
MarkLogic Server

Scripting Administrative Tasks Guide

Release 4.2
October, 2010

Last Revised: 4.2-1, October, 2010

Copyright

© Copyright 2002-2012 by MarkLogic Corporation. All rights reserved worldwide.

This Material is confidential and is protected under your license agreement.

Excel and PowerPoint are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. This document is an independent publication of MarkLogic Corporation and is not affiliated with, nor has it been authorized, sponsored or otherwise approved by Microsoft Corporation.

Contains LinguistX, from Inxight Software, Inc. Copyright © 1996-2006. All rights reserved. www.inxight.com.

Antenna House OfficeHTML Copyright © 2000-2008 Antenna House, Inc. All rights reserved.

Argus Copyright ©1999-2008 Icenit Technology Ltd. All rights reserved.

Contains Rosette Linguistics Platform 6.0 from Basis Technology Corporation, Copyright © 2004-2008 Basis Technology Corporation. All rights reserved.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (<http://www.openssl.org/>) Copyright © 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved. Copyright © 1998-2001 The OpenSSL Project. All rights reserved.

Contains software derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm. Copyright © 1991-1992, RSA Data Security, Inc. Created 1991. All rights reserved.

Contains ICU with the following copyright and permission notice:

Copyright © 1995-2010 International Business Machines Corporation and others. All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

Copyright	2
1.0 Scripting Administrative Tasks in MarkLogic Server	7
1.1 The Administration APIs	7
1.2 Common Use Cases	8
1.3 Chapter Summary	8
2.0 Using the Admin API	9
2.1 Tasks You Can Perform Using the Admin API	9
2.2 Transactional Considerations When Using the Admin API	10
2.3 Privileges Required For Running Admin APIs	10
2.4 General Steps for Scripting Administrative Tasks	10
2.4.1 Importing the Admin Library Module	11
2.4.2 Getting the Current Configuration in Memory	11
2.4.3 Saving the Configuration Changes	11
2.5 Techniques for Making Multiple Changes to a Configuration	12
2.5.1 Saving versus Passing the Configuration	12
2.5.2 Creating and Configuring Objects in a Single Transaction	13
2.5.3 Making Transactions Idempotent	15
2.5.4 When Separate Transactions are Needed	17
2.6 Sequence for Creating Server and Security Objects	18
2.7 Sequence for Deleting Server and Security Objects	19
2.8 Executing Queries in Select Databases	20
3.0 Server Configuration Scripts	21
3.1 Creating and Configuring Forests and Databases	21
3.1.1 Creating Forests and Databases	22
3.1.2 Attaching Forests to Databases	23
3.1.3 Adding a Database Field and Included Element	23
3.1.4 Adding Indexes to a Database	24
3.1.5 Creating a Scheduled Backup of a Database	25
3.1.6 Creating and Configuring Databases in a Single Transaction	26
3.1.7 Deleting a Forest and Database	29
3.2 Creating and Configuring Groups	30
3.2.1 Creating a Group	30
3.2.2 Enabling Auditing on a Group	31
3.2.3 Creating a New Namespace for a Group	31
3.2.4 Creating and Configuring a Group in a Single Transaction	32
3.2.5 Deleting a Group	33
3.3 Creating and Configuring App Servers	33
3.3.1 Creating an App Server	34
3.3.2 Setting a URL Rewriter on an App Server	34
3.3.3 Setting the Concurrent Request Limit on an App Server	35
3.3.4 Enabling Display Last-Login on an App Server	35

3.3.5	Creating and Configuring an App Server in a Single Transaction	36
3.3.6	Deleting an App Server	37
3.4	Creating and Configuring Roles and Users	37
4.0	Sample Configuration Program	39
4.1	How to Execute the Sample Configuration Program	40
4.2	Structure of the Sample Configuration Program	42
4.3	install.xqy	43
4.4	uninstall.xqy	45
4.5	configure-server.xqy	47
4.5.1	create-forests function	48
4.5.2	create-databases function	48
4.5.3	create-server function	51
4.5.4	remove-config function	52
4.6	groups.xqy	53
4.6.1	create-group function	54
4.6.2	create-server function	54
4.6.3	app server configuration functions	58
4.6.4	get-security-db function	60
4.6.5	delete-group function	60
4.6.6	delete-server function	61
4.6.7	set-host function	62
4.7	databases.xqy	63
4.7.1	create-forest function	64
4.7.2	create-database function	64
4.7.3	database configuration functions	65
4.7.4	delete-database function	70
4.7.5	delete-forest function	71
4.8	configure-security.xqy	72
4.8.1	configure-roles function	73
4.8.2	configure-users function	74
4.8.3	remove-users function	74
4.8.4	remove-roles function	75
5.0	Server Maintenance Operations	76
5.1	Group Maintenance Operations	76
5.1.1	Enabling Auditing on a Group	77
5.1.2	Disabling Auditing on a Group	77
5.1.3	Removing Events to be Audited on a Group	78
5.1.4	Adding a Namespace to a Group	78
5.1.5	Returning the Namespace Settings on a Group	78
5.1.6	Deleting a Namespace from a Group	79
5.1.7	Returning the System Log Settings	79
5.1.8	Resetting the System Log Settings	80
5.1.9	Creating a New Hourly Task	81

5.1.10	Deleting all Scheduled Tasks from a Group	81
5.2	App Server Maintenance Operations	82
5.2.1	Modifying the App Server Root for an HTTP App Server	82
5.2.2	Changing the App Server Root and Cloning the Changed App Server ...	83
5.2.3	Enabling SSL on an App Server	83
5.2.3.1	Creating a Certificate Template	84
5.2.3.2	Generating a Certificate Request	85
5.2.3.3	Importing a Signed Certificate into the Database	85
5.2.3.4	Setting a Certificate Template on an App Server	86
5.3	Database Maintenance Operations	87
5.3.1	Creating a Database by Cloning an Existing Database Configuration	87
5.3.2	Returning the Size of the Forests in a Database	88
5.3.3	Disabling Database Merges	88
5.3.4	Deleting Element and Attribute Range Indexes	89
5.3.5	Adding a Fragment Root to a Database	90
5.3.6	Returning the Fragment Roots Set in a Database	90
5.3.7	Deleting a Fragment Root from a Database	90
5.3.8	Merging the Forests in a Database	91
5.3.9	Backing Up a Database	91
5.3.10	Restoring from a Previous Backup	91
5.3.11	Scheduling Forest Backups	92
5.3.12	Alerting the Administrator if the Forest Grows Beyond its Maximum Allowable Size	92
5.3.13	Rotating Forest Update Types	94
5.4	Host Maintenance Operations	98
5.4.1	Returning the Status of the Host	98
5.4.2	Returning the Time Host was Last Started	98
5.4.3	Restarting MarkLogic Server on all Hosts in the Cluster	99
5.5	User Maintenance Operations	99
5.5.1	Removing all Users with a Specific Role	99
5.5.2	Removing a Specific Role, if Present	100
5.5.3	Retrieving the Last-Login Information	100
6.0	Scripting Content Processing Framework (CPF) Configuration	101
6.1	General Procedure for Configuring CPF	101
6.2	Creating CPF Pipelines	102
6.3	Inserting Existing CPF Pipelines	104
6.4	Creating a CPF Domain	105
6.5	Configuring a CPF Restart Trigger	106
7.0	Scripting Flexible Replication Configuration	107
7.1	Preliminary Configuration Procedures	107
7.2	Configuring the Master Database	109
7.3	Creating a Replication Configuration Element	110
7.4	Creating a Replication Target	111

7.5	Creating a Push Replication Scheduled Task	112
7.6	Configuring Pull Replication	113
7.6.1	Disabling Push Replication on the Master Database	113
7.6.2	Creating a Pull Replication Configuration	114
7.6.3	Creating a Pull Replication Scheduled Task	115
8.0	Technical Support	116

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:

- [The Administration APIs](#)
- [Common Use Cases](#)
- [Chapter Summary](#)

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 9.
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 21, “Sample Configuration Program” on page 39, and “Server Maintenance Operations” on page 76 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 [Sample Configuration Program](#) chapter describes the modules that make up the Sample Configuration Program, which creates an entire configuration that supports the Oscars Explorer application described in [Using the Oscars Sample Application](#) in the *Application Builder Developer’s Guide*.

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 7 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 10 and “Techniques for Making Multiple Changes to a Configuration” on page 12.

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

- [Tasks You Can Perform Using the Admin API](#)
- [Transactional Considerations When Using the Admin API](#)
- [Privileges Required For Running Admin APIs](#)
- [General Steps for Scripting Administrative Tasks](#)
- [Techniques for Making Multiple Changes to a Configuration](#)
- [Sequence for Creating Server and Security Objects](#)
- [Sequence for Deleting Server and Security Objects](#)
- [Executing Queries in Select Databases](#)

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 12.

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 *Understanding and Using 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 26 :

```
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 13 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 `xmpp:eval` or `xmpp: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
<code>admin:database-attach-forest</code>	Forest must exist.
<code>admin:appserver-set-database</code>	Content database must exist.
<code>admin:appserver-set-modules-database</code>	Modules database must exist.
<code>admin:host-set-group</code>	Group must exist.
<code>admin:http-server-create</code>	Group and databases must exist.
<code>admin:xdbc-server-create</code>	Group and databases must exist.
<code>admin:webdav-server-create</code>	Group and databases must exist.
<code>admin:group-delete</code>	All hosts must be reassigned to another group.
<code>admin:database-delete</code>	All app servers that use the database must be deleted or reassigned another database.
<code>admin:forest-delete</code>	All databases that attach the forest must be deleted or attached to another forest.

See “Sequence for Creating Server and Security Objects” on page 18 and “Sequence for Deleting Server and Security Objects” on page 19 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. For example, if you configure your App Server as shown in “How to Execute the Sample Configuration Program” on page 40, your queries are executed in the Documents database.

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 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, “Sample Configuration Program” on page 39. 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](#)

3.1 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*.

The sample configuration modules that create and configure forests and databases are: [configure-server.xqy](#) and [databases.xqy](#). The [configure-server.xqy](#) module specifies the configuration-specific functions, which call the generic functions in the [databases.xqy](#) module. This section shows individual XQuery scripts that perform similar tasks. For more examples on how to use the Admin functions to modify your databases and monitor their operation, see “Database Maintenance Operations” on page 87.

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.

3.1.1 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)
```

3.1.2 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)
```

3.1.3 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) )
```

3.1.4 Adding Indexes to a Database

As described in [Understanding Element and Attribute 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-m1";
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))
```

3.1.5 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-m1";
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))
```

3.1.6 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 13, 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 15.

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)
```

3.1.7 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 26:

```
xquery version "1.0-m1";

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)
```

3.2 Creating and Configuring Groups

The general information on groups is provided in the [Groups](#) chapter in the *Administrator's Guide*.

The sample configuration module that creates and configures groups is [groups.xqy](#). Though the code exists from creating a new group, it is not called by the [configure-server.xqy](#) module because the `sample-server` is created in the existing Default group. This section shows individual XQuery scripts that create and configure groups. See “Group Maintenance Operations” on page 76 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](#)

3.2.1 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"))
```

3.2.2 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()))
```

3.2.3 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")))
```

3.2.4 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 13, 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 15.

```
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)
```

3.2.5 Deleting a Group

The following script deletes the group created in “Creating and Configuring a Group in a Single Transaction” on page 32:

```
xquery version "1.0-m1";
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")))
```

3.3 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*.

The sample configuration modules that create and configure App Servers are: [configure-server.xqy](#) and [groups.xqy](#). The [configure-server.xqy](#) module specifies the configuration-specific functions, which call the generic functions in the [groups.xqy](#) module. This section shows individual XQuery scripts that perform similar tasks. See “App Server Maintenance Operations” on page 82 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](#)
- [Setting a URL Rewriter on an App Server](#)
- [Setting the Concurrent Request Limit on an App Server](#)
- [Enabling Display Last-Login on an App Server](#)
- [Creating and Configuring an App Server in a Single Transaction](#)
- [Deleting an App Server](#)

3.3.1 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);
```

3.3.2 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)
```

3.3.3 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-m1";  
  
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))
```

3.3.4 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-m1";  
  
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()))
```

3.3.5 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 13, 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 15.

```
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)

```

3.3.6 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 36:

```

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)

```

3.4 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*.

The sample configuration module that creates and configures roles and users is: [configure-security.xqy](#). 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 99 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 20 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-m1";
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-m1";
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",
  (),
  ())
```

4.0 Sample Configuration Program

The sample configuration program uses the Admin APIs to construct a MarkLogic Server configuration that supports the Oscars Explorer application described in [Using the Oscars Sample Application](#) in the *Application Builder Developer's Guide*.

The sample configuration program is located in `MarkLogic/Samples/admin`. The objective of this sample program is to educate users on the use of the Admin APIs. Though many parts of the code can be reused, the modules in this example are structured to highlight the use of the Admin API functions. You, however, may want to structure your program to hide the complexities of the APIs and define all of your configuration variables in a separate module or XML file, so that all changes to the configuration can be made in one place.

The topics in this section are:

- [How to Execute the Sample Configuration Program](#)
- [Structure of the Sample Configuration Program](#)
- [install.xqy](#)
- [uninstall.xqy](#)
- [configure-server.xqy](#)
- [groups.xqy](#)
- [databases.xqy](#)
- [configure-security.xqy](#)

4.1 How to Execute the Sample Configuration Program

Before you can run the Sample Configuration Program, you must create an HTTP App Server that has a root path to the location of the program files.

To create an App Server for the Sample Configuration Program, do the following:

1. Open the Admin Interface and create a new HTTP App Server, as described in [Creating a New HTTP Server](#) in the *Administrator's Guide*. Set the App Server root to `Samples/admin` and assign it a unique port number, such as port `8055` in this example:

http server -- A HTTP server specification.

server name
The server name.

root
The root document directory pathname.

port*
The server socket bind internet port number.

modules
The database that contains application modules.

database
The database name.

2. In a browser window, open the `install.xqy` file at the App Server URL. For example:

```
http://localhost:8055/install.xqy
```

You should see output like:

```
Server is Configured

Done loading XML data into Sample-Database

Done loading application data into Sample-Modules

Role IDs are:
1067450794701390527
10780693773632192097
8870937581372109438
User IDs are:
3884989588766400714
Users and Roles are Configured
```

3. To start the sample application, enter: `http://localhost:8016`. You should see the Oscars Explorer screen:

The screenshot shows the Oscar Explorer web application. The header includes the title "Oscar® Explorer" and a user greeting "Welcome, gfurush". A search bar is located at the top right. On the left, a "BROWSE" sidebar lists categories: Award, Lead Actor (9), Decade (1940s (5), 1930s (4)), and Winners (Nominated (6), Won (3)). The main content area is divided into "TOP PEOPLE" and "TOP FILMS".

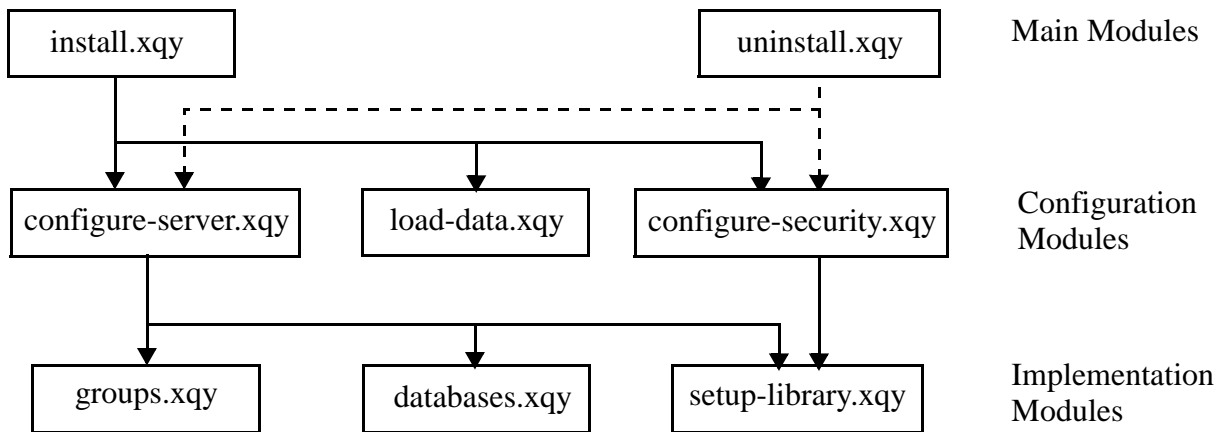
TOP PEOPLE	TOP FILMS
James Stewart (2)	Angels with Dirty Faces (1)
Charles Laughton (1)	Body and Soul (1)
Fredric March (1)	Citizen Kane (1)
Gregory Peck (1)	Gentleman's Agreement (1)
James Cagney (1)	It's a Wonderful Life (1)
John Garfield (1)	Mr. Smith Goes to Washington (1)
Orson Welles (1)	The Best Years of Our Lives (1)
Victor McLaglen (1)	The Informer (1)
	The Private Life of Henry VIII (1)

At the bottom of the main content area, there is a message: "Load the [full sample dataset](#) (approximately 2 - 5 minutes; 20 MB download, ~2100 documents inserted)".

The footer contains navigation links: "Oscar® Explorer Help | Contact Mark Logic | Terms of Use" and a copyright notice: "© 2009, Mark Logic Corporation, All Rights Reserved."

4.2 Structure of the Sample Configuration Program

The sample configuration program consists of the modules illustrated below:



- [install.xqy](#) is the main module that calls the configuration-specific functions in the [configure-server.xqy](#), [configure-security.xqy](#), and `load-data.xqy` modules to create the sample configuration and load the data into the databases.
- [uninstall.xqy](#) is the main module that calls the functions in the [configure-server.xqy](#) and [configure-security.xqy](#) modules to remove the sample configuration.
- [configure-server.xqy](#) defines the configuration-specific functions that call the implementation functions in the [groups.xqy](#) and [databases.xqy](#) modules to create and configure the app server and databases needed to support the Oscars Explorer application.
- [configure-security.xqy](#) contains the implementation functions that create and remove the roles and users. The functions in this module call the functions in the MarkLogic Server APIs.
- [groups.xqy](#) contains the implementation functions for creating, configuring, and removing groups and servers. The functions in this module call the functions in the MarkLogic Server APIs and are general enough to be used to create configurations to support applications other than Oscars Explorer.
- [databases.xqy](#) contains the implementation functions for creating, configuring, and removing forests and databases. The functions in this module call the functions in the MarkLogic Server APIs and are general enough to be used to create configurations to support applications other than Oscars Explorer.
- **load-data.xqy** contains the functions that load the content database and application module database. This module exists only to load the data in order to get the Oscars application working and its implementation is outside the scope of this document.
- **setup-library.xqy** contains the generic code for logging status and errors.

4.3 install.xqy

The `install.xqy` module is the main module that calls the functions in the [configure-server.xqy](#), [configure-security.xqy](#), and `loadData.xqy` modules to configure the MarkLogic Server and load the data needed to support the Oscars Explorer application. The functions in the [configure-security.xqy](#) and `loadData.xqy` modules are executed against their respective databases by calling them within `xdmp:eval` functions with the `<database>` option. For more detail on executing queries in a database other than that assigned to the App Server, see “Executing Queries in Select Databases” on page 20.

Note: The `loadData.xqy` module is needed to load the application and content data into the databases, but its implementation is not within the scope of this guide.

```
xquery version "1.0-ml";

import module namespace conf-server =
  "http://marklogic.com/sampleConfig/conf-server"
  at "configure-server.xqy";

import module namespace grp="http://marklogic.com/sampleConfig/groups"
  at "groups.xqy";

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

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

(: Get the Security database used by the Admin AppServer :)
let $securityDB := grp:get-security-db($config, "Default", "Admin")

(: Create and configure the databases and server :)
let $config := conf-server:create-forests($config)
let $config := conf-server:create-databases($config)
let $config := conf-server:create-server($config)

(: These variables define the functions that create new roles and
   users and load the data into the databases. Each function is
   then executed by xdmp:eval against a specific database :)

let $create-roles:= '
  xquery version "1.0-ml";
  import module namespace conf-security=
    "http://marklogic.com/sampleConfig/conf-security" at
    "configure-security.xqy";
  conf-security:configure-roles()'
```

```

let $create-users := '
  xquery version "1.0-ml";
  import module namespace conf-security=
    "http://marklogic.com/sampleConfig/conf-security"
    at "configure-security.xqy";
  conf-security:configure-users () '

let $load-XML := '
  xquery version "1.0-ml";
  import module namespace load-data =
    "http://marklogic.com/sampleConfig/load-data" at "loadData.xqy";
  load-data:load-sample-XML(
    "C:/Program Files/MarkLogic/Assets/appbuilder/templates/oscars/data") '

let $load-modules := '
  xquery version "1.0-ml";
  import module namespace setup =
    "http://marklogic.com/sampleConfig/setup-lib"
    at "setup-library.xqy";
  import module namespace load-data =
    "http://marklogic.com/sampleConfig/load-data" at "loadData.xqy";
  load-data:load-sample-module($setup:LOADDATA) '

return (

  admin:save-configuration($config),
  "Server is Configured",

  (: Call the above functions in their respective databases :)

  xdmp:eval(
    $load-XML,
    (),
    <options xmlns="xdmp:eval">
      <database>{xdmp:database("Sample-Database")}</database>
    </options>),

  xdmp:eval(
    $load-modules,
    (),
    <options xmlns="xdmp:eval">
      <database>{xdmp:database("Sample-Modules")}</database>
    </options>),

  "Role IDs are:",
  xdmp:eval(
    $create-roles,
    (),
    <options xmlns="xdmp:eval">
      <database>{$securityDB}</database>
    </options>),

```

```

    "User IDs are:",
    xdm:eval(
      $create-users,
      (),
      <options xmlns="xdmp:eval">
        <database>{$securityDB}</database>
      </options>),

    "Users and Roles are Configured"

  )

```

4.4 uninstall.xqy

The `uninstall.xqy` module is the main module that calls the functions in the [configure-server.xqy](#) and [configure-security.xqy](#) modules to remove the MarkLogic Server objects created by the [install.xqy](#) module. The functions in the [configure-security.xqy](#) module are executed against their respective databases by calling them within `xdmp:eval` functions with the `<database>` option.

```

xquery version "1.0-ml";

import module namespace conf-server =
  "http://marklogic.com/sampleConfig/conf-server"
  at "configure-server.xqy";

import module namespace grp="http://marklogic.com/sampleConfig/groups"
  at "groups.xqy";

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

let $remove-users := '
  xquery version "1.0-ml";

import module namespace conf-security =
  "http://marklogic.com/sampleConfig/conf-security"
  at "configure-security.xqy";

conf-security:remove-users() '

let $remove-roles := '
  xquery version "1.0-ml";

import module namespace conf-security =
  "http://marklogic.com/sampleConfig/conf-security"
  at "configure-security.xqy";

conf-security:remove-roles() '

let $config := admin:get-configuration()
let $securityDB := grp:get-security-db($config, "Default", "Admin")

```

```
return (  
  
  xdm:eval($remove-roles, (),  
    <options xmlns="xdmp:eval">  
      <database>{$securityDB}</database>  
    </options>),  
  
  xdm:eval($remove-users, (),  
    <options xmlns="xdmp:eval">  
      <database>{$securityDB}</database>  
    </options>),  
  
  "Removed Users and Roles",  
  
  conf-server:remove-config(),  
  
  "Removed Server Configuration"  
  
)
```

4.5 configure-server.xqy

The `configure-server.xqy` module defines the high-level, configuration-specific functions that call the more “generic” functions in the [groups.xqy](#) and [databases.xqy](#) modules to create, configure, and delete the forests, databases and app server used by the Oscars Explorer application.

The functions in this module are:

- [create-forests function](#)
- [create-databases function](#)
- [create-server function](#)
- [remove-config function](#)

Before defining the functions, we define this module’s namespace as `conf-server` and import the following modules:

```
xquery version "1.0-m1";

(: Create/Remove Server/Database Configuration :)

module namespace conf-server =
  "http://marklogic.com/sampleConfig/conf-server";

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

import module namespace db=
  "http://marklogic.com/sampleConfig/database" at "databases.xqy";

import module namespace grp=
  "http://marklogic.com/sampleConfig/groups" at "groups.xqy";

import module namespace setup=
  "http://marklogic.com/sampleConfig/setup-lib"
  at "setup-library.xqy";
```

4.5.1 create-forests function

The `create-forests` function calls the functions in the [databases.xqy](#) module to create the forests:

```
declare function create-forests($config as element(configuration))
{
  try {

    (: Create forests :)
    let $config := db:create-forest(
      $config,
      "SampleDB-Forest")

    let $config := db:create-forest(
      $config,
      "SampleModules-Forest")

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

  } catch($e) {
    setup:log($e)
  }
};
```

4.5.2 create-databases function

The `create-databases` function calls the functions in the [databases.xqy](#) module to create and configure the databases to hold the application and content data for the Oscars Explorer application.

```
declare function create-databases($config as element(configuration))
{
  try {

    (: Create databases :)
    let $config := db:create-database(
      $config,
      "Sample-Database")

    let $config := db:create-database(
      $config,
      "Sample-Modules")

  }
};
```

```
(: Attach forests to databases :)
let $config := db:attach-forests(
  $config,
  "Sample-Database",
  "SampleDB-Forest")

let $config :=db:attach-forests(
  $config,
  "Sample-Modules",
  "SampleModules-Forest")

(: Add a 'wiki-suggest' field to the Sample-Database :)
let $config := db:add-field(
  $config,
  "Sample-Database",
  "wiki-suggest")

(: Add included elements to 'wiki-suggest' field :)
let $config := db:add-field-included(
  $config,
  "Sample-Database",
  "wiki-suggest",
  "name")

let $config := db:add-field-included(
  $config,
  "Sample-Database",
  "wiki-suggest",
  "film-title")

(: Add indexes to the Sample-Database :)
let $config := db:add-range-element-index(
  $config,
  "Sample-Database",
  "string",
  "http://marklogic.com/wikipedia",
  "name")

let $config := db:add-range-element-index(
  $config,
  "Sample-Database",
  "string",
  "http://marklogic.com/wikipedia",
  "film-title")

let $config := db:add-range-element-attribute-index(
  $config,
  "Sample-Database",
  "gYear",
  "http://marklogic.com/wikipedia",
  "nominee",
  "",
  "year",
  "")
```

```
let $config := db:add-range-element-attribute-index(
  $config,
  "Sample-Database",
  "string",
  "http://marklogic.com/wikipedia",
  "nominee",
  "",
  "award",
  "http://marklogic.com/collation/")

let $config := db:add-range-element-attribute-index(
  $config,
  "Sample-Database",
  "string",
  "http://marklogic.com/wikipedia",
  "nominee",
  "",
  "winner",
  "http://marklogic.com/collation/")

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

} catch($e) {
  setup:log($e)
}
};
```

4.5.3 create-server function

The `create-server` function calls the functions in the [groups.xqy](#) module to create and configure an HTTP App Server to support the Oscars Explorer application. Note that, rather than define the app server settings as parameters, the `create-server` function in the [groups.xqy](#) module parses the settings from the `$server-config` XML node defined in this function.

```

declare function create-server($config as element(configuration))
{
  try {

    (: Create a node containing the basic server settings :)
    let $server-config :=
      <http-server>
        <name>Sample-Server</name>
        <port>8016</port>
        <root>application/</root>
        <modules>Sample-Modules</modules>
        <database>Sample-Database</database>
      </http-server>

    (: Create the server :)
    let $config := grp:create-server(
      $config,
      $server-config,
      "Default")

    (: Set server error handler and url rewriter :)
    let $config := grp:set-error-handler(
      $config,
      "Default",
      "Sample-Server",
      "error.xqy")

    let $config := grp:set-url-rewriter(
      $config,
      "Default",
      "Sample-Server",
      "rewrite.xqy")

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

  } catch($e) {
    setup:log($e)
  }
};

```

4.5.4 remove-config function

The `remove-config` function calls functions in the [groups.xqy](#) and [databases.xqy](#) modules to remove the forests, databases, and app server created by the `create-server` and `create-databases` functions.

```

declare function remove-config()
{
  try {

    (: Remove the app server :)
    let $config := admin:get-configuration()
    let $config := grp:delete-server(
      $config,
      "Sample-Server",
      "Default")
    let $hosts1 := admin:save-configuration-without-restart($config)

    (: Remove the databases :)
    let $config := db:delete-database($config, "Sample-Database")
    let $config := db:delete-database($config, "Sample-Modules")
    let $hosts2 := admin:save-configuration-without-restart($config)

    (: Remove the forests :)
    let $config := db:delete-forest($config, "SampleDB-Forest")
    let $config := db:delete-forest($config, "SampleModules-Forest")

    (: Save the configuration and restart MarkLogic server :)
    return (admin:save-configuration($config),
      if ($hosts1, $hosts2)
      then (
        "Restarting Hosts -- wait until restart has completed before
        re accessing MarkLogic Server",
        admin:restart-hosts(($hosts1, $hosts2))
      )
      else ())
  } catch($e) {
    setup:log($e)
  }
};

```

4.6 groups.xqy

The `groups.xqy` module defines functions that call the `admin` functions that create, configure, and remove groups and app servers. The functions in this module are more “generic” than the configuration-specific functions in the [configure-server.xqy](#) module that call them. Consequently, the functions in this module could be applied to a variety of configurations.

- The functions in this module are:[create-group function](#)
- [create-server function](#)
- [app server configuration functions](#)
- [get-security-db function](#)
- [delete-group function](#)
- [delete-server function](#)
- [set-host function](#)

Each function in this module first checks for the presence of the object it is to create or remove before taking any action. This is done to enable partial changes to be made to the configuration without the entire transaction failing, as described in “Making Transactions Idempotent” on page 15.

Before defining the functions, we define this module’s namespace as `grp` and import the following modules:

```
xquery version "1.0-m1";

module namespace grp = "http://marklogic.com/sampleConfig/groups";
declare namespace error = "http://marklogic.com/xdmp/error";

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

import module namespace setup =
  "http://marklogic.com/sampleConfig/setup-lib"
  at "setup-library.xqy";
```

4.6.1 create-group function

The `create-group` function first checks to see whether the specified groups exists. If not, it calls the `admin:group-create` function to create the new group. This function is not called any function in the [configure-server.xqy](#) module, but could be used by other high-level modules to create another configuration.

```
declare function create-group(
  $config as element(configuration),
  $NewGroup as xs:string)
{
  try {

    let $log := setup:log(fn:concat("creating group :", $NewGroup))

    (: Get all of the existing groups :)
    let $ExistingGroups :=
      for $id in admin:get-group-ids($config)
      return admin:group-get-name($config, $id)

    (: Check to see if group already exists. If not, create new group :)
    let $config :=
      if ($NewGroup = $ExistingGroups)
      then $config
      else admin:group-create($config, $NewGroup)

    return $config

  } catch($e) {
    setup:log($e)
  }
};
```

4.6.2 create-server function

The `create-server` function first checks to see whether the specified app server exists. If not, it calls the `admin` functions to create and configure the new app server. The app server type (HTTP, WebDAV, XDBC) is specified by the root element of the `$server-config` node passed from the [configure-server.xqy](#) module.

```
declare function create-server(
  $config as element(configuration),
  $server as node(),
  $group-name as xs:string)
{
  try {

    let $group-id := admin:group-get-id($config, $group-name)
    let $NewServer := fn:data($server/name)
    let $root := fn:data($server/root)
    let $port := xs:integer(fn:data($server/port))
    let $modules := fn:data($server/modules)
```

```

let $module-id :=
  if($modules eq "filesystem")
  then 0
  else xdmp:database($modules)
let $database-id := xdmp:database(fn:data($server/database))

(: Get all of the existing servers :)
let $ExistingServers :=
  for $id in admin:get-appserver-ids($config)
  return admin:appserver-get-name($config, $id)

(: Check to see if server already exists. If not, create new server :)
let $config :=
  if ($NewServer = $ExistingServers)
  then $config
  else
    if (fn:name($server) eq "http-server")
    then
      create-http-server(
        $config,
        $group-id,
        $NewServer,
        $root,
        $port,
        $module-id,
        $database-id)
    else if (fn:name($server) eq "xdbc-server") then
      create-xdbc-server(
        $config,
        $group-id,
        $NewServer,
        $root,
        $port,
        $module-id,
        $database-id)
    else if (fn:name($server) eq "webdav-server") then
      create-webdav-server(
        $config,
        $group-id,
        $NewServer,
        $root,
        $port,
        $database-id)
    else ()

  return $config

} catch($e) {
  setup:log($e)
}
};

```

```
declare function create-xdbc-server(
  $config as element(configuration),
  $group-id,
  $server-name,
  $root,
  $port,
  $module-id,
  $database-id)
{
  try {

    let $config := admin:xdbc-server-create(
      $config,
      $group-id,
      $server-name,
      $root,
      $port,
      $module-id,
      $database-id)

    let $log := setup:log(fn:concat(
      "Creating XDBC server ",
      $server-name))

    return $config

  } catch($e) {
    setup:log($e)
  }
};

declare function create-webdav-server(
  $config as element(configuration),
  $group-id,
  $server-name,
  $root,
  $port,
  $database-id)
{
  try {

    let $config := admin:webdav-server-create(
      $config,
      $group-id,
      $server-name,
      $root,
      $port,
      $database-id)
```

```
        let $log := setup:log(fn:concat(
            "Creating WebDAV server ",
            $server-name))

        return $config

    } catch($e) {
        setup:log($e)
    }
};

declare function create-http-server(
    $config as element(configuration),
    $group-id,
    $server-name,
    $root,
    $port,
    $module-id,
    $database-id)
{
    try {

        let $config := admin:http-server-create(
            $config,
            $group-id,
            $server-name,
            $root,
            $port,
            $module-id,
            $database-id)

        let $log := setup:log(fn:concat(
            "Creating HTTP server ",
            $server-name))

        return $config

    } catch($e) {
        setup:log($e)
    }
};
```

4.6.3 app server configuration functions

The functions in this section configure the new app server. The `admin:appserver-get-id` function is used to get the id of the app server from the configuration. This enables the calling functions in the [configure-server.xqy](#) module to configure the newly created app server before saving the configuration to MarkLogic Server. For more details on the operation of `admin:appserver-get-id` and similar functions, see [Creating and Configuring Objects in a Single Transaction](#).

The following functions set the error handler and URL rewriter for the app server. As with all other functions in this module, we first check for the presence of an object before creating it.

```

declare function set-error-handler(
  $config as element(configuration),
  $group-name as xs:string,
  $server-name as xs:string,
  $ErrorHandler as xs:string)
{
  try {

    let $appserver-id :=
      admin:appserver-get-id(
        $config,
        admin:group-get-id($config, $group-name),
        $server-name)

    (: Get all of the existing error handlers for this server :)
    let $ExistingHandler :=
      admin:appserver-get-error-handler(
        $config,
        $appserver-id)

    (: Check to see if error handler is already set. If not, set
    error handler :)
    let $config :=
      if ($ErrorHandler = $ExistingHandler)
      then $config
      else
        admin:appserver-set-error-handler(
          $config,
          $appserver-id,
          $ErrorHandler)

    let $log := setup:log(fn:concat(
      "Setting error handler ",
      $server-name))

    return $config

  } catch($e) {
    setup:log($e)
  }
};

```

```
declare function set-url-rewriter(  
  $config as element(configuration),  
  $group-name as xs:string,  
  $server-name as xs:string,  
  $URLrewriter as xs:string)  
{  
  try {  
  
    let $appserver-id :=  
      admin:appserver-get-id(  
        $config,  
        admin:group-get-id($config,  
          $group-name),  
        $server-name)  
  
    (: Get all of the existing URL writers for this server :)  
    let $ExistingURLrewriterr :=  
      admin:appserver-get-url-rewriter(  
        $config,  
        $appserver-id)  
  
    (: Check to see if URL writer is already set. If not, set URL writer :)  
    let $config :=  
      if ($URLrewriter = $ExistingURLrewriterr)  
      then $config  
      else  
        admin:appserver-set-url-rewriter(  
          $config,  
          $appserver-id,  
          $URLrewriter)  
  
    let $log := setup:log(fn:concat(  
      "Setting URL rewriter ",  
      $server-name))  
  
    return $config  
  
  } catch($e) {  
    setup:log($e)  
  }  
};
```

4.6.4 get-security-db function

Because there may be more than one database named “Security,” the `get-security-db` function is used by the [install.xqy](#) and [uninstall.xqy](#) modules to identify the Security database used by the Admin app server.

```
declare function get-security-db(
  $config as element(configuration),
  $group-name as xs:string,
  $server-name as xs:string)
{
  try {

    let $appserver-id := admin:appserver-get-id(
      $config,
      admin:group-get-id($config, $group-name),
      $server-name)

    return admin:appserver-get-database(
      $config,
      $appserver-id)

  } catch($e) {
    setup:log($e)
  }
};
```

4.6.5 delete-group function

The `delete-group` function first checks to see whether the specified groups exists. If so, it calls the `admin:group-delete` function to remove the group. This function is not called any function in the [configure-server.xqy](#) module, but could be used by other high-level modules to remove a group from the configuration.

```
declare function delete-group(
  $config as element(configuration),
  $GroupName as xs:string)
{
  try {

    let $group := xdmp:group($GroupName)

    (: Get all of the existing groups :)
    let $ExistingGroups :=
      for $id in admin:get-group-ids($config)
      return admin:group-get-name($config, $id)
```

```

(: Check to see if group exists. If so, remove the group :)
  let $config :=
    if ($GroupName = $ExistingGroups)
    then admin:group-delete($config, $group)
    else $config

  return $config

} catch($e) {
  setup:log($e)
}
};

```

4.6.6 delete-server function

The `delete-server` function first checks to see whether the specified app server exists. If so, it calls the `admin:appserver-delete` function to remove the app server.

```

declare function delete-server(
  $config as element(configuration),
  $ServerName as xs:string,
  $group-name as xs:string)
{
  try {

    let $group := xdmp:group($group-name)

    (: Get all of the existing servers :)
    let $ExistingServers :=
      for $id in admin:get-appserver-ids($config)
      return admin:appserver-get-name($config, $id)

    (: Check to see if server exists. If so, remove the server :)
    let $config :=
      if ($ServerName = $ExistingServers)
      then admin:appserver-delete(
        $config,
        admin:appserver-get-id(
          $config,
          $group,
          $ServerName))
      else $config

    return $config

  } catch($e) {
    setup:log($e)
  }
};

```

4.6.7 set-host function

The `set-host` function calls the `admin:host-set-group` function to set the host for the specified group. This function is not called any function in the [configure-server.xqy](#) module, but could be used by other high-level modules to create another configuration.

```
declare function set-host(
  $config as element(configuration),
  $group-name as xs:string,
  $host-name as xs:string)
{
  try {

    let $group := xdmp:group($group-name)
    let $host := xdmp:host($host-name)

    return admin:host-set-group(
      $config,
      $host,
      $group)

  } catch($e) {
    setup:log($e)
  }
};
```

4.7 databases.xqy

The `databases.xqy` module defines functions that call the `admin` functions that create, configure, and remove the forests and databases. The functions in this module are more “generic” than the configuration-specific functions in the [configure-server.xqy](#) module that call them. Consequently, the functions in this module could be applied to a variety of configurations.

The functions in this module are:

- [create-forest function](#)
- [create-database function](#)
- [database configuration functions](#)
- [delete-database function](#)
- [delete-forest function](#)

Each function in this module first checks for the presence of the object it is to create or remove before taking any action. This is done to enable partial changes to be made to the configuration without the entire transaction failing, as described in “Making Transactions Idempotent” on page 15.

Before defining the functions, we define this module’s namespace as `db` and import the following modules:

```
xquery version "1.0-ml";

module namespace db = "http://marklogic.com/sampleConfig/database";
declare namespace error = "http://marklogic.com/xdmp/error";
declare namespace xdmpdb = "http://marklogic.com/xdmp/database";

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

import module namespace setup =
  "http://marklogic.com/sampleConfig/setup-lib"
  at "setup-library.xqy";
```

4.7.1 create-forest function

The `create-forest` function first checks to see whether the specified forest exists. If not, it calls the `admin:forest-create` function to create the new forest.

```
declare function create-forest(
  $config as element(configuration),
  $NewForest as xs:string)
{
  try {

    (: Get all of the existing forests :)
    let $ExistingForests :=
      for $id in admin:get-forest-ids($config)
      return admin:forest-get-name($config, $id)

    (: Check to see if forest already exists. If not, create new forest :)
    let $config :=
      if ($NewForest = $ExistingForests)
      then $config
      else
        admin:forest-create(
          $config,
          $NewForest,
          xdmp:host(),
          "")

    return $config

  } catch($e) {
    setup:log($e)
  }
};
```

4.7.2 create-database function

The `create-forest` function first checks to see whether the specified database exists. If not, it calls the `admin:database-create` function to create and configure the new database.

```
declare function create-database(
  $config as element(configuration),
  $NewDatabase as xs:string)
{
  try {

    let $security-db := admin:database-get-id($config, "Security")
    let $schema-db := admin:database-get-id($config, "Schemas")

    (: 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 database already exists. If not, create new
database :)
  let $config :=
    if ($NewDatabase = $ExistingDatabases)
    then $config
    else
      admin:database-create(
        $config,
        $NewDatabase,
        $security-db,
        $schema-db)

  return $config

} catch($e) {
  setup:log($e)
}
};
```

4.7.3 database configuration functions

The functions in this section configure the new database. The `admin:database-get-id` function is used to get the id of the database from the configuration. This enables the calling functions in the [configure-server.xqy](#) module to attach the forests and configure the newly created database before saving the configuration to MarkLogic Server. For more details on the operation of `admin:database-get-id` and similar functions, see [Creating and Configuring Objects in a Single Transaction](#).

The following functions attach the forests and add fields and indexes to the database. As with all other functions in this module, we first check for the presence of an object before creating it.

```
(: Attach forest to database function :)
declare function attach-forests(
  $config as element(configuration),
  $dbname as xs:string,
  $forest as xs:string)
{
  try {

    (: Get all of the forests attached to this database :)
    let $AttachedForests :=
      admin:forest-get-name(
        $config,
        (admin:database-get-attached-forests(
          $config,
          admin:database-get-id($config, $dbname)) ))
```

```

(: Check to see if forest is already attached to the database.
If not, attach the forest to the database :)
  let $config :=
    if ($forest = $AttachedForests)
    then $config
    else
      admin:database-attach-forest (
        $config,
        admin:database-get-id($config, $dbname),
        admin:forest-get-id($config, $forest) )

  return $config

} catch($e) {
  setup:log($e)
}
};

(: Add field to database function :)
declare function add-field(
  $config as element(configuration),
  $dbname as xs:string,
  $field as xs:string)
{
  try {

    let $dbid := admin:database-get-id($config, $dbname)

    (: Construct a specification for this field :)
    let $fieldspec := admin:database-field($field, fn:false())

    (: Get all of the existing fields for this database :)
    let $ExistingFields :=
      fn:data(admin:database-get-fields(
        $config,
        $dbid)/xdmpdb:field-name)

    (: Check to see if field already exists for this database.
    If not, add the field :)
    let $config :=
      if ($field = $ExistingFields)
      then $config
      else
        admin:database-add-field($config, $dbid, $fieldspec)

    return $config

  } catch($e) {
    setup:log($e)
  }
};

```

```
(: Add include element to field function :)
declare function add-field-included(
  $config as element(configuration),
  $dbname as xs:string,
  $field as xs:string,
  $included as xs:string)
{
  try {

    let $dbid := admin:database-get-id(
      $config,
      $dbname)

    (: Construct a specification for this include element :)
    let $fieldspec := admin:database-included-element(
      "http://marklogic.com/wikipedia",
      $included,
      1.0,
      "",
      "",
      "")

    (: Get all of the existing included elements for this field:)
    let $ExistingIncludes :=
      fn:data(admin:database-get-field-included-elements(
        $config,
        $dbid,
        $field)/xdmpdb:localname)

    (: Check to see if field already has the included elements.
    If not, add the included elements to the field :)
    let $config :=
      if ($included = $ExistingIncludes)
      then $config
      else
        admin:database-add-field-included-element(
          $config,
          $dbid,
          $field,
          $fieldspec)

    return $config

  } catch($e) {
    setup:log($e)
  }
};
```

```
(: Add range element index to database function :)
declare function add-range-element-index(
  $config as element(configuration),
  $dbname as xs:string,
  $type as xs:string,
  $namespace-uri as xs:string,
  $localname as xs:string)
{
  try {

    let $dbid := admin:database-get-id(
      $config,
      $dbname)

    (: Define a new range element index :)
    let $range-index := admin:database-range-element-index(
      $type,
      $namespace-uri,
      $localname,
      "http://marklogic.com/collation/",
      fn:false() )

    (: Get all of the existing range element indexes for this database :)
    let $ExistingREindexes :=
      fn:data(admin:database-get-range-element-indexes(
        $config,
        $dbid)/xdmpdb:localname)

    (: Check to see if range element index already exists for this
       database.
       If not, add the range element index :)
    let $config :=
      if ($localname = $ExistingREindexes)
      then $config
      else
        admin:database-add-range-element-index(
          $config,
          $dbid,
          $range-index)

    return $config

  } catch($e) {
    setup:log($e)
  }
};
```

```

(: Add range element attribute index to database function :)
declare function add-range-element-attribute-index(
  $config as element(configuration),
  $dbname as xs:string,
  $type as xs:string,
  $parent-namespace-uri as xs:string,
  $parent-localname as xs:string,
  $namespace-uri as xs:string,
  $localname as xs:string,
  $collation as xs:string)
{
try {

  let $dbid := admin:database-get-id(
    $config,
    $dbname)

  (: Define a new range element attribute index :)
  let $range-attribute-index := admin:database-range-element-attribute-index(
    $type,
    $parent-namespace-uri,
    $parent-localname,
    $namespace-uri,
    $localname,
    $collation,
    fn:false() )

  (: Get all of the existing range element attribute indexes for
  this database :)
  let $ExistingREIndexes :=
    fn:data(admin:database-get-range-element-attribute-indexes(
      $config,
      $dbid)/xdmpdb:localname)

  (: Check to see if range element attribute index already exists for
  this database. If not, add the range element attribute index. :)
  let $config :=
    if ($localname = $ExistingREIndexes)
    then $config
    else
      admin:database-add-range-element-attribute-index(
        $config,
        $dbid,
        $range-attribute-index)

  return $config

} catch($e) {
  setup:log($e)
}
};

```

4.7.4 delete-database function

The `delete-database` function first checks to see whether the specified database exists. If so, it calls the `admin:database-delete` function to remove the database.

```
declare function delete-database(
  $config as element(configuration),
  $DatabaseName as xs:string)
{
  try {

    let $database := admin:database-get-id($config, $DatabaseName)

    (: 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 database exists. If so, remove the database :)
    let $config :=
      if ($DatabaseName = $ExistingDatabases)
      then admin:database-delete(
        $config,
        admin:database-get-id($config, $DatabaseName))
      else $config

    return $config

  } catch($e) {
    setup:log($e)
  }
};
```

4.7.5 delete-forest function

The `delete-forest` function first checks to see whether the specified forest exists. If so, it calls the `admin:forest-delete` function to remove the forest.

```
declare function delete-forest (
  $config as element(configuration),
  $ForestName as xs:string)
{
  try {

    let $forest := admin:forest-get-id($config, $ForestName)

    (: Get all of the existing forests :)
    let $ExistingForests :=
      for $id in admin:get-forest-ids($config)
      return admin:forest-get-name($config, $id)

    (: Check to see if forest exists. If so, remove the forest :)
    let $config :=
      if ($ForestName = $ExistingForests)
      then admin:forest-delete(
        $config,
        admin:forest-get-id($config, $ForestName),
        fn:true())
      else $config

    return $config

  } catch($e) {
    setup:log($e)
  }
};
```

4.8 **configure-security.xqy**

Calls the functions in the `security` module to create new roles and users. Unlike the [configure-server.xqy](#), which calls the `admin` functions indirectly through the [groups.xqy](#) and [databases.xqy](#) modules, this module calls the `security` functions directly. Consequently, the functions in this module are specific to the configuration.

The functions in this module are:

- [configure-roles function](#)
- [configure-users function](#)
- [remove-users function](#)
- [remove-roles function](#)

Each function in this module first checks for the presence of the object it is to create or remove before taking any action. This is done to enable partial changes to be made to the configuration without the entire transaction failing, as described in “Making Transactions Idempotent” on page 15.

Before defining the functions, we define this module’s namespace as `conf-security` and import the following modules:

```
xquery version "1.0-m1";

module namespace conf-security=
  "http://marklogic.com/sampleConfig/conf-security";

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

import module namespace setup=
  "http://marklogic.com/sampleConfig/setup-lib"
  at "setup-library.xqy";
import module namespace sec="http://marklogic.com/xdmp/security"
  at "/MarkLogic/security.xqy";
```

4.8.1 configure-roles function

The `configure-roles` function first checks to see whether the specified roles exist. If not, it calls the `sec:create-role` functions to create the new roles.

```

declare function configure-roles()
{
  try {

    (: Get all of the existing roles :)
    let $ExistingRoles := fn:data(/sec:role/sec:role-name)

    (: Check to see if role already exists. If not, create new role :)
    return (

      if ("Sample" = $ExistingRoles)
      then " Sample role already exists"
      else
        sec:create-role(
          "Sample",
          "Temporary access",
          ("filesystem-access"),
          (xdmp:permission("security", "read")),
          ("testDocument")),

      if ("Developer" = $ExistingRoles)
      then " Developer role already exists"
      else
        sec:create-role(
          "Developer",
          "Developer access",
          ("filesystem-access", "pipeline-execution"),
          (xdmp:permission("security", "read"), xdmp:permission("sec
urity", "update")),
          ("testDocument")),

      if ("Tester" = $ExistingRoles)
      then " Tester role already exists"
      else
        sec:create-role(
          "Tester",
          "Tester access",
          ("filesystem-access"),
          (),
          ("testDocument"))

    )

  } catch($e) {
    setup:log($e)
  }
};

```

4.8.2 configure-users function

The `configure-users` function first checks to see whether the specified user exists. If not, it calls the `sec:create-user` function to create the new user and assign it a role created by the previous [configure-roles function](#).

```
declare function configure-users()
{
  try {

    (: Get all of the existing users :)
    let $ExistingUsers := fn:data(/sec:user/sec:user-name)

    (: Check to see if user already exists. If not, create new user :)
    return (
      if ("Jim" = $ExistingUsers)
      then " User Jim already exists"
      else
        sec:create-user(
          "Jim",
          "Jim the temp",
          "newguy",
          "Sample",
          (xdmp:permission("security", "read")),
          ("http://marklogic.com/dev_modules"))
    )

  } catch($e) {
    setup:log($e)
  }
};
```

4.8.3 remove-users function

The `remove-users` function first checks to see whether the specified user exists. If so, it calls the `sec:remove-user` function to remove the user.

```
declare function remove-users()
{
  try {

    (: Get all of the existing users :)
    let $ExistingUsers := fn:data(/sec:user/sec:user-name)

    (: Check to see if user exists. If so, remove user :)
    return (
      if ("Jim" = $ExistingUsers)
      then sec:remove-user("Jim")
      else ()
    )

  }
};
```

```
    } catch($e) {  
        setup:log($e)  
    }  
};
```

4.8.4 remove-roles function

The `remove-roles` function first checks to see whether the specified roles exist. If so, it calls the `sec:remove-role` functions to remove the roles.

```
declare function remove-roles()  
{  
  try {  
  
    (: Get all of the existing roles :)  
    let $ExistingRoles := fn:data(/sec:role/sec:role-name)  
  
    (: Check to see if role exists. If so, remove role :)  
    return (  
  
      if ("Sample" = $ExistingRoles)  
      then sec:remove-role("Sample")  
      else (),  
  
      if ("Developer" = $ExistingRoles)  
      then sec:remove-role("Developer")  
      else (),  
  
      if ("Tester" = $ExistingRoles)  
      then sec:remove-role("Tester")  
      else ()  
    )  
  
  } catch($e) {  
    setup:log($e)  
  }  
};
```

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:

- [Group Maintenance Operations](#)
- [App Server Maintenance Operations](#)
- [Database Maintenance Operations](#)
- [Host Maintenance Operations](#)
- [User Maintenance Operations](#)

5.1 Group Maintenance Operations

The operations for creating and deleting groups are described in “Creating and Configuring Groups” on page 30. 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:

- [Enabling Auditing on a Group](#)
- [Disabling Auditing on a Group](#)
- [Removing Events to be Audited on a Group](#)
- [Adding a Namespace to a Group](#)
- [Returning the Namespace Settings on a Group](#)
- [Deleting a Namespace from a Group](#)
- [Returning the System Log Settings](#)
- [Resetting the System Log Settings](#)
- [Creating a New Hourly Task](#)
- [Deleting all Scheduled Tasks from a Group](#)

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-m1";
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-m1";
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 33. 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 *Administrator's 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-m1";
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 83:

```
xquery version "1.0-m1";
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-m1";
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-m1";
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 21. 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](#)
- [Disabling Database Merges](#)
- [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](#)
- [Backing Up a Database](#)
- [Restoring from a Previous 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("myOldData"),
  "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 Disabling Database Merges

The following script disables merges on 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:save-configuration(
  admin:database-set-merge-enable(
    $config,
    xdmp:database("Sample-Database"),
    fn:false()))
```

5.3.4 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-m1";
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))
```

5.3.5 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-m1";
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))
```

5.3.6 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-m1";
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"))
```

5.3.7 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-m1";
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))
```

5.3.8 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>)
```

5.3.9 Backing Up a Database

The following script immediately backs up 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")
```

5.3.10 Restoring from a Previous Backup

The following script restores the “Sample-Database” database from the backup taken on 3/19/2009:

```
xquery version "1.0-ml";

xdmp:database-restore (
  xdmp:database-forests (xdmp:database ("Sample-Database")),
  "c:/backup-dir/20090319-1")
```

5.3.11 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)
```

5.3.12 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")))

```

5.3.13 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 (
        xdm:log(fn:concat(
            "Setting ",
            $f_name,
            " forest with updates set to ",
            $f_updates,
            " forest to 'all'")),

        xdm: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-m1";

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($config))
        then xdm:set($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 :=  
  xdm:database-forests(xdm: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](#)
- [Returning the Time Host was Last Started](#)
- [Restarting MarkLogic Server on all Hosts in the Cluster](#)

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 $id in xdmp:host-status(xdmp:host())
  /host:hosts//host:host/host-id
  return fn:data($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 20.

The topics in this section are:

- [Removing all Users with a Specific Role](#)
- [Removing a Specific Role, if Present](#)
- [Retrieving the Last-Login Information](#)

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' (893641345095093063) who last
successfully logged in on 2008-07-15T16:13:54-07:00.
```

6.0 Scripting Content Processing Framework (CPF) Configuration

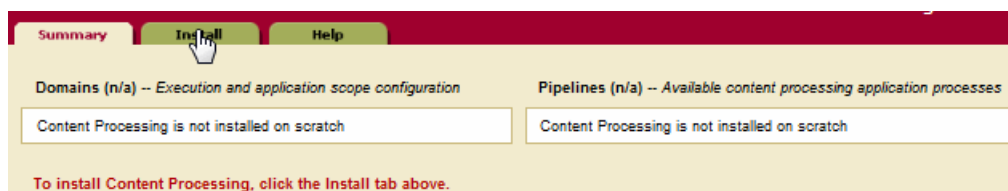
The MarkLogic Server Content Processing Framework (CPF) is described in detail in the *Content Processing Framework Guide* Guide. This chapter describes how to use the CPF API to programmatically configure CPF. The main topics in this chapter are:

- [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 105.

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-m1";

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 107.

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 Replicated Content. The scope of the domain is the root directory of the content database that uses the domain. 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-m1";

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 pipelines and domains, call the `dom:configuration-create` function to configure your database with a restart trigger. The restart trigger needs to be 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";

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

7.0 Scripting Flexible Replication Configuration

This chapter describes how to use the Flexible Replication API to configure flexible replication. For details on flexible replication, see the *Flexible Replication Guide*. The main topics in this chapter are:

- [Preliminary Configuration Procedures](#)
- [Configuring the Master Database](#)
- [Creating a Replication Configuration Element](#)
- [Creating a Replication Target](#)
- [Creating a Push Replication Scheduled Task](#)
- [Configuring Pull Replication](#)

7.1 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 22.

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 22.

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 23.

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, xdm:database("Master"),
  xdm: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 104.
- 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 105.

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 34.

7.2 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)
```

7.3 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)
```

7.4 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)
```

7.5 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-m1";

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)
```

7.6 Configuring Pull Replication

This section describes how 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](#)

7.6.1 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)
```

7.6.2 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)

```

7.6.3 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-m1";

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)

```

8.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement. For evaluation licenses, MarkLogic may provide support on an “as possible” basis.

For customers with a support contract, we invite you to visit our support website at <http://support.marklogic.com> to access information on known and fixed issues.

For complete product documentation, the latest product release downloads, and other useful information for developers, visit our developer site at <http://developer.marklogic.com>.

If you have questions or comments, you may contact MarkLogic Technical Support at the following email address:

support@marklogic.com

If reporting a query evaluation problem, please be sure to include the sample XQuery code.