

---

# MarkLogic Server

---

## Understanding and Using Security

Release 3.2  
May, 2007

## Copyright

© Copyright 2002-2008 by Mark Logic 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 Mark Logic 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](http://www.inxight.com).

Antenna House OfficeHTML Copyright © 2000-2006 Antenna House, Inc. All rights reserved.

Argus Copyright ©1999-2005 Icen Technology Ltd. All rights reserved.

---



---

## Table of Contents

---

### Understanding and Using Security

Copyright .....	2
1.0 Introduction to Security .....	6
1.1 Security Overview .....	6
1.1.1 Authentication .....	6
1.1.2 Authorization .....	7
1.1.3 Administration .....	7
1.2 Components of the Security Model .....	7
1.2.1 Role-Based Security Model (Authorization) .....	7
1.2.2 Access Control With the Security Database .....	9
1.2.3 Security Administration With the Admin Interface and XQuery Functions 10	10
1.3 Terminology .....	10
1.3.1 User .....	10
1.3.2 Role .....	10
1.3.3 Execute Privilege .....	10
1.3.4 URI Privilege .....	11
1.3.5 Permission .....	11
2.0 Roles and the Role-Based Security Model .....	12
2.1 Understanding Roles .....	12
2.1.1 Assigning Privileges to Roles .....	12
2.1.1.1 Execute Privileges .....	12
2.1.1.2 URI Privileges .....	13
2.1.2 Associating Permissions With Roles .....	13
2.1.3 Default Permissions in Roles .....	13
2.1.4 Assigning Users to Roles .....	13
2.1.5 Roles, Privileges, Document Permissions, and Users .....	14
2.2 The admin and security Roles .....	15
2.3 Example—Introducing Roles, Users and Execute Privileges .....	15
3.0 URI Privileges and Permissions on Documents .....	18
3.1 Creating Documents .....	18
3.1.1 URI Privileges .....	18
3.1.2 Built-In any-uri and unprotected-uri Execute Privileges .....	19
3.2 Permissions on Documents .....	19
3.2.1 Capabilities Associated Through Permissions .....	20

3.2.1.1	Read .....	20
3.2.1.2	Update .....	20
3.2.1.3	Insert .....	20
3.2.1.4	Execute .....	21
3.2.2	Setting Document Permissions .....	21
3.3	Securing Collection Membership .....	21
3.4	Default Permissions .....	22
3.5	Example—Using Permissions .....	22
3.5.1	Explicit Parameter .....	23
3.5.2	Default Permission Settings .....	23
4.0	Execute Privileges and Amps .....	27
4.1	Execute Privileges .....	27
4.1.1	Using Execute Privileges .....	27
4.1.2	Execute Privileges and HTTP, WebDAV, and XDBC Servers .....	28
4.1.3	Execute Privileges When Creating and Updating Collections .....	28
4.2	Amps .....	28
5.0	Users and Authentication .....	30
5.1	Users .....	30
5.2	Types of Authentication .....	30
5.2.1	Basic .....	30
5.2.2	Digest .....	31
5.2.3	Digest-Basic .....	31
5.2.4	Limitations of Digest and Basic Authentication .....	31
5.2.5	Application Level .....	31
6.0	Administering Security .....	32
6.1	Overview of the Security Database .....	32
6.2	Associating a Security Database With a Documents Database .....	33
6.3	Managing and Using Objects in the Security Database .....	34
6.3.1	Using the Admin Interface .....	34
6.3.2	Using the security.xqy Module Functions .....	34
6.4	Backing Up the Security Database .....	34
6.5	Example: Using the Security Database in Different Servers .....	35
7.0	Sample Scenarios For Defining Security Policies .....	38
7.1	General Steps for Creating and Implementing Security Policies .....	38
7.2	Restricting Document Access to a Particular Program .....	39
7.3	Choosing the Access Control Option for an Application .....	40
7.3.1	No Access Restriction .....	40
7.3.2	Uniform Access for All Users in the Security Database .....	40
7.3.3	Limiting Application Access to a Subset of Users .....	41
7.3.4	Custom Login Pages .....	42

7.3.5 Access Control Based on Client IP Address .....	43
Technical Support .....	48

## 1.0 Introduction to Security

Whenever you create systems that store and retrieve data, the importance of providing secure access to the data and of having a fine level of control over such data access becomes clear the first time a user needs to access the data. Does this user have the proper authority to see the data? Should the user be able to load new data or update existing data in the system? Questions like these (and their answers) are necessary to provide the basis for the security requirements for your application.

MarkLogic Server includes a powerful role-based security model which has the flexibility to protect your data according to your application security requirements. There is always a trade-off between security and usability. If a system has no security, then it is open to malicious or unmalicious unauthorized access; if a system is too locked down, however, then it might become too difficult (or impossible) for the people who need to use it. Before implementing your application security model, it is very important to understand the core concepts and features in the MarkLogic Server security model. This chapter introduces those concepts and includes the following sections:

- [Security Overview](#)
- [Components of the Security Model](#)
- [Terminology](#)

### 1.1 Security Overview

This section provides an overview of the three main principles used in security in a MarkLogic Server database:

- [Authentication](#)
- [Authorization](#)
- [Administration](#)

#### 1.1.1 Authentication

*Authentication* is the process of verifying user credentials for a named user. In other words, authentication makes sure you are who you say you are. Users are typically authenticated with a username and password. Every request to MarkLogic Server is issued from an authenticated user. There are several ways to set up server authentication in MarkLogic Server. Authentication simply verifies user credentials and associates that session with the authenticated user; it does not grant any access or authority to perform any actions on the system. For details on authentication, see “Users and Authentication” on page 30.

### 1.1.2 Authorization

*Authorization* provides the mechanism to control document access, XQuery code execution, and document creation. In other words, given the user who you are authenticated as, authorization allows the system to control what you are allowed to do. For example, authorization is the process that can allow the user named *Melanie* to be able to read and update a document, the user named *Roger* to only be able to read the document, and the user named *Hal* to not even have access to the document. Authorization is used to protect documents and to protect the execution of XQuery code. For details on authorization in MarkLogic Server, see “URI Privileges and Permissions on Documents” on page 18 and “Execute Privileges and Amps” on page 27.

### 1.1.3 Administration

*Administration* is the process of defining users, roles, privileges, and permissions to set up your security policies. Administration is used to create and manage the objects used in authentication and authorization. For details on how security administration works in MarkLogic Server, see “Security Administration With the Admin Interface and XQuery Functions” on page 10 and the *Administrator’s Guide*.

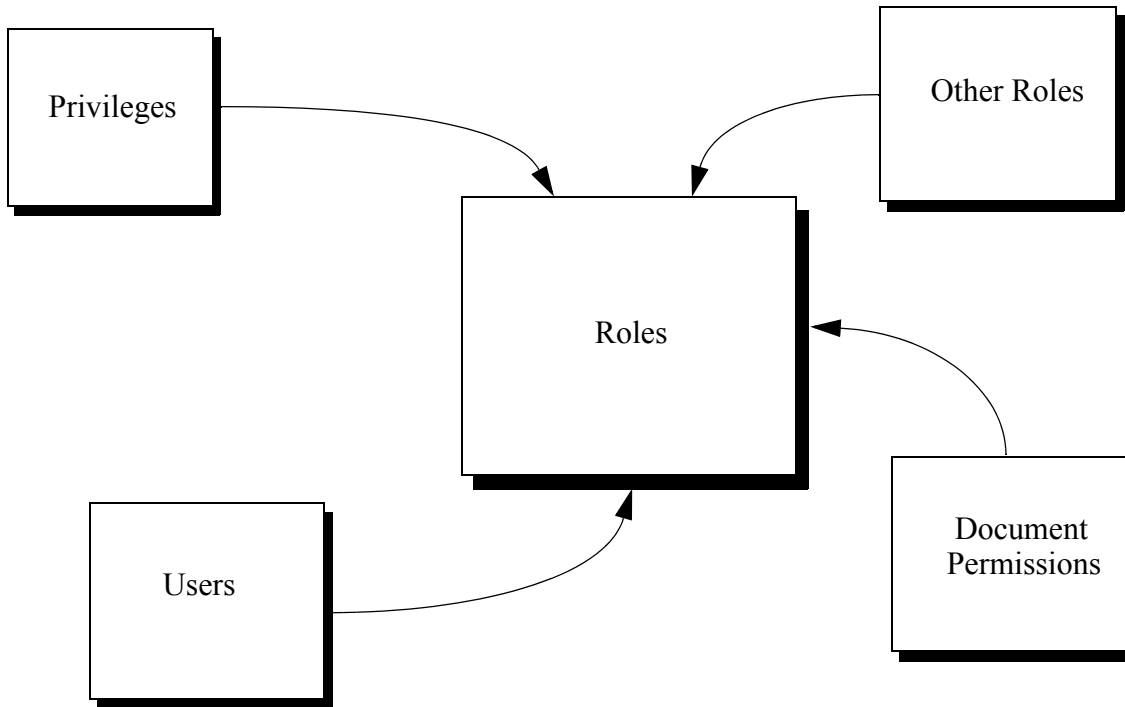
## 1.2 Components of the Security Model

The MarkLogic Server security model is very flexible, allowing you to set up application security with the level of granularity dictated by your security requirements. The security model has the following components:

- [Role-Based Security Model \(Authorization\)](#)
- [Access Control With the Security Database](#)
- [Security Administration With the Admin Interface and XQuery Functions](#)

### 1.2.1 Role-Based Security Model (Authorization)

Roles are the central point of authorization in the MarkLogic Server security model. Privileges, users, other roles, and document permissions all point to roles. The following figure shows how each of these entities points into one or more roles.



Privileges are assigned zero or more roles, roles are assigned to zero or more other roles, and users are assigned to zero or more roles. Documents are assigned permissions based on roles and capabilities (`read`, `insert`, `update`, `execute`). Users inherit the sum of all of the privileges and permissions from their roles.

There are two types of privileges: URI privileges and execute privileges. URI privileges are used to control the creation of documents with certain URIs. Execute privileges are used to protect the execution of functions in XQuery code. A privilege is like a door and, when the door is locked, you need to have the key to the door in order to open it. If the door is unlocked (no privileges), then you can walk right through. The keys to the doors are distributed to users through roles; that is, if a user inherits a privilege through the set of roles to which she is assigned, then she has the keys to unlock those inherited privileges.

Permissions are used to protect documents. Documents are assigned permissions, either at load time or as a separate administrative action. Each permission is a combination of a role and a capability (`read`, `insert`, `update`, `execute`).

#### Permission

Role	Capability (read, insert, update, <b>or</b> execute)
------	--

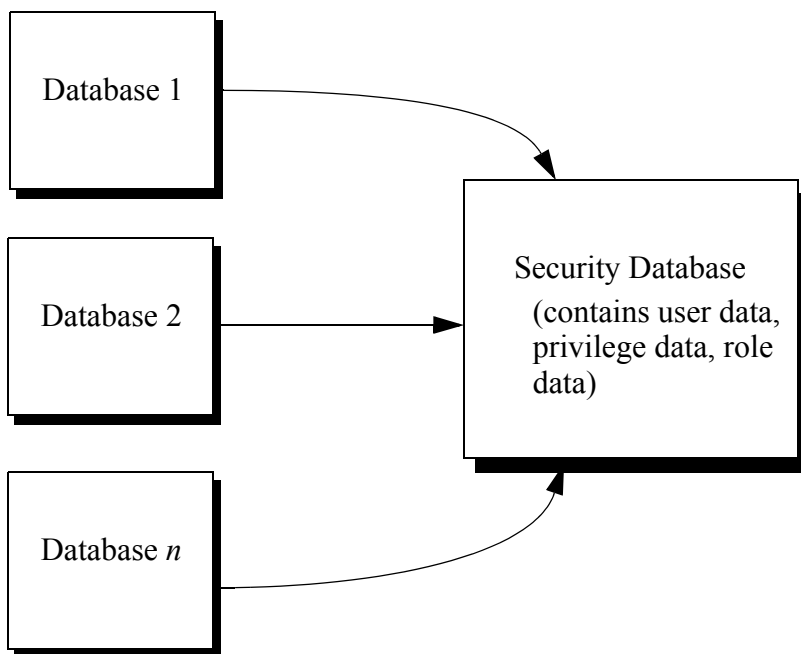
Users assigned the role corresponding to the permission have the ability to perform the capability. You can set any number of permissions on a document.

For more details on how roles work in MarkLogic Server, see “Roles and the Role-Based Security Model” on page 12. For more details on privileges and permissions, see “URI Privileges and Permissions on Documents” on page 18.

### 1.2.2 Access Control With the Security Database

The *security database* is a MarkLogic Server database in which the user data, privilege data, role data, and other security information is stored. Each database in MarkLogic Server references a security database. A database named *Security* is created as part of the installation process, and the *Security* database functions as the default security database.

The following figure shows that many databases can reference the same security database for authentication and authorization.



The security database is accessed to authenticate users and to control access to documents. For details on authentication, the security database, and ways to administer objects in the security database, see “Users and Authentication” on page 30 and “Administering Security” on page 32.

### 1.2.3 Security Administration With the Admin Interface and XQuery Functions

MarkLogic Server administrators are privileged users who have the authority to perform tasks such as creating/deleting/modifying users, roles, privileges, and so on. These tasks change or add data in the security database. Users who perform these tasks must have the *security* role, either explicitly or by inheriting it from another role (for example, from the *admin* role). Typically, users who perform these tasks have the *admin* role, which provides the authority to perform any tasks in the database. Use caution when assigning users to the *security* and/or *admin* roles; users who belong to the *admin* role can perform any task on the system, including deleting data.

There are two main ways to administer security in MarkLogic Server: using the Admin interface and using the XQuery functions supplied to administer security. For details on administering security, see “Administering Security” on page 32.

## 1.3 Terminology

This section defines the following terms, which are used throughout the security documentation:

- [User](#)
- [Role](#)
- [Execute Privilege](#)
- [URI Privilege](#)
- [Permission](#)

### 1.3.1 User

A *user* is a named entity used to authenticate a request to an HTTP, WebDAV, or XDBC server. For details on users, see “Users and Authentication” on page 30.

### 1.3.2 Role

A *role* is a named entity that provides authorization privileges and permissions to other roles or to users. Roles can include assignments to other roles (which can in turn include assignments to other roles, and so on). Roles are the fundamental building block with which you build your security policies. For details on roles, see “Roles and the Role-Based Security Model” on page 12.

### 1.3.3 Execute Privilege

An *execute privilege* provides the authority to perform a protected action. Examples of protected actions are the ability to execute a specific user-defined function, the ability to execute a built-in function (for example, `xdmp:document-insert`), and so on. For details on execute privileges, see “Execute Privileges and Amps” on page 27.

### 1.3.4 URI Privilege

A *URI privilege* provides the authority to create documents within a base URI. When a URI privilege exists for a base URI, only users with roles assigned to the URI privilege can create documents with URIs starting with the base string. For details on URI privileges, see “URI Privileges and Permissions on Documents” on page 18.

### 1.3.5 Permission

A *permission* provides a role with the capability to perform certain capabilities (`read`, `insert`, `update`, `execute`) on a document or a collection. Permissions are assigned to documents and collections. For details on permissions, see “URI Privileges and Permissions on Documents” on page 18.

## 2.0 Roles and the Role-Based Security Model

MarkLogic Server uses a role-based security model. Each security entity is associated with a role. This chapter describes the components of the role-based security model and includes the following sections:

- [Understanding Roles](#)
- [The admin and security Roles](#)
- [Example—Introducing Roles, Users and Execute Privileges](#)

### 2.1 Understanding Roles

As described in “Role-Based Security Model (Authorization)” on page 7, roles are the central point of authorization in MarkLogic Server. This section describes how the other security entities relate to roles, and includes the following sections:

- [Assigning Privileges to Roles](#)
- [Associating Permissions With Roles](#)
- [Default Permissions in Roles](#)
- [Assigning Users to Roles](#)
- [Roles, Privileges, Document Permissions, and Users](#)

#### 2.1.1 Assigning Privileges to Roles

Privileges are used to control access to XQuery code (execute privileges) or to control access to creating documents in a given URI range (URI privileges). You associate roles with privileges by assigning privileges to roles.

##### 2.1.1.1 Execute Privileges

Execute privileges allow developers to control authorization for the execution of an XQuery function. You can add code to any user-defined XQuery function to check if the user executing the code has a certain privilege. That privilege is assigned to a user through a role. Also, there are many execute privileges predefined in the security database to control authorization to execute a variety of built-in XQuery functions.

For more details on execute privileges, see “Execute Privileges and Amps” on page 27.

### 2.1.1.2 URI Privileges

URI privileges control authorization for creation of a document with a given URI prefix. In order to create a document with a prefix that has a URI privilege associated with it, a user must be part of a role to which the needed URI privilege is assigned.

For more details on how URI privileges interact with document creation, see “URI Privileges and Permissions on Documents” on page 18.

### 2.1.2 Associating Permissions With Roles

Permissions are characteristics of documents that associate a role with a capability (`read`, `insert`, `update`, or `delete`). Users gain the authority to perform these capabilities on a document if they are part of a role to which a permission is associated.

For more details on how permissions interact with documents, see “Permissions on Documents” on page 19.

### 2.1.3 Default Permissions in Roles

Roles are one of the places where you specify *default permissions*. Default permissions are used when creating documents. They are a set of initial permissions that are applied to a document at the time the document is created (if specific permissions are not explicitly specified). The system determines the default permissions for a user based on the roles the user is part of, calculating the total set of default permissions from all inherited roles.

For more details on how default permissions interact with document creation, see “Default Permissions” on page 22.

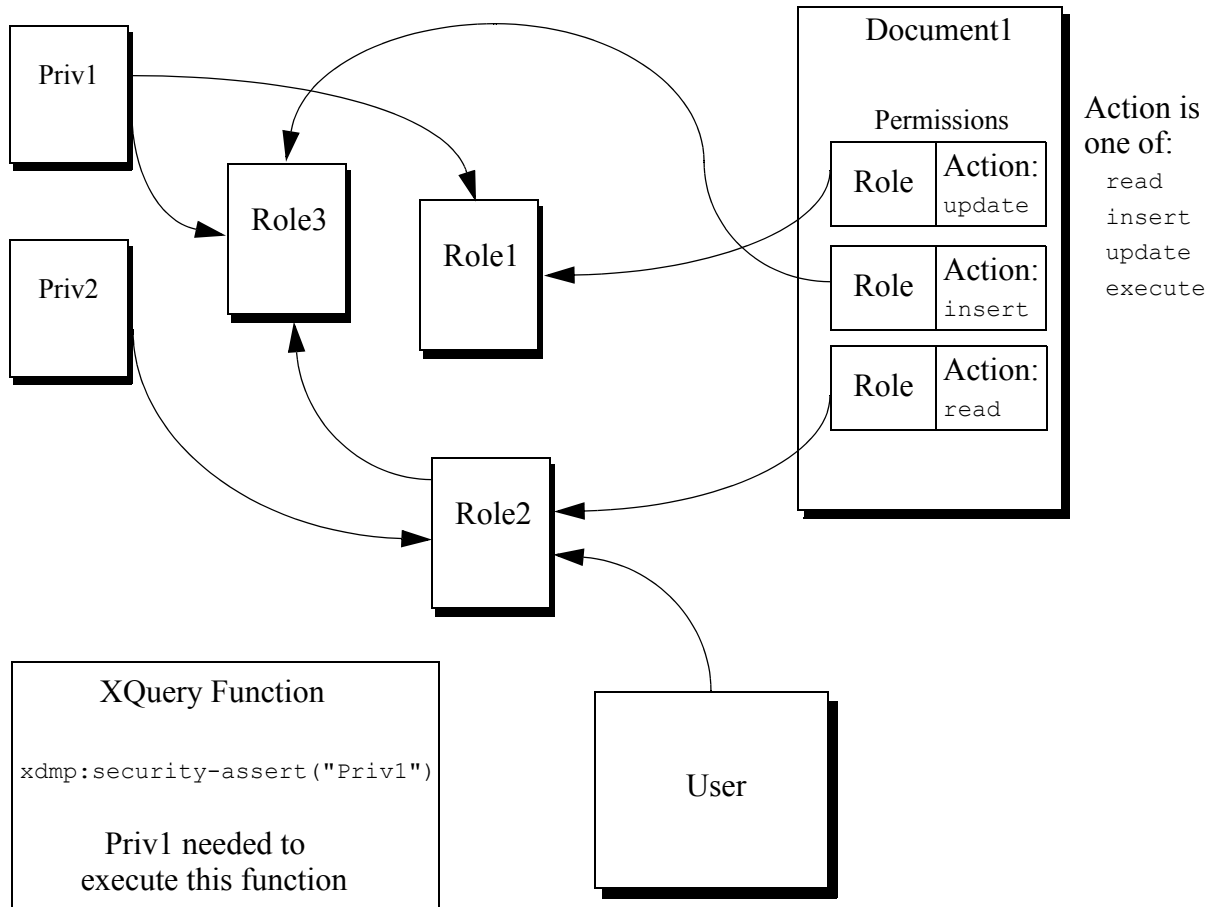
### 2.1.4 Assigning Users to Roles

When accessing a database, users are authenticated against the security database associated with that database. Roles are the mechanism from which authorization entities (privileges, permissions, default permissions) are carried to code and documents. You assign a user to roles, and the roles provide that user with the authority to perform actions against code and documents. The roles provide the user with a set of privileges and permissions. At any given time, a user *possesses* this set of privileges and permissions; this authorization set is the sum of all of the privileges and permissions inherited from all of the roles to which that user is currently assigned. Use the Admin interface to display the set of privileges and permissions for a given user; do not try and calculate it yourself as it can get easily get fairly complex when a system has many roles.

For more details on users, see “Users and Authentication” on page 30.

### 2.1.5 Roles, Privileges, Document Permissions, and Users

Privileges, document permissions, and users all interact with roles to define your security policies. The following diagram shows an example of how these entities interact.



Notice how all of the arrows point into the roles; that is because the roles are the center of all security administration in MarkLogic Server. In this diagram, the user is part of *Role2*, and *Role2* inherits *Role3*. Therefore, even though the user has only been assigned *Role2*, she possesses all of the privileges and permissions from both *Role2* and *Role3*. Following the arrows pointing into *Role2* and *Role3*, the user possesses *Priv1*, *Priv2*, and `insert` and `read` permissions on *Document1*.

Because the user possesses *Priv1* (based on role inheritance), she will be able to execute code protected with a `xdmp:security-assert("Priv1", "execute")` call; users who do not have the *Priv1* privilege will not be able to execute such code.

## 2.2 The admin and security Roles

MarkLogic Server has a special role named *admin*. The *admin* role has full authority to do everything in MarkLogic Server, regardless of any permissions or privileges set. In general, the *admin* role is only for administrative activities and should not be used to load data and run applications. Use extreme caution when assigning users the *admin* role, because it gives them the authority to perform any activity on the system, included adding or deleting users, adding or deleting documents, changing passwords, and so on.

MarkLogic Server also has a built-in role named *security*. Users who are part of the *security* role have execute privileges to perform security-related tasks on the system using the functions in the `security.xqy` module file.

**Note:** The *security* role does not have enough privileges to allow access to the Admin interface; to access the Admin interface, a user must have the *admin* role. The *security* role does provide the privileges to execute functions in the `security.xqy` module file, which has functions to perform actions such as creating users, creating roles, and so.

## 2.3 Example—Introducing Roles, Users and Execute Privileges

Consider a simple system with two roles – *engineering* and *sales*. The *engineering* role is responsible for making widgets and has privileges needed to perform activities related to making widgets. The *sales* role is responsible for selling widgets and has privileges to perform activities related to selling widgets.

To begin, create two roles in MarkLogic Server named *engineering* and *sales* respectively.

The *engineering* role needs to be able to make widgets. You can create an execute privilege with the name *make-widget*, and action URI `http://widget.com/make-widget` to represent that privilege. The *sales* role needs to sell widgets so you create an execute privilege with the name *sell-widget* and action URI `http://widget.com/sell-widget` to represent that privilege.

**Note:** Names for execute privileges are used only as display identifiers in the Admin interface. The action URIs are used within XQuery code to identify the privilege.

Ron is an engineer in your company so you create a user for Ron and assign the *engineering* role to the newly created user. Emily is an account representative so you create a user for Emily and assign her the *sales* role.

In your XQuery code, use `xdmp:security-assert(<action-uri>, <privilege-type>)` to ensure that only engineers make widgets and only account representatives sell widgets. For example:

```
define function make-widget(...) as ...
{
  xdmp:security-assert("http://widget.com/make-widget",
    "execute"), make-widget...
}
```

If Ron is logged into the application and executes the `make-widget()` function, `xdmp:security-assert("http://widget.com/make-widget", "execute")` succeeds since Ron is of the engineering role which has the execute privilege to make widgets.

If Emily attempts to execute the `make-widget()` function, `xdmp:security-assert("http://widget.com/make-widget", "execute")` throws an exception. You can catch the exception and handle it with a `try/catch` in the code. If the exception is not caught, the transaction that called this function is rolled back.

Some functions are common to several protected actions. You can protect such a function with a single `xdmp:security-assert()` call by providing the appropriate action URIs in a list. For example, if a user needs to execute the `count-widgets()` function when making or selling widgets, you might protect the function as follows:

```
define function count-widgets(...) as ...
{
  xdmp:security-assert( ("http://widget.com/make-widget",
    "http://widget.com/sell-widget"), "execute"),
  count-widget...
}
```

If there is a function that requires more than one privilege before it can be performed, place the `xdmp:security-assert()` calls sequentially. For example, if you need to be a manager in the sales department to give discounts when selling the widgets, you can protect the function as follows:

```
define function discount-widget(...) as ...
{
  xdmp:security-assert( "http://widget.com/sell-widget",
    "execute"),
  xdmp:security-assert( "http://widget.com/change-price",
    "execute"),
  discount widget...
}
```

where `http://widget.com/change-price` is an action URI for a *change-price* execute privilege assigned to the *manager* role. A user needs to have the *sales* role and the *manager* role, which provides the user with the *sell-widget* and *change-price* execute privileges, to be able to execute this function.

## 3.0 URI Privileges and Permissions on Documents

The MarkLogic Server security model has a set of tools used to control access to documents. These authorization tools control creating, inserting into, updating, and reading documents in a database. This chapter describes those tools and includes the following sections:

- [Creating Documents](#)
- [Permissions on Documents](#)
- [Securing Collection Membership](#)
- [Default Permissions](#)
- [Example—Using Permissions](#)

### 3.1 Creating Documents

In order to have the authority to create a document in a MarkLogic Server database, a user must possess the needed privileges to create a document with a given URI. The ability to create documents based on the URI specified is controlled with URI privileges and with two built-in execute privileges (`any-uri` and `unprotected-uri`). In order for a user to possess a privilege, the user must be part of a role (either directly or indirectly, through role inheritance) to which the privilege is assigned. This section describes these different privileges.

#### 3.1.1 URI Privileges

URI privileges control the ability to create a new document with a given URI prefix. Creating a URI privilege with a given URI protects that URI from new document creation; only users possessing the URI privilege can create a new document with the prefix.

For example, the screenshot below shows a URI privilege with `/widget.com/sales/` as the protected URI. Any URI with `/widget.com/sales/` as the prefix will be protected, and users must be part of the sales role in order to create documents with URIs beginning with this prefix. In this example, you need this URI privilege (or a privilege with at least as much authority) to create a document with the URI `/widget.com/sales/my_process.xml`.

**New URI Privilege** ok cancel

**uri privilege** -- *Privilege representation.*

**privilege name**   
Privilege name (unique)  
**Required. You must supply a value for privilege-name.**

**uri**   
A URI to protect.  
**Required. You must supply a value for action.**

**roles** -- *The roles assigned.*

- admin
- admin-builtins
- domain-management
- filesystem-access
- merge
- pipeline-execution
- pipeline-management
- read
- sales

### 3.1.2 Built-In any-uri and unprotected-uri Execute Privileges

The following built-in execute privileges control the creation of URIs:

- `any-uri`
- `unprotected-uri`

The `any-uri` privilege provides the authority to create a document with any URI in the database, even if the URI is protected with a URI privilege. The `unprotected-uri` privilege provides the authority to create a document at any URI in the database except for URIs that are protected with a URI privilege.

## 3.2 Permissions on Documents

Permissions on a document control access to capabilities (`read`, `insert`, `update`, and `execute`) on that document. Each permission consists of a capability and a corresponding role. This section describes the capabilities for permissions and describes the ways to set permissions on a document. It includes the following subsections:

- [Capabilities Associated Through Permissions](#)
- [Setting Document Permissions](#)

### 3.2.1 Capabilities Associated Through Permissions

Permissions on a document pair a role with a capability, and you can add multiple permissions to a document. The capability provides any user who possesses the permission with the capability of performing that action. Users possess these capabilities through roles; if the user is part of the role (either directly or through inheriting the role) with the corresponding capability for a document, then the user has that capability for the given document. Each permission associates a role with one of the following capabilities:

- [Read](#)
- [Update](#)
- [Insert](#)
- [Execute](#)

#### 3.2.1.1 Read

The `read` capability provides the authority to see the content in the document. Being able to see the content does not allow you to make any changes to the document.

#### 3.2.1.2 Update

The `update` capability provides the authority to modify content in the document as well as delete the document. However, it does not provide the authority to read the document; the ability to read the document requires the addition of the `read` capability. Without `read` capability, users with `update` capability can call `xdmp:document-delete()` and `xdmp:document-insert()`. However, any node update function (`xdmp:node-replace()`, `xdmp:node-delete()` and `xdmp:node-insert()`) cannot be called. Node update functions require a node from the document as a parameter. If a user cannot read the document, he cannot access the node in the document and supply it as a parameter.

There is a way to get around the issue with node update functions. The `update` capability provides the authority to change the permissions on a document. Therefore, you can use `xdmp:add-permissions()` to add a new permission to the document with `read` capability for a given role. Once the user has both `read` and `update` capabilities, he is able to call node update functions.

#### 3.2.1.3 Insert

The `insert` capability has a subset of the capabilities of the `update` capability. The `insert` capability on a document provides the authority to add new content to the document. The `insert` capability by itself does not allow a user to change or remove an existing document (that is, calls to `xdmp:document-insert()` and `xdmp:document-delete()` on an existing document will fail). Furthermore, you need the `read` capability on the document to perform actions that use any of the node insert functions (`node-insert-before()`, `node-insert-after()`, `node-insert-child()`), as explained above in the description for `update`. Therefore, a permission with an `insert` capability must be paired with a permission with a `read` capability to be useful.

### 3.2.1.4 Execute

The `execute` capability on a document provides the authority to execute application code contained in that document, if the document is stored in a database which is configured as a modules database. Without possessing a permission with `execute` capability on a stored module, users will not be able to execute that module.

## 3.2.2 Setting Document Permissions

When you create documents in a database, you must think about setting permissions on the document. If a document has no permission set on it, no one (other than users with the admin role) can read, update, insert, or delete it.

You can set permissions on a document in the following ways:

- You can explicitly set permissions on a document at load time (as a parameter to `xdmp:load` or `xdmp:document-insert`, for example).
- You can explicitly set and remove permissions on a document with the `xdmp:document-add-permissions`, `xdmp:document-set-permissions`, and `xdmp:document-remove-permissions` functions.
- You can implicitly set the permissions on a document at document creation time based on the default permissions of the user who creates the document.

For examples of setting permissions on documents, see “Example—Using Permissions” on page 22.

## 3.3 Securing Collection Membership

Just like you can assign permissions to a document that map roles with capabilities (`read`, `insert`, `update`, `execute`), you can also secure membership in collections by assigning permissions to collections. The difference with collections is that you must use the Admin interface or the `security.xqy` XQuery functions to assign permissions to collections; you cannot assign permissions to collections implicitly with default permissions.

For more information about permissions on collections, see “Introduction to Collections” in the *Developer’s Guide*.

### 3.4 Default Permissions

When a document is created, it is initialized with a set of permissions. If permissions are not explicitly set (as a parameter to `xmmp:load` or `xmmp:document-insert`, for example), then the permissions are set to the *default permissions*. The default permissions are determined based on the roles assigned (both assigned explicitly and inherited from roles assigned to other roles) to the user who creates the document and on any default permissions assigned directly to the user.

If users will be creating documents in a database, it is very important to set up default permissions for roles to which that user is assigned. Without default permissions, it is easy to create documents that no users (except those who are part of the *admin* role) can read, update, or delete.

### 3.5 Example—Using Permissions

It is very important to consider document permissions whenever you load content into a database, whether you load data using the built-in functions (for example, `xmmp:load` or `xmmp:document-insert`), WebDAV (for example, dragging and dropping files into a WebDAV folder), or a custom program to load data. In each case, setting permissions, whether explicitly or using default permissions, is necessary if you have any security policies set up. This example shows several ways of setting permissions on documents.

Suppose that Ron of the *engineering* role is given the task to create a document to describe new features that will be added to the next version of the widget. Once the document is created, other users of the *engineering* role will contribute to the document and add the features they are working on. Ian, of the *engineering-manager* role decides that users of the *engineering* role should only be allowed to read and add to the document. This enables Ian to control the process of removing or changing features in the document. The document will therefore be created with `read` and `insert` permissions for the *engineering* role, and `read` and `update` permissions for the *engineering-manager* role.

There are two ways to apply permissions to documents at creation time:

- [Explicit Parameter](#)
- [Default Permission Settings](#)

### 3.5.1 Explicit Parameter

Assume that the following code snippet is executed as Ron who has the *engineering* role. The code inserts a document with read and insert permissions for the *engineering* role and update and read permissions for the *engineering-manager* role.

```
...
xdmp:document-insert("/widget.com/engineering/features/2004-q1.xml",
  <new-features>
    <feature>
      <name>blue whistle</name>
      <assigned to>Ron</assigned to>
      ...
    </feature>
    ...
  </new-features>,
  (xdmp:permission("engineering", "read"),
   xdmp:permission("engineering", "insert"),
   xdmp:permission("engineering-manager", "read"),
   xdmp:permission("engineering-manager", "update"))
...

```

If you specify permissions explicitly to the function call, as shown above, those permissions override any default permission settings associated with the user (through user and role inheritance).

### 3.5.2 Default Permission Settings

If there is a set of permission requirements that meets the needs of most application scenarios, we recommend creating the appropriate default permission settings at the role or user level. This avoids having to explicitly create and set document permissions each time you call `xdmp:load()` or `xdmp:document-insert()`.

Default permission settings that apply to a user (either through a role or through the user definition) are also very important when loading documents via a WebDAV client (for example, dragging and dropping into a WebDAV folder). When you drag and drop files into a WebDAV folder, the permissions are automatically set based on the default permissions of the user in which the WebDAV client is logged in. For more information about WebDAV servers, see the *Administrator's Guide*.

The screenshot below displays a portion of the Role configuration screen for the *engineering* role. It shows insert and update capabilities being added to the *engineering* role's default permissions.

**default permissions** -- *The default set of permissions used in document creation.*

[Keep] Role Name (capability)

[add] engineering read

[add] engineering insert

[add] read

A user's set of default permissions is additive; it is the aggregate of the default permissions for all of the user's role(s) as well as for the user himself. Below is another screenshot of a User configuration screen for Ron. It shows read and update capabilities being added to the *engineering-manager* role as Ron's default permissions at the user level.

**default permissions** -- *The default set of permissions used in document creation.*

[Keep] Role Name (capability)

[add] engineering update

[add] engineering read

[add] read

**Note:** Ron is of the *engineering* role and does not have the *engineering-manager* role. A user does not need to have a certain role in order to specify that role in its default permission set.

You can also use a hybrid of the two methods described above. Assume that read and insert capabilities for the *engineering* role are specified as default permissions at the *engineering* role level as shown in the first screenshot. However, update and read capabilities are not specified for the *engineering-manager* at the user or *engineering* role level.

Further assume that the following code snippet is executed by Ron. It will achieve the desired objective of giving the *engineering-manager* role read and update capabilities on the document, and the *engineering* role read and insert capabilities.

```
...
xdmp:document-insert("/widget.com/engineering/features/2004-q1.xml",
  <new-features>
    <feature>
      <name>blue whistle</name>
      <assigned to>Ron</assigned to>
      ...
    </feature>
    ...
  </new-features>,
  (xdmp:default-permissions(),
   xdmp:permission("engineering-manager", "read")
   xdmp:permission("engineering-manager", "update")))
...
```

`xdmp:default-permissions()` returns Ron's default permissions (all at the role level in this example) of read and insert capabilities for the *engineering* role. Read and update capabilities for the *engineering-manager* role are then added explicitly as function parameters.

**Note:** `xdmp:document-insert()` turns into an update (rather than a create) function if a document with the specified document URI already exists. Consequently, if Ron calls the `xdmp:document-insert()` function the second time with the same document URI, the call will fail since Ron does not have update capability on the document.

Suppose that Ian, of the *engineering-manager* role, decides to give users of the sales role read permission on the document. (He wisely withholds update or insert capability or there will surely be an explosion of features!) The code snippet below shows how to add permissions to a document after it has been created.

```
...
xdmp:document-add-permissions(
  "/widget.com/engineering/features/2004-q1.xml",
  xdmp:permission("sales", "read"))
...
```

Update permission is needed to add permissions to a document. Therefore, the code snippet will only succeed if executed by Ian, or another user of the *engineering-manager* role. This prevents Ron from giving Emily, his buddy in *sales*, insert capability on the document.

**Note:** Changing default permissions for a role or a user does not affect the permissions associated with existing documents. To change permissions on existing documents, you need to use the permission update functions. See the MarkLogic Server built-in documentation for details.

## 4.0 Execute Privileges and Amps

Execute privileges provide authorization control for executing XQuery functions. There are built-in execute privileges which provide access to certain protected functions such as `xdrm:load`, and you can create your own execute privileges to protect your own application code. You can also temporarily allow a user to have more authority than they possess with an amp. This chapter describes the following:

- [Execute Privileges](#)
- [Amps](#)

### 4.1 Execute Privileges

If you want to protect the execution of an individual function in XQuery code, you can use *execute privileges*. Execute privileges require that a user must possess a specific privilege in order to run the protected code.

**Note:** Execute privileges operate at the function level; if you want to protect an entire XQuery document that is stored in a modules database, you can use execute permissions. For details on execute permissions, see “Permissions on Documents” on page 19.

This section describes aspects of execute privileges and includes the following parts:

- [Using Execute Privileges](#)
- [Execute Privileges and HTTP, WebDAV, and XDBC Servers](#)
- [Execute Privileges When Creating and Updating Collections](#)

#### 4.1.1 Using Execute Privileges

The following are the basic steps in using execute privileges:

- Create the privilege.
- Assign the privilege to a role.
- Write code to test for the privilege.

You create privileges and assign them to roles using the Admin interface. You use the built-in function `xdrm:security-assert` in your XQuery code to test for a privilege. Adding the security assert to XQuery code causes the system to test if the user running the code has the specified privilege. If the user possesses the privilege, then the code continues to execute. If the user does not possess the privilege, then the server throws an exception (which the application developer can catch and handle, if desired).

For example, if you want create an execute privilege to control the access to an XQuery function called `display-salary`, you can use the Admin interface to create an execute privilege called `allow-display-salary` and give the execute privilege any URI (for example, `http://my/privs/allow-display-salary`). Then you assign a role to the privilege. Finally, in your `display-salary` XQuery function, include an `xdmp:security-assert` call to test for the `allow-display-salary` execute privilege as follows:

```
define function display-salary (
  $employee-id as xs:unsignedLong)
as xs:decimal
{
  xdmp:security-assert("allow-display-salary", "execute"),
  ...
}
```

### 4.1.2 Execute Privileges and HTTP, WebDAV, and XDBC Servers

You can also use an execute privilege to control access to an HTTP, WebDAV, or XDBC server. In the Admin interface, there is a Privilege drop list on the server administration page. From there, you can specify a privilege required for server access. Any users that access the server must then possess the specified privilege. If a user tries to access an application on the server and does not possess the specified privilege, an exception is thrown. For an example of using this technique to control server access, see “Example: Using the Security Database in Different Servers” on page 35.

### 4.1.3 Execute Privileges When Creating and Updating Collections

If you create or update a document and add it to a collection, an execute permission is required. If the collection is a protected collection (that is, a collection that is created using the Admin Interface), then you either need permissions to update that collection or the `any-collection` execute privilege. If the collection is an unprotected collection, then you need the `unprotected-collections` execute privilege. For details on adding collections while creating a document, see the documentation for `xdmp:document-load`, `xdmp:document-insert`, and `xdmp:document-add-collections` in the *Mark Logic Built-In and Module Functions Reference*.

## 4.2 Amps

*Amps* are security objects that temporarily grant roles to unprivileged users. An amp grants the roles only for the execution of a given function. While executing an “amped” function, the user temporarily is part of the amped role, which in turn temporarily grant the user the additional privileges and permissions given by the roles configured in the amp.

Amps allow you to grant users additional privileges and permissions at a very granular level—executing a certain function. Assigning the user these additional roles permanently could compromise the security of the system. Amps enable you to limit the effect of the additional roles (privileges and permissions) to a specific function.

For example, a user may need a count of all the documents in the database in order to create a report. However, the user does not have read permissions on all the documents in the database. Therefore, queries run by the user will not “see” all the documents in the database. You can use an amp for the `document-count()` function to elevate the user to an admin role (or to a role that has read permissions for all documents) temporarily while the user is executing the function to count the documents in the system.

You use the Admin interface to create amps. Amps are specific to a single function, which you specify by URI and local name when creating the amp. You can only amp a function that resides in the Modules directory (`<install_dir>/Modules`) or in the Modules database for the server in which the function is executed; you cannot amp functions in any other XQuery module (for example, you cannot amp a function in a module installed under the filesystem root of an HTTP server). The reason that functions must reside in the Modules database or in the Modules directory is because these locations are trusted; allowing amped functions from under a server root or from functions submitted by a client can compromise security. For details on creating amps, see the “Security Administration” chapter of the *Administrator’s Guide*.

For an example that uses an amp, see “Access Control Based on Client IP Address” on page 43.

## 5.0 Users and Authentication

MarkLogic Server authenticates users when they access an application. This chapter describes users and the available authentication schemes, and includes the following sections:

- [Users](#)
- [Types of Authentication](#)

### 5.1 Users

A user in MarkLogic Server is used to authenticate requests to a server. Users are assigned to roles, which associate them with security attributes (privileges, permissions, and default permissions).

You configure users in the Admin interface, where you assign a user a name, a password, a set of roles, and a set of default permissions. To see a report of all of the security attributes associated with a given user, click on the `User:username` link in the Admin interface screen for the given user. For details on configuring users in the Admin interface, see the “Security Administration” chapter in the *Administrator’s Guide*.

### 5.2 Types of Authentication

For HTTP and WebDAV servers, you can control the type of authentication. XDBC servers always authenticate using digest-basic authentication. This section describes the following authentication schemes:

- [Basic](#)
- [Digest](#)
- [Digest-Basic](#)
- [Application Level](#)

#### 5.2.1 Basic

Basic authentication is the typical authentication scheme for web applications. When a user accesses an application page, she is prompted for a username and password. In basic mode, there is a light level of encryption on the password.

### 5.2.2 Digest

Digest authentication works the same way as basic, but offers considerably better encryption of passwords sent over the network. When a user accesses an application page, she is prompted for a username and password.

**Note:** If you change an HTTP or WebDAV server from basic to digest authentication, it will invalidate all passwords in the security database. You must then reenter the passwords in the Admin interface. Alternatively, you can migrate to digest-basic mode initially, then switch to digest-only mode once all users have accessed the server at least once. The first time the user accesses the server after changing from basic to digest-basic scheme, the server will compute the digest password by extracting the relevant information from the credentials supplied in basic mode.

### 5.2.3 Digest-Basic

The digest-basic authentication scheme uses the more secure digest whenever possible, but reverts to basic authentication when needed. Some older browsers, for example, do not support digest authentication. Digest-basic scheme is also useful if you had previously used basic authentication, but want to migrate to using digest. The first time the user accesses the server after changing from basic to digest-basic authentication scheme, the server will compute the digest password by extracting the relevant information from the credentials supplied in basic mode.

### 5.2.4 Limitations of Digest and Basic Authentication

Since the browser does not provide a way to clear a user's authentication information in basic or digest mode, the user remains logged in until the browser is shut down. In addition, there is no way to create a custom login page using these schemes. For certain deployments, application-level authentication may be more appropriate.

### 5.2.5 Application Level

Application-level authentication bypasses all authentication and automatically logs all users in as a specified *default user*. You specify the default user in the Admin interface, and any users accessing the server automatically inherit the security attributes (roles, privileges, default permissions) of the default user. Application-level authentication is available on HTTP and WebDAV servers.

The default user should have required privileges to at least read the initial page of the application. In many application scenarios, the user is then given the opportunity to explicitly log in to the rest of the application from that page. How much of the application and what data a user can access before explicitly logging in depends on the application and the roles that the default user is part of. For an example of this type of configuration, see “Custom Login Pages” on page 42.

## 6.0 Administering Security

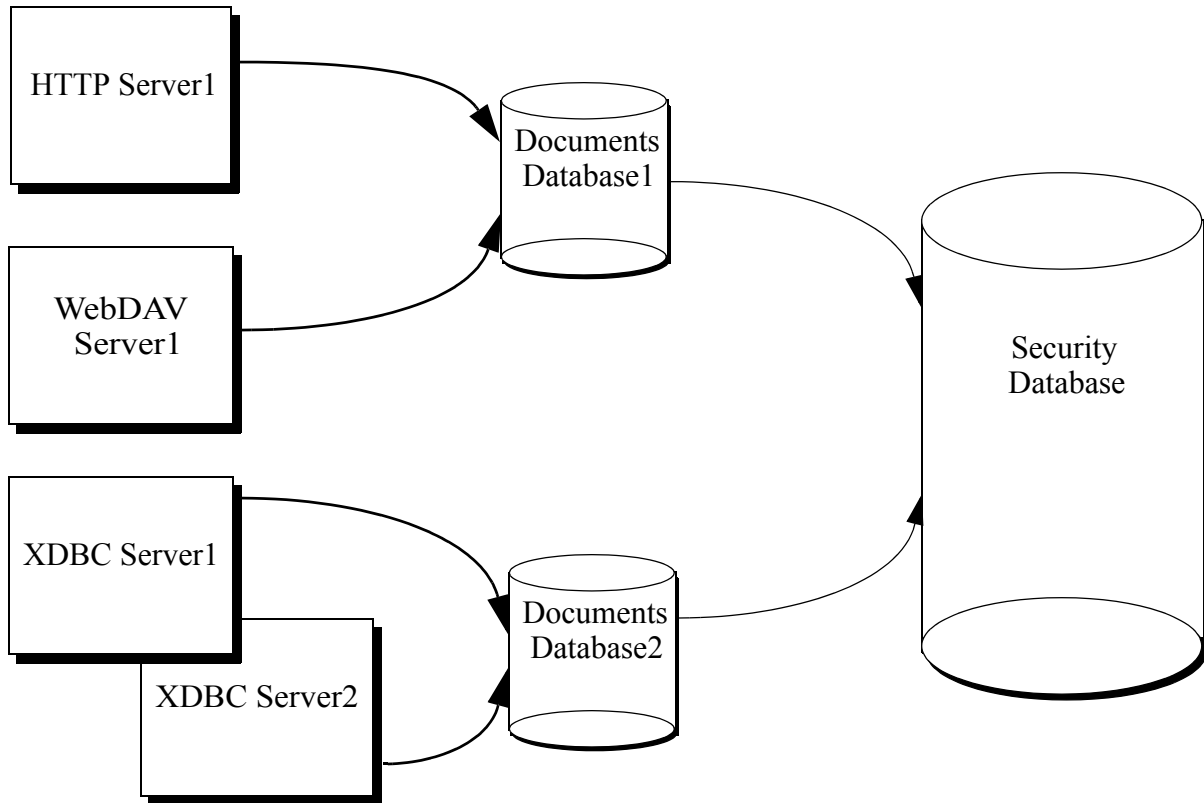
This chapter describes the basic steps needed to administer security in MarkLogic Server. It does not provide the detailed procedures for creating users, roles, privileges, and so on. For those procedures, see the “Security Administration” chapter of the *Administrator’s Guide*. This chapter includes the following sections:

- [Overview of the Security Database](#)
- [Associating a Security Database With a Documents Database](#)
- [Managing and Using Objects in the Security Database](#)
- [Backing Up the Security Database](#)
- [Example: Using the Security Database in Different Servers](#)

### 6.1 Overview of the Security Database

Authentication in MarkLogic Server occurs via the *security database*. The security database contains security objects such as privileges, roles, and users. A security database is associated with each HTTP, WebDAV, or XDBC server; typically, a single security database services all of the servers configured in a system. Actions against the server are authorized based on the security database. The security database works the same way for Enterprise Edition clustered systems as it does for Standard Edition single-node systems; there is always a single security database associated with each HTTP, WebDAV, or XDBC server.

The configuration which associates the security database with the database and servers is at the database level. HTTP, WebDAV, and XDBC servers each access a single documents database, and each database in turn accesses a single security database. Multiple documents databases can access the same security database. The following figure shows many servers accessing some shared and some different documents databases, but all accessing the same security database.

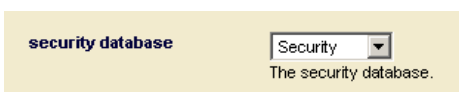


Sharing the security database across multiple servers provides a common security configuration. You can set up different privileges for different databases if that makes sense, but they are all stored in a common security database. For an example of this type of configuration, see “Example: Using the Security Database in Different Servers” on page 35.

Besides storing users, roles, and privileges that you create, the security database also stores many pre-defined privileges and several pre-defined roles. These objects help to control access to privileged activities in MarkLogic Server. Examples of privileged activities include loading data, accessing URIs, and so on. The security database is initialized during the installation process. For a list of all of the pre-defined privileges and roles, see the corresponding appendixes in the *Administrator's Guide*.

## 6.2 Associating a Security Database With a Documents Database

When you configure a database, you must specify which database is its security database. You can associate the security database to another database in the database configuration screen of the Admin interface. This configuration specifies which database the server will use to authenticate users and authorize requests. By default, the security database is named *Security*. The following screen shot shows the server configuration screen drop-list which specifies the security database.



## 6.3 Managing and Using Objects in the Security Database

There are two mechanisms you can use to add, change, delete, and use objects in the security database: the Admin interface and the `security.xqy` XQuery functions. This section describes the types of things you can do with each of these mechanisms, and includes the following subsections:

- [Using the Admin Interface](#)
- [Using the security.xqy Module Functions](#)

### 6.3.1 Using the Admin Interface

The Admin interface is an application installed with MarkLogic Server for administering databases, servers, clusters, and security objects. The Admin interface is essentially an application designed to manage the objects in the security database (although it manages other things, such as configuration information, too). You use the Admin interface to create, change, or delete objects in the security database. Activities such as creating users, creating roles, assigning privileges to roles, and so on are all done in the Admin interface. By default, the Admin interface application runs on port 8001.

For the procedures for creating, deleting, and modifying security objects, see the *Administrator's Guide*.

### 6.3.2 Using the security.xqy Module Functions

The installation process installs an XQuery library to help you use security objects in your XQuery code. The `security.xqy` module file includes many functions which help you to access user and privilege information, as well as functions to create, modify, and delete objects in the security database.

The functions in `security.xqy` must be executed against the security database. You can use these functions to do a wide variety of things. For example, you can write code to test which collections a user has access to, and use that information in some logic in your code. There are many functions and a very wide variety of things you can do with the `security.xqy` functions.

For the signatures and descriptions of the functions in `security.xqy`, see the [“MarkLogic Server Built-In and Module Function Reference.”](#)

## 6.4 Backing Up the Security Database

The security database is the central entry point to all of your MarkLogic Server applications. If the security database becomes unavailable, no users will be able to access any applications. Therefore, it is very important to create a backup of the security database. Use the database backup utility in the Admin interface to back up the security database. For details, see the “Backing Up and Restoring a Database” chapter of the *Administrator's Guide*.

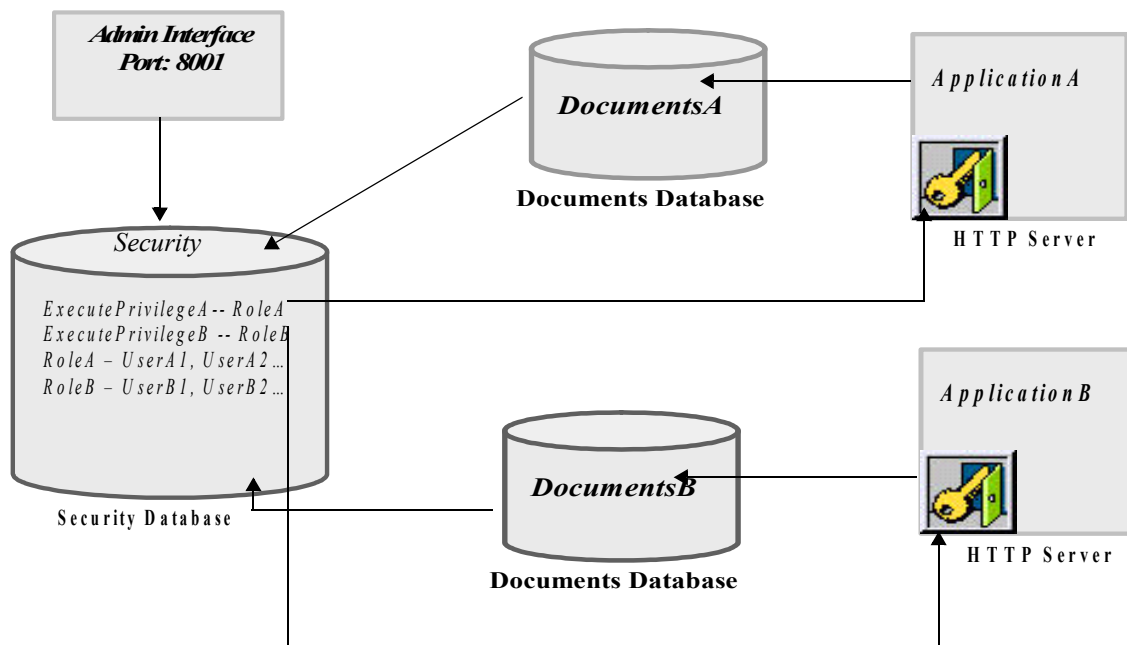
### 6.5 Example: Using the Security Database in Different Servers

The security database typically is used for the entire system, including all of the HTTP, WebDAV, and XDBC servers configured. You can create distinct privileges to control access to each server. If each server accesses a different document database, these privileges can effectively control access to each database (because the database is associated with the server). Users then must have the appropriate *login privileges* to login to the a server, and they therefore have no way of accessing either the applications or the content stored in the database accessed through that server without possessing the appropriate privilege. This example describes such a scenario.

Consider an example with two databases—*DocumentsA* and *DocumentsB*. *DocumentsA* and *DocumentsB* share a single security database, *Security*. *Security* is the default security database managed by the Admin interface on port 8001. There are two HTTP servers, *ApplicationA* and *ApplicationB*, connected to *DocumentsA* and *DocumentsB* respectively.

*ExecutePrivilegeA* controls login access to *ApplicationA*, and *ExecutePrivilegeB* to *ApplicationB*. *RoleA* is granted *ExecutePrivilegeA* and *RoleB* is granted *ExecutePrivilegeB*.

With this configuration, users who are assigned *RoleA* can access documents in *DocumentsA* and users of *RoleB* can access documents in *DocumentsB*. Assuming that *ExecutePrivilegeA* or *ExecutePrivilegeB* are appropriately configured as login privileges on every HTTP and XDBC server that accesses either *DocumentsA* or *DocumentsB*, user access to these databases can conveniently be managed by assigning users the role(s) *RoleA* and/or *RoleB* as required.



**Note:** The Admin interface at port 8001 is also used to configure all databases, HTTP servers, hosts, and so on. The connection between the Admin interface and *Security* in the diagram simply indicates that the Admin interface is storing all security objects—users, roles, and privileges—in *Security*.

The steps below outline the process to create the configuration in the above example.

1. Create two document databases: *DocumentsA* and *DocumentsB*. Leave the security database for the document databases as *Security* (the default setting).
2. Create two execute privileges: *ExecutePrivilegeA* and *ExecutePrivilegeB*. They represent the privilege to access *ApplicationA* and *ApplicationB* respectively. *ApplicationA* and *ApplicationB* are two HTTP servers that will be created later in the process.

**Note:** The new execute privileges created using the Admin interface are stored in *Security*. The new roles and users created below will also be stored in *Security*.

3. Create two new roles. These roles will be used to organize users into groups and to facilitate granting access to users as a group.
  - a. Create a new role. Name it *RoleA*.
  - b. Scroll down to the Execute Privileges section and select *ExecutePrivilegeA*. This associates *ExecutePrivilegeA* with *RoleA*. Any user assigned *RoleA* will be granted *ExecutePrivilegeA*.
  - c. Repeat the steps for *RoleB*, selecting *ExecutePrivilegeB* instead.
4. Create two new HTTP servers:
  - a. Create a new HTTP server. Name it *ApplicationA*.
  - b. Select *DocumentsA* as the database. *ApplicationA* is now attached to *DocumentsA* which in turn uses *Security* as its security database.
  - c. Select basic, digest or digest-basic authentication scheme.
  - d. Select *ExecutePrivilegeA* in the privilege drop down menu. This indicates that *ExecutePrivilegeA* is required to access *ApplicationA*.
  - e. Repeat the steps for *ApplicationB*, selecting *ExecutePrivilegeB* instead.
5. Create new users.
  - a. Create a new user named *UserA1*.

- b. Scroll down to the Roles section and select *RoleA*.
- c. Repeat the steps for *UserB1*, selecting *RoleB* in the roles section.

*UserA1* will be granted *ExecutePrivilegeA* by virtue of its role (*RoleA*) and will have login access to *ApplicationA*. Because *ApplicationA* is connected to *DocumentsA*, *UserA1* will be able to access documents in *DocumentsA* assuming no additional security requirements are implemented in *ApplicationA*, or added to documents in *DocumentsA*. The corresponding is true for *UserB1*.

The configuration process is now complete. Additional users can be created by simply repeating step 5 and selecting the appropriate role. All users assigned *RoleA* will have login access to *ApplicationA* and all users assigned *RoleB* will have login access to *ApplicationB*.

This approach can also be easily extended to handle additional discrete databases and user groups by creating additional document databases, roles and execute privileges as necessary.

## 7.0 Sample Scenarios For Defining Security Policies

This chapter describes some common scenarios for defining security policies in your applications. The scenarios shown here are by no means exhaustive; there are many possibilities for how to set up security in your applications. The following sections are included:

- [General Steps for Creating and Implementing Security Policies](#)
- [Restricting Document Access to a Particular Program](#)
- [Choosing the Access Control Option for an Application](#)

### 7.1 General Steps for Creating and Implementing Security Policies

This section defines the general steps needed when using security in an application. Because of the flexibility of the MarkLogic Server security model, there are many different ways to implement similar security policies. These steps are simple guidelines; the actual steps you take will depend on the security policies you need to implement.

1. Determine your security policies. Try to have answers for the following types of questions:
  - What documents do you want to protect?
  - What code do you want to control the execution of?
  - Are there any natural categories you can define based on business function (for example, marketing, sales, engineering)?
  - What is the level of risk posed by your users? Will the applications be used only by trusted, internal people or will they be open to a wider audience?
  - How sensitive is the data you are protecting?
2. Based on your security policies, plan out roles and privileges.
  - a. Determine the level of granularity with which you need to protect objects in the database.
  - b. Determine how you want to group privileges together in roles.
3. Create needed URI and execute privileges.
4. Create roles.
5. Create users.
6. Assign users to roles.

7. Set default permissions for users, either indirectly through roles or directly through the users.
8. Protect code with `xmmp:security-assert` functions, where needed.
9. Load data with the appropriate permissions. If needed, change the permissions of existing data using the `xmmp:document-add-permissions`, `xmmp:document-set-permissions`, and `xmmp:document-remove-permissions` functions.
10. Assign access privileges to HTTP, WebDAV, and XDBC servers as needed.

## 7.2 Restricting Document Access to a Particular Program

If you load XQuery code into a modules database, you can use `execute` permissions to control who can run a particular XQuery module document. Then, a user must possess `execute` permissions to run the module. To set up a module to do this, perform the following steps:

1. Specify a modules database in the configuration for the application server (HTTP or WebDAV) in which you want to control the execution of an XQuery module.
2. Load the XQuery module into the modules database, using a URI with a `.xqy` extension.
3. Set `execute` permissions on the `.xqy` document for a given role. For example, if you want users with the `run_application` role to be able to execute an XQuery module with a URI `http://modules/my_module.xqy`, run a query similar to the following:

```
xmmp:document-set-permissions ("http://modules/my_module.xqy",  
                               xmmp:permission("run_application", "execute") )
```

Now only users with the `run_application` role can execute this document, effectively disallowing everyone who is not part of the role from running the application.

**Note:** Because the application code might also contain `amped` functions, this technique can help restrict access to applications that use `amps` to run functions at higher levels of authority than those possessed by the user who runs them.

## 7.3 Choosing the Access Control Option for an Application

The role-based security model in MarkLogic Server combined with the supported authentication schemes give developers numerous application access control options. This section describes the following common application access control alternatives:

- [No Access Restriction](#)
- [Uniform Access for All Users in the Security Database](#)
- [Limiting Application Access to a Subset of Users](#)
- [Custom Login Pages](#)
- [Access Control Based on Client IP Address](#)

For details on the different authentication schemes, see “Types of Authentication” on page 30.

### 7.3.1 No Access Restriction

This approach is appropriate if security is not a concern for your MarkLogic Server implementation or if you are just getting started and want to explore the capabilities of MarkLogic Server before contemplating your security architecture.

You can turn off access control for each HTTP or WebDAV server individually by following these steps using the Admin interface:

1. Go to the Configure tab for the HTTP server for which you want to turn off access control.
2. Scroll down to the authentication field and choose application-level for the authentication scheme.
3. Choose a user with the *admin* role for the default user. For example, you may choose the admin user you created when installing the software.

**Note:** To assist with identifying users with the *admin* role, the default user selection field places (*admin*) next to admin users.

In this scenario, all users accessing the application server are automatically logged in with a user that has the *admin* role. By default, the admin role has the privileges and permissions to perform any action and access any document in the server. Therefore, security is essentially turned off for the application. All users have full access to the application and database associated with the application server.

### 7.3.2 Uniform Access for All Users in the Security Database

This approach allows you to restrict application access to users in your security database, and gives those users full access to all application servers defined in MarkLogic Server. There are multiple ways to achieve the same objective but this is the simplest way. Using the Admin interface:

1. Go to the Users tab under security.
2. Give all users in security database the *admin* role.
3. Go to the Configuration tabs for all HTTP and WebDAV servers in the system.
4. Go to the authentication field and choose digest, basic or digest-basic authentication.
5. Leave the privilege field blank since it will have no effect in this scenario. This field specifies the privilege that is needed to log into application server. However, the users are assigned the *admin* role and are treated as having all privileges.

In this scenario, all users must authenticate with a username and password. Once they are authenticated, however, they have full access to all functions and data in the server.

### 7.3.3 Limiting Application Access to a Subset of Users

This application access control method can be modified or extended to meet the requirements in many application scenarios. At the same time, it uses more of the available security features and therefore requires a better understanding of the security model.

To limit application access to a subset of the users in the security database, perform the following steps using the Admin interface:

1. Create an execute privilege to represent the privilege to access the application server. We will refer to this execute privilege as *exe-priv-appl*.
2. Create a role named *role-appl* that has *exe-priv-appl* execute privilege.
3. Add *role-appl* to the roles of all users in the security database who should have access to the application server.
4. In the Configuration page for the application server, scroll down to the authentication field and select digest, basic or digest-basic. If you want to use application-level authentication to achieve the same objective, a custom login page is required. See the next section for details.
5. Select *exe-priv-appl* for the privilege field. Once this is done, only the users who have the *exe-priv-appl* by virtue of their role(s) will be able to access this application server.

**Note:** If you want any user in the security database to be able to access the application, leave the privilege field blank.

At this point, the application access control is configured. However, this method of authentication needs to be accompanied by the appropriate security configuration. For example, `xdmp:document-insert()` and `xdmp:load()` will throw exceptions unless the user possesses the appropriate privileges. Also, users must have the appropriate default permissions when creating new documents in a database, otherwise the documents are created but nobody (except users with the `admin` role) will be able to access them, including the user who created them.

### 7.3.4 Custom Login Pages

Digest and basic authentication use the browser's username and password prompt to obtain user credentials. The server then authenticates the credentials against the security database. There is no good way to create a custom login page using digest and basic authentication. To create custom login pages, you need to use application-level authentication.

To configure MarkLogic Server to use a custom login page for an application server, perform the following steps:

1. Go to the Configuration tab for the HTTP or XDBC server for which you want to create a custom login page.
2. Scroll down to the authentication field and select application-level.
3. Choose *nobody* as the default user. The *nobody* user is automatically created when MarkLogic Server is installed. It does not have an associated role and therefore has no privileges. *Nobody* can only access pages and perform functions for which no privileges are required.
4. Create a custom login page that meets your needs. We will refer to this page as `login.xqy`.
5. Make `login.xqy` the default page displayed by the application server. Do not require any privilege to access `login.xqy` (that is, do not place `xmmp:security-assert()` in the beginning of the code for `login.xqy`. This makes `login.xqy` accessible by *nobody*, the default user specified above, until the actual user logs in with his credentials.

The `login.xqy` page will likely contain a snippet of the code as shown below:

```
...return
if xdmp:login($username, $password) then
  ... protected page goes here...
else
  ... redirect to login page or display error page...
```

The rest of this example assumes that all valid users can access all the pages and functions within the application.

6. Create a role called *application-user-role*.

7. Create an execute privilege called *application-privilege*. Add this privilege to the *application-user-role*.
8. Add the *application-user-role* to all users who are allowed to access the application.
9. Add this snippet of code before the code that displays each of the pages in the application, except for `login.xqy`:

```
try
{
  xdmp:security-assert("application-privilege","execute")
}
catch($e)
{
  xdmp:redirect-response("login.xqy")
}
```

or

```
if(not(xdmp:has-privilege("application-privilege","execute")))
then
(
  xdmp:redirect-response("login.xqy")
)
else ()
```

This ensures that only a user who has the *application-privilege* by virtue of his role can access these protected pages.

Similar to the previous approach, this method of authentication requires the appropriate security configuration. See “Introduction to Security” on page 6 for background on the security model.

### 7.3.5 Access Control Based on Client IP Address

MarkLogic Server supports deployments in which a user is automatically given access to the application based on the client IP address.

Consider a scenario in which a user is automatically logged in if he is accessing the application locally (as *local-user*) or from an approved subnet (as *site-user*). Otherwise, the user is asked to login explicitly. The steps below describe how to configure MarkLogic Server to achieve the desired access control.

1. The first step is to configure the application server to use a custom login page:
  - a. Go to the Configuration tab for the HTTP or WebDAV server for which you want to create a custom login page.

- b. Scroll down to the authentication field and select application-level.
- c. For this example, choose *nobody* as the default user. The user *nobody* is automatically created when MarkLogic Server is installed. It does not have an associated role and hence has no privileges. *Nobody* can only access pages and perform functions for which no privileges are required.
- d. Add the following code snippet to the beginning of the default page displayed by the application, e.g. *default.xqy*.

```
declare namespace widget = "http://widget.com"
import module "http://widget.com" at "/login-routine.xqy"

let $login := widget:try-ip-login()
return
if($login) then
  <html>
    <body>
      The protected page goes here.
      You are {xdmp:get-current-user()}
    </body>
  </html>
else
  xdmp:redirect-response("login.xqy")
```

The `try-ip-login()` function is defined in *login-routine.xqy*. It is used to determine if the user can be automatically logged in based on the client IP address. If the user cannot be logged in automatically, he will be redirected to a login page called `login.xqy` where he has to log in explicitly. See “Custom Login Pages” on page 42 for example code for `login.xqy`.

2. Next, we will define `try-ip-login()`:
  - a. Create a file named *login-routine.xqy* and place the file in the *Modules* directory within the MarkLogic Server program directory. You will create an amp for `try-ip-login()` in *login-routine.xqy* in the next code sample. For security reasons, all “amped” functions must be located in the specified *Modules* directory or in the Modules database for the server.

- b. Add the following code to *login-routine.xqy*:

```
module "http://widget.com"
declare namespace widget ="http://widget.com"

define function try-ip-login() as xs:boolean
{
  let $ip := xdmp:get-ip()
  return
  if(compare($ip,"127.0.0.1") eq 0) then (:local host:)
    xdmp:login("localuser",())
  else if(starts-with($ip,<approved-subnet>)) then
    xdmp:login("site-user",())
  else
    false()
}
```

If the user is accessing the application from an approved IP address, `try-ip-login()` logs in the user with username *local-user* or *site-user* as appropriate and returns `true()`. Otherwise, `try-ip-login()` returns `false()`.

**Note:** In the code snippet above, the empty sequence `()` is supplied in place of the actual passwords for *local-user* and *site-user*. The `xdmp-login` pre-defined execute privilege grants the right to call `xdmp:login()` without the actual password. This makes it possible to create deployments in which users can be automatically logged in without storing user passwords outside the system.

3. Finally, to ensure that the code snippet above is called with the requisite `xdmp-login` privilege, we will configure an amp for `try-ip-login()`:
  - a. Create a role in the Admin interface called *login-role*.
  - b. Assign the `xdmp-login` pre-defined execute privilege to *login-role*. The `xdmp-login` privilege gives a user of the *login-role* the right to call `xdmp:login()` for any user without supplying the password.
  - c. Create an amp for `try-ip-login()` as shown below:

**New Amp** ok cancel

**amp** -- A role amplification.

**local name**   
A function local-name.  
**Required. You must supply a value for local-name.**

**namespace**   
A namespace.  
**Required. You must supply a value for namespace.**

**document uri**   
A document's URI.  
**Required. You must supply a value for document-uri.**

**database**   
A database the module is found in.

**roles** -- The roles assigned.

admin

admin-builtins

domain-management

filesystem-access

login-role

An amp temporarily assigns additional role(s) to a user only for the execution of the specified function. The amp above gives any user who is executing `try-ip-login()` the *login-role* temporarily for the execution of the function.

In this example, *default.xqy* is executed as *nobody*, the default user for the application. When `try-ip-login()` is called, *nobody* is temporarily amped to the *login-role*. *Nobody* is temporarily assigned the `xdmp:login` execute privilege by virtue of the *login-role*. This enables *nobody* to call `xdmp:login()` in `try-ip-login()` for any user without the corresponding password. Once the login process is completed, the user will access the application with the permissions and privileges of *local-user* or *site-user* as appropriate.

4. The remainder of the example assumes that *local-user* and *site-user* can access all the pages and functions within the application.
  - a. Create a role called *application-user-role*.
  - b. Create an execute privilege called *application-privilege*. Add this privilege to the *application-user-role*.
  - c. Add the *application-user-role* to *local-user* and *site-user*.

- d. Add this snippet of code before the code that displays each of the subsequent pages in the application:

```
try
{
  xdmp:security-assert("application-privilege", "execute")
  ...
}
catch($e)
{
  xdmp:redirect-response("login.xqy")
}
```

or

```
if(not(xdmp:has-privilege("application-privilege", "execute")))
then
(
  xdmp:redirect-response("login.xqy")
)
else ()
```

This ensures that only the user who has the *application-privilege* by virtue of his role can access these protected pages.

## Technical Support

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

For registered customers, we invite you to visit our support website at <http://support.marklogic.com> to access our full suite of documentation and help materials. For all customers, including community licensed customers, visit the Mark Logic Developer’s site at <http://developer.marklogic.com>, which includes full product documentation, downloads, and developer community open-source projects.

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

[support@marklogic.com](mailto:support@marklogic.com)

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