/actions/log/rul
es?uri=http://acme.com/alerting
```

7.4.9 Configuring Pull Replication

To configure pull replication, you need the domain id and target id of the user2-target.

1. You can return the domain id and target id by doing a GET on the following endpoints (note that these are executed on port `8010`, which is the `master-flexrep` App Server):

   ```
   http://localhost:8010/v1/domains/my-cpf-domain/targets?format=json

   http://localhost:8010/v1/domains/my-cpf-domain/targets/user2-target?fo
   rmat=json
   ```

2. In this example, the domain id returned is `7860365094706821763` and the target id of user2-target is `15425733952606449897`. Use `POST:/manage/v2/databases/{id|name}/flexrep/pulls` to configure pull replication on the `pull` database:

   ```
   curl -X POST  --anyauth --user admin:admin \
   -H "Content-type: application/json" \
   -d '{"pull-name": "from-master",
        "domain-id": "7860365094706821763",
        "target-id": "15425733952606449897",
   ```

```
     "url": [ "http://localhost:8010/" ],
     "http-options": {
          "username": "user2",
          "password": "password2"}
     }' \
http://localhost:8002/manage/v2/databases/pull/flexrep/pulls
```

7.4.10 Creating a Push and Pull Replication Scheduled Task

The master database needs a scheduled task to handle retries and zero-day replication. The pull target's database also needs a scheduled task to pull content from the master database.

1. Use `POST:/manage/v2/tasks` to create a flexrep scheduled task to push replicated content to the push database:

```
curl -X POST --anyauth --user admin:admin \
-H "Content-type: application/json" \
-d '{"task-priority": "higher",
     "task-user": "admin",
     "task-database": "master",
     "task-path": "/MarkLogic/flexrep/tasks/push-local-forests.xqy",
     "task-root": "Modules/",
     "task-type": "minutely",
     "task-period": "1"}' \
http://localhost:8002/manage/v2/tasks?group-id=Default
```

1. Use `POST:/manage/v2/tasks` to create a flexrep scheduled task to pull replicated content from the `master` database into the `pull` database:

```
curl -X POST --anyauth --user admin:admin \
-H "Content-type: application/json" \
-d '{"task-priority": "higher",
     "task-user": "admin",
     "task-database": "pull",
     "task-path": "/MarkLogic/flexrep/tasks/pull.xqy",
     "task-root": "Modules/",
     "task-type": "minutely",
     "task-period": "1"}' \
http://localhost:8002/manage/v2/tasks?group-id=Default
```

7.4.11 Inserting Some Documents to Test

Use `PUT:/v1/documents` to insert some JSON and XML documents into the `master` database. (Note that these are executed on port `8000` and that single quotes are used in the URI because of the multiple parameters.)

> **Note:** These calls require that you have the `eval-in` privilege, as described in Security Requirements in the *REST Application Developer's Guide*. Also, you must insert each document with the `alert-user:read` permission.

1. Insert some documents that include the word 'apple.' For example:

```
curl -X PUT --anyauth  --user admin:admin -H "Content-type: application/json" \
-d '{"produce":{"fruit": "apple"}}' \
'http://localhost:8000/LATEST/documents?uri=/json/apple.json&database=master&p
erm:alert-user=read'

curl --anyauth  --user admin:admin -X PUT -H "Content-type: application/xml" \
-d '<produce>
        <fruit>apple</fruit>
    </produce>' \
'http://localhost:8000/LATEST/documents?uri=/xml/apple.xml&database=master&per
m:alert-user=read'
```

2. Insert some documents that include the word ‘orange.’ For example:

```
curl -X PUT --anyauth  --user admin:admin -H "Content-type: application/json" \
-d '{"produce":{"fruit": "orange"}}' \
'http://localhost:8000/LATEST/documents?uri=/json/orange.json&database=master&
perm:alert-user=read'

curl --anyauth  --user admin:admin -X PUT -H "Content-type: application/xml" \
-d '<produce>
        <fruit>orange</fruit>
    </produce>' \
'http://localhost:8000/LATEST/documents?uri=/xml/orange.xml&database=master&pe
rm:alert-user=read'
```

3. Insert some documents that include the word ‘banana.’ For example:

```
curl -X PUT --anyauth  --user admin:admin -H "Content-type: application/json" \
-d '{"produce":{"fruit": "banana"}}' \
'http://localhost:8000/LATEST/documents?uri=/json/banana.json&database=master&
perm:alert-user=read'

curl --anyauth  --user admin:admin -X PUT -H "Content-type: application/xml" \
-d '<produce>
        <fruit>banana</fruit>
    </produce>' \
'http://localhost:8000/LATEST/documents?uri=/xml/banana.xml&database=master&pe
rm:alert-user=read'
```

Based on the rules created in “Creating Alerting Rules” on page 98, the documents containing `apple` and `orange` should be replicated to the `push` database and the documents containing `apple` and `banana` should be replicated to the `pull` database.

8.0 Scripting Database Replication Configuration

This chapter describes how to use the REST API to configure database replication. For details on database replication, see the *Database Replication Guide*. The procedures are:

8.1 Coupling the Master and Replica Clusters

Use `GET:/manage/v2/properties` to return the properties of each cluster to participate in the database replication configuration and pass the properties to `POST:/manage/v2/clusters` to couple the clusters.

The example Python script below gets the properties (in JSON format) from the `master` and `replica` clusters and passes the `master` properties to the `replica` cluster and the `replica` properties to the master `cluster`.

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import json
import requests
from requests.auth import HTTPDigestAuth
from pprint import pprint

def main():

        # Get the properties for each cluster to be coupled

        auth = HTTPDigestAuth('admin', 'admin')
        headers = {'accept': 'application/json'}
        masterURI = 'http://master.marklogic.com:8002/manage/v2/properties'
        replicaURI = 'http://replica.marklogic.com:8002/manage/v2/properties'

        g1 = requests.get(masterURI, auth=auth, headers=headers)
        master_info = g1.json()

        g2 = requests.get(replicaURI, auth=auth, headers=headers)
        replica_info = g2.json()

        # Couple the clusters

        masterURI = 'http://master.marklogic.com:8002/manage/v2/clusters'
        replicaURI = 'http://replica.marklogic.com:8002/manage/v2/clusters'
        headers = {'content-type': 'application/json'}

        p1 = requests.post(masterURI, auth=auth,
                           data=json.dumps(replica_info),
                           headers=headers)
```

```
        p2 = requests.post(replicaURI, auth=auth,
                           data=json.dumps(master_info),
                           headers=headers)

        # Return responses from the REST calls

        print p1.text
        print p2.text

main()
```

`POST:/manage/v2/clusters` accepts all of the properties returned from `GET:/manage/v2/properties` as its payload. Should you have a need for finer control over the cluster properties when coupling clusters, you can "manually" build the payload for `POST:/manage/v2/clusters`.

The Python script below extracts the properties from the `master` cluster and builds the payload needed by `POST:/manage/v2/clusters` to couple the `replica` cluster to the `master` cluster. You would need to write similar code to couple the `master` cluster to the `replica` cluster.

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import json
import requests
from requests.auth import HTTPDigestAuth
from pprint import pprint

def main():

        # Get the properties for master cluster

        auth = HTTPDigestAuth('admin', 'admin')
        headers = {'accept': 'application/json'}
        masterURI = 'http://master.marklogic.com:8002/manage/v2/properties'

        g1 = requests.get(masterURI, auth=auth, headers=headers)
        master_info = g1.json()

        # Extract the properties needed by POST:/manage/v2/clusters

        fhostid = master_info['bootstrap-host'][0]['bootstrap-host-id']
        fhostn = master_info['bootstrap-host'][0]['bootstrap-host-name']
        fcid = master_info['cluster-id']
        fcn = master_info['cluster-name']
        fccert = master_info['xdqp-ssl-certificate']

        # Build the payload for POST:/manage/v2/clusters

        payload = {
          "foreign-cluster-id": fcid,
          "foreign-cluster-name": fcn,
```

```
          "foreign-protocol": "http",
          "foreign-ssl-certificate": fccert,
          "xdqp-ssl-enabled": True,
          "xdqp-ssl-allow-sslv3": True,
          "xdqp-ssl-allow-tls": True,
          "xdqp-ssl-ciphers": "ALL:!LOW:@STRENGTH",
          "xdqp-timeout": 10,
          "host-timeout": 30,
          "foreign-bootstrap-host": [{
          "foreign-host-id": fhostid,
          "foreign-host-name": fhostn,
          "foreign-connect-port": 7998
          }]}

        # Couple the replica cluster to the master cluster.

          url = 'http://replica.marklogic.com:8002/manage/v2/clusters'
          headers = {'content-type': 'application/json'}

          r = requests.post(url, auth=auth, data=json.dumps(payload),
                            headers=headers)

        # Return responses from the REST call

          print r.text

main()
```

8.2 Setting the Master and Replica Databases

Once the clusters are coupled you can use `POST:/manage/v2/databases/{id|name}` to configure the master and replica databases.

The Python script below uses `POST:/manage/v2/databases/{id|name}` to configure the `Documents` database on the `master` cluster as the master database and the `Documents` database on the `replica` cluster as the replica database.

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import json
import requests
from requests.auth import HTTPDigestAuth
from pprint import pprint

def main():

        # Define the operations to set the master and replica databases.

         set_master = {
           "operation": "set-foreign-master",
           "foreign-master": {
              "foreign-cluster-name": "master.marklogic.com-cluster",
```

```
            "foreign-database-name": "Documents",
            "connect-forests-by-name": True}}

        set_replica = {
          "operation": "add-foreign-replicas",
          "foreign-replica": [{
             "foreign-cluster-name": "replica.marklogic.com-cluster",
             "foreign-database-name": "Documents",
             "connect-forests-by-name": True,
             "lag-limit": 23,
             "enabled": True}]}

        masterURI =
          'http://master.marklogic.com:8002/manage/v2/databases/Documents'

        replicaURI =
          'http://replica.marklogic.com:8002/manage/v2/databases/Documents'

        headers = {'content-type': 'application/json',
                   'accept': 'application/json'}

       # Set the master and replica databases.

        r1 = requests.post(replicaURI, auth=HTTPDigestAuth('admin', 'admin'),
                           data=json.dumps(set_master), headers=headers)

        r2 = requests.post(masterURI, auth=HTTPDigestAuth('admin', 'admin'),
                           data=json.dumps(set_replica), headers=headers)

       # Return responses from the REST calls

        print r1.text
        print r2.text

main()
```

9.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For technical questions, we encourage you to ask your question on Stack Overflow.

10.0 Copyright

MarkLogic Server 10.0 and supporting products.
Last updated: February, 2022

This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2, US 8,892,599, and US 8,935,267.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices for your version of MarkLogic.