Securing MarkLogic Server

MarkLogic 11

Publication date 2024-04-25

Table of Contents

1. Introduction to Security

When you create systems that store and retrieve data, it is important to protect the data from unauthorized use, disclosure, modification or destruction. Ensuring that users have the proper authority to see the data, load new data, or update existing data is an important aspect of application development. Do all users need the same level of access to the data and to the functions provided by your applications? Are there subsets of users that need access to privileged functions? Are some documents restricted to certain classes of users? The answers to questions like these help provide the basis for the security requirements for your application.

MarkLogic Server includes a powerful and flexible role-based security model to protect your data according to your application security requirements. There is always a trade-off between security and usability. When a system has no security, then it is open to malicious or unmalicious unauthorized access. When a system is too tightly secured, it might become difficult to use successfully. Before implementing your application security model, it is important to understand the core concepts and features in the MarkLogic Server security model. This section introduces the MarkLogic Server security model.

1.1. Licensing

Some MarkLogic Server security features require an Advanced Security License in addition to the regular license. The Advanced Security License option is required when using:

- Compartment Security
- Redaction
- An external Key Management System (KMS) or keystore with encryption at rest
- Query-Based Access Control

For more about redaction, see Redacting Document Content in the *Application Developer's Guide*. See Query-Based Access Control in this guide for more about query-based access control.

1.2. Security Overview

This section provides an overview of the three main principles used in MarkLogic Server security.

1.2.1. Authentication and Access Control

Authentication is the process of verifying user credentials for a named user. Authentication makes sure you are who you say you are. Users are typically authenticated with a username and password. Authentication verifies user credentials and associates an application session with the authenticated user. Every request to MarkLogic Server is issued from an authenticated user. Authentication, by itself, does not grant access or authority to perform specific actions. There are several ways to set up server authentication in MarkLogic Server.

Authentication by username and password is only part of the story. You might grant access to users based on something other than identity, something such as the originating IP address for the requests. Restricting access based on something other than the identity of the user is generally referred to as *access control*.

For details on authentication, see Authenticating Users.

1.2.2. Authorization

Authorization provides the mechanism to control document access, XQuery and JavaScript code execution, and document creation. For an authenticated user, authorization determines what you are allowed to do. For example, authorization is what allows the user named *Melanie* to read and update a document, allows the user named *Roger* to only read the document, and prevents the user named

Hal from knowing the document exists at all. In MarkLogic Server, authorization is used to protect documents stored in a database and to protect the execution of XQuery or JavaScript code. For details on authorization in MarkLogic Server, see Protecting Documents and Protecting XQuery and JavaScript Functions with Privileges.

1.2.3. Administration

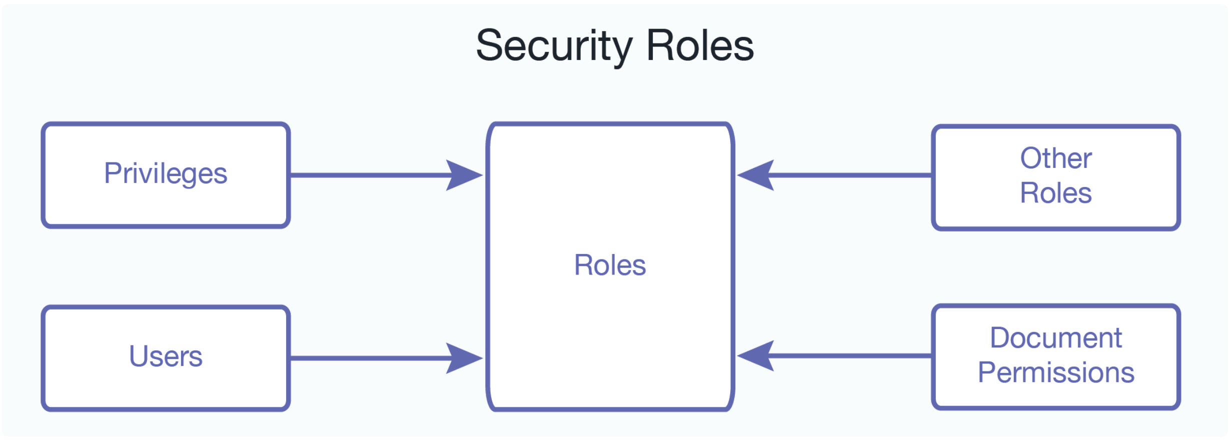
Administration is the process of defining, configuring, and managing the security objects, such as users, roles, privileges, and permissions that implement your security policies. For details on security administration procedures in MarkLogic Server, see Security Administration and *Administrating MarkLogic Server*.

1.3. MarkLogic Server Security Model

The MarkLogic Server security model is flexible and enables you to set up application security with the level of granularity needed by your security requirements.

1.3.1. 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 relate directly to roles. The following conceptual diagram shows how each of these entities points into one or more roles:



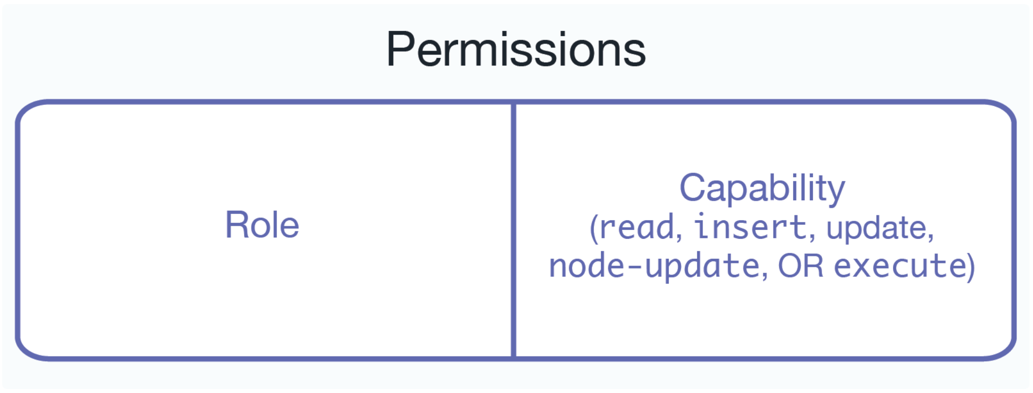
There are two types of privileges: URI privileges and execute privileges. URI privileges are used to control the creation of documents with certain URIs. Execute privileges are used to protect the execution of functions in XQuery or JavaScript code.

NOTE

For execute privileges' type, you may achieve finer granularity access control over configuration and various administration abilities through defining *granular privileges*. For information on granular privileges, see Compartment Security.

Privileges are assigned to zero or more roles, roles are assigned to zero or more other roles, and users are assigned to zero or more roles. 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. Permissions are assigned to documents, either at load time or as a separate administrative action. Each permission is a combination of a role and a capability (`read`, `insert`, `update`, `node-update`, `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.

Capabilities represent actions that can be performed. There are five capabilities in MarkLogic Server:

- `read`
- `insert`
- `update`
- `node-update`
- `execute`

Users inherit the sum of the privileges and permissions from their roles.

For more details on how roles work in MarkLogic Server, see Role-Based Security Model. For more details on privileges and permissions, see Protecting Documents.

1.3.2. Element Level Security

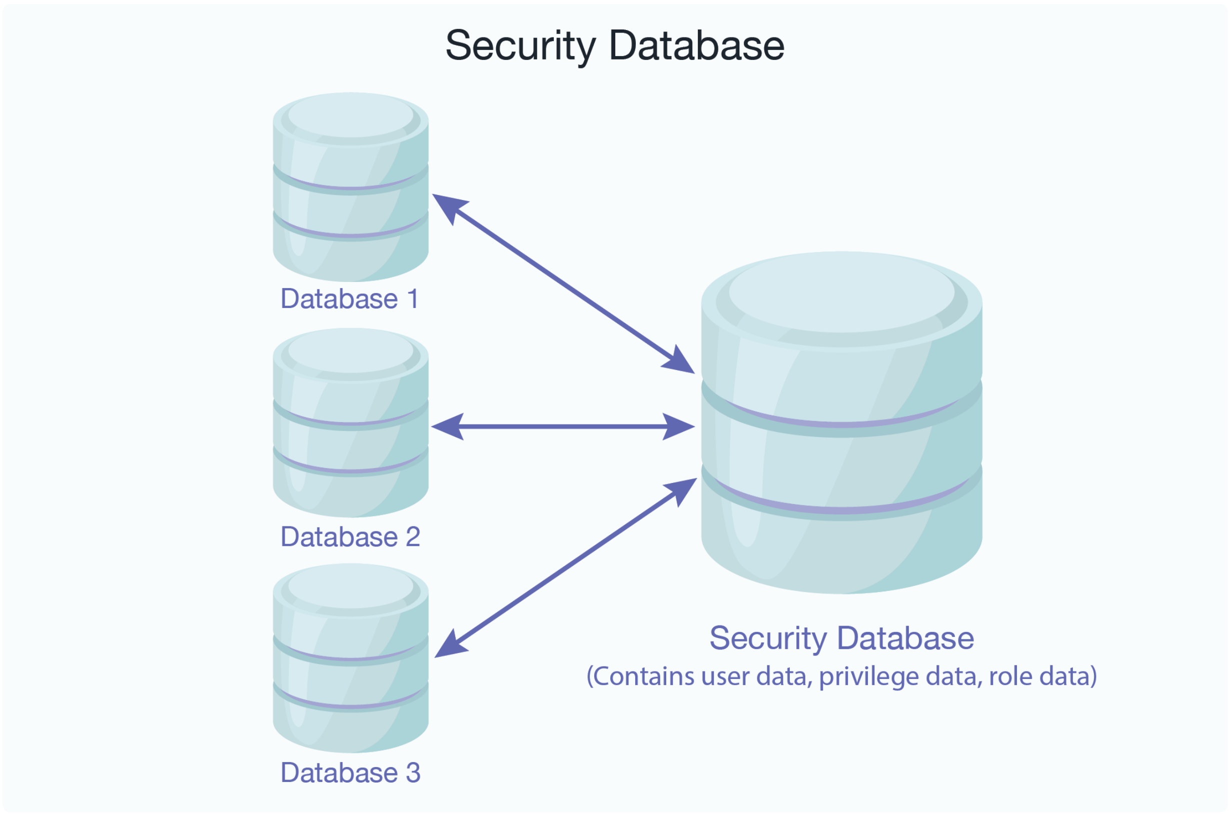
Element level security uses protected paths to conceal certain elements in document from specific users, while leaving other parts of a document available to search and view. You can use element level security to control access to specific JSON properties or XML elements within documents. This means that specific information inside a document may be hidden from a particular user based on the user's role, while still providing access to other information in the document.

Element level security can be used in addition to and along with existing document level security and compartment security. For more information about element level security, see Element Level Security.

1.3.3. Access Control with the Security Database

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

The following figure shows that many databases can be configured to use 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 Authenticating Users and Administering Security.

There may be circumstances in which a cluster is configured with more than one `Security` database, such as when using database replication. When multiple `Security` databases are used, there should be an equal number of Admin servers with different ports, one for each `Security` database. Each `Security` database can then be upgraded by its respective Admin Interface.

The name of the Security database used by the Admin Interface is shown in the upper right corner of the Security Configuration page.

1.3.4. Security Administration

MarkLogic Server administrators are privileged users who have the authority to perform tasks such as creating, deleting, and modifying users, roles, and privileges. 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 are assigned the `admin` role can perform any task on the system, including deleting data.

MarkLogic Server provides the following ways to administer security:

- Admin Interface
- REST Management API
- XQuery and JavaScript server-side security administration functions

For details on administering security, see Administering Security.

1.4. Terminology

This section defines the terms used throughout the security documentation.

1.4.1. User

A *user* is a named entity used to authenticate a request to an HTTP, WebDAV, ODBC, or XDBC server. For details on users, see Authenticating Users.

1.4.2. Role

A *role* is a named entity that provides authorization privileges and permissions to other roles or to users. You can assign roles to other roles (which can in turn include assignments to other roles, and so on). Roles are the fundamental building blocks that you use to implement your security policies. For details on roles, see Role-Based Security Model.

1.4.3. 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 Protecting XQuery and JavaScript Functions with Privileges.

1.4.4. 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 assigned to roles that have the URI privilege can create documents with URIs starting with the base string. For details on URI privileges, see Protecting Documents.

1.4.5. Permission

A *permission* provides a role with the capability to perform certain actions (`read`, `insert`, `update`, `node-update, execute`) on a document or a collection. Permissions consist of a role and a capability. Permissions are assigned to documents and collections. For details on permissions, see Protecting Documents.

1.4.6. Amp

An *amp* provides a user with the additional authorization to execute a specific function by temporarily giving the user additional roles. For details on amps, see Temporarily Increasing Privileges with Amps.

2. Role-Based Security Model

MarkLogic Server uses a role-based security model. Each security entity is associated with a role. This section describes the role-based security model.

2.1. Understanding Roles

As described in Role-Based Security Model (Authorization), roles are the central point of authorization in MarkLogic Server. This section describes how the other security entities relate to roles.

2.1.1. Assigning Privileges to Roles

Execute privileges control access to XQuery or JavaScript code. URI privileges control access to creating documents in a given URI range. You associate roles with privileges by assigning the privileges to the roles.

Execute Privileges

Execute privileges allow developers to control authorization for the execution of an XQuery or JavaScript function. If an XQuery or JavaScript function is protected by an execute privilege, the function must include logic to check if the user executing the code has the necessary execute privilege. That privilege is assigned to a user through a role that includes the specific execute privilege. There are many execute privileges pre-defined in the security database to control execution of built-in XQuery and JavaScript functions.

For more details on execute privileges, see Protecting XQuery and JavaScript Functions with Privileges.

URI Privileges

URI privileges control authorization for creation of a document with a given URI prefix. To create a document with a prefix that has a URI privilege associated with it, a user must be part of a role that has the needed URI privilege.

For more details on how URI privileges interact with document creation, see Protecting Documents.

2.1.2. Associating Permissions with Roles

Permissions are security characteristics of documents that associate a role with a capability. The capabilities are the following:

- `read`
- `insert`
- `update`
- `node-update`
- `execute`

Users gain the authority to perform these capabilities on a document if they are assigned a role to which a permission is associated.

For more details on how permissions interact with documents, see Document Permissions.

2.1.3. Default Permissions in Roles

Roles are one of the places where you can specify *default permissions*. If permissions are not explicitly specified when a document is created, the default permissions of the user creating the document are applied. The system determines the default permissions for a user based on the user’s roles. The total set of default permissions is derived from the user’s roles and all inherited roles.

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

2.1.4. Assigning Roles to Users

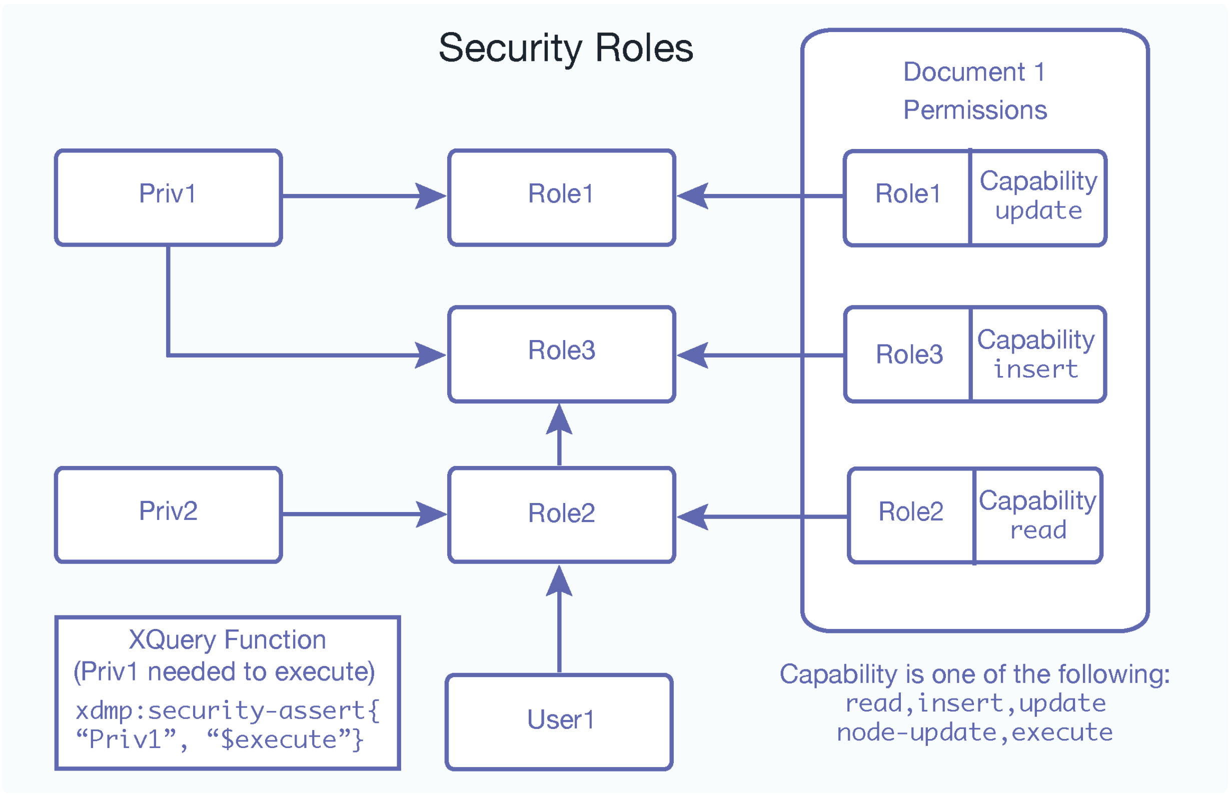
Users are authenticated against the security database configured for the database being accessed. Roles are the mechanism by which authorization information is derived. You assign roles to a user. The roles provide the user with a set of privileges and permissions that grant the authority to perform actions against code and documents. At any given time, a user *possesses* a set of privileges and default permissions that is the sum of the privileges and default permissions inherited from all of the roles currently assigned to that user.

Use the Admin Interface to display the set of privileges and default permissions for a given user; do not try and calculate it yourself as it can easily get fairly complex when a system has many roles. To display a user's security settings, use Admin Interface > Security > User > Describe. You need to select a specific user to see the Describe tab.

For more details on users, see Authenticating Users.

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, `User1` is part of `Role2`, and `Role2` inherits `Role3`. Therefore, even though `User1` has only been assigned `Role2`, `User1` possesses all of the privileges and permissions from both `Role2` and `Role3`. Following the arrows pointing into `Role2` and `Role3`, you can see that the user possesses `Priv1` and `Priv2` based on the privileges assigned to these roles and `insert` and `read` capabilities based on the permissions applied to `Document1`.

Because `User1` possesses `Priv1` (based on role inheritance), `User1` is able to execute code protected with a `xdmp:security-assert("Priv1", "execute")` call; users who do not have the `Priv1` privilege cannot 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 the 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 in MarkLogic Server, included adding or deleting users, adding or deleting documents, changing passwords, and so on.

Users with the `admin-ui-user` role may view the Admin Interface but do not have access to data or the ability to make administrative changes. For more information, see The admin-ui-user role in *Administrating MarkLogic Server*.

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` Library Module. Use extreme caution when assigning users the `security` role, because it gives the user the ability to utilize or assign the `admin` role.

The `security` role does not have access to the Admin Interface. To access the Admin Interface, a user must have the `admin` role or the `admin-ui-user` role. The `security` role provides the privileges to execute functions in the `security.xqy` module, which has functions to perform actions such as creating users and creating roles. For details on managing security objects programmatically, see Creating and Configuring Roles and Users and User Maintenance Operations in the *Scripting Administrative Tasks Guide.*

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

Consider a simple scenario 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 or JavaScript 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 the `xdmp:security-assert()` function to ensure that only engineers make widgets and only account representatives sell widgets (if you are using JavaScript, you can similarly call `xdmp.securityAssert()` in your JavaScript function to protect the code). For example:

```
xquery version "1.0-ml"
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, the `xdmp:security-assert()` function call 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:

```
xquery version "1.0-ml"
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:

```
xquery version "1.0-ml"
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. Protecting Documents

The MarkLogic Server security model has a set of tools you can use to control access to documents. These tools control creating, inserting into, updating, and reading documents in a database. This section describes those tools.

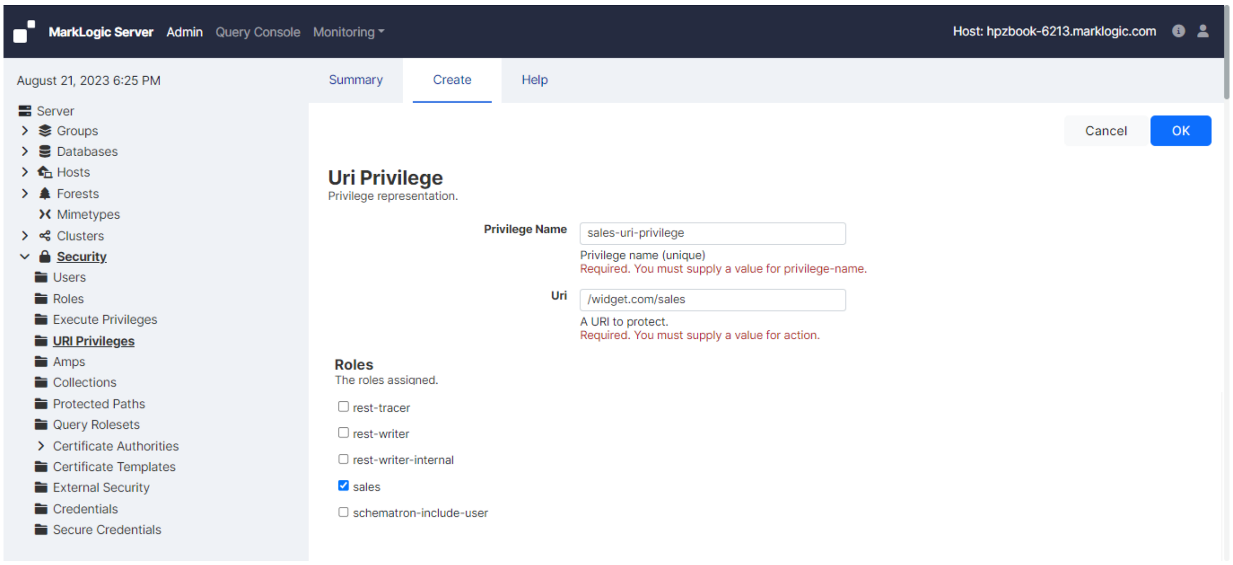
3.1. Creating Documents

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 is controlled with URI privileges and with two built-in execute privileges (`any-uri` and `unprotected-uri`). 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.

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 is protected. Users must be part of the `sales` role 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`.



August 21, 2023 6:25 PM

Uri Privilege
Privilege representation.

Privilege Name	sales-uri-privilege Privilege name (unique) Required. You must supply a value for privilege-name.
Uri	/widget.com/sales A URI to protect. Required. You must supply a value for action.

Roles
The roles assigned.

- ☐ rest-tracer
- ☐ rest-writer
- ☐ rest-writer-internal
- ☑ sales
- ☐ schematron-include-user

3.1.2. Built-In 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. Document Permissions

Permissions set on a document define access to capabilities (`read`, `insert`, `update`, `node-update`, and `execute`) for that document. Each permission consists of a capability and a role. This section describes how to set permissions on a document.

3.2.1. Capabilities Associated through Permissions

Document permissions pair a role with a capability to perform some action on a document. You can add multiple permissions to a document. If a user is part of a role (either directly or through inheriting the role) specified as part of a document permission, then the user has that capability for the given document. This section describes the capabilities you can assign to a role using permissions.

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 modify the document.

Update

The `update` capability provides the authority to modify content in the document or delete the document. However, `update` does not provide the authority to read the document. Reading the document requires the `read` capability. Users with `update` capability, but not `read` capability, can call the `xdmp:document-delete()` and `xdmp:document-insert()` functions successfully. However, node update functions, such as `xdmp:node-replace()`, `xdmp:node-delete()`, and `xdmp:node-insert-after()`, cannot be called successfully. 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 the `xdmp:document-add-permissions()` function to add a new permission to the document with `read` capability for a given role. A user with both `read` and `update` capabilities can call node update functions successfully.

Node Update

The `node-update` capability provides a subset of the `update` capability, enabling permission to update nodes within a document. The `node-update` capability offers finer control of updates when combined with element level security. The `node-update` capability covers `xdmp:node-replace()` and `xdmp:node-delete()` and can also be used in built-ins on properties, including `xdmp:document-add-properties()`, `xdmp:document-set-property()`, `xdmp:document-set-properties()` and `xdmp:document-remove-properties()`. Note that if a role has the `update` capability, it automatically includes the `node-update` capability as well.

Insert

The `insert` capability provides a subset of the `update` capability. The `insert` capability provides the authority to add new content to the document. The `insert` capability by itself does not allow a user to change existing content or remove an existing document (for example, calls to `xdmp:document-insert()` and `xdmp:document-delete()` on an existing document fail). Furthermore, you need `read` capability on the document to perform actions that use any of the node insert functions (`xdmp:node-insert-before()`, `xdmp:node-insert-after()`, `xdmp: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.

Execute

The `execute` capability 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. Users without permissions for the `execute` capability on a stored module, are not 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. Additionally, non-admin users must add update permissions on documents when creating them; attempts to create a document without at least one update permission result in an `XDMP-MUSTHAVEUPDATE` exception.

You set document permissions in the following ways:

- Explicitly set permissions on a document at load time (as a parameter to `xdmp:document-load()` or `xdmp:document-insert()`, for example).
- Explicitly set and remove permissions on a document using the following functions:
  - `xdmp:document-add-permissions()`
  - `xdmp:document-set-permissions()`
  - `xdmp:document-remove-permissions()`
- Implicitly set permissions when the document is created based on the default permissions of the user who creates the documents. Permissions are applied to 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.

3.3. Securing Collection Membership

You can also secure membership in collections by assigning permissions to collections. To assign permissions to collections, you must use the Admin Interface or the `security.xqy` Library Module functions. You cannot assign permissions to collections implicitly with default permissions.

For more information about permissions on collections, see Collections and Security in the *Search 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 (by using `xdmp:document-load()` or `xdmp:document-insert()`, for example), then the permissions are set to the *default permissions*. The default permissions are determined based on the roles assigned (both 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 are creating documents in a database, it is important to configure default permissions for the roles assigned to that user. Without default permissions, it is easy to create documents that no users (except those with the `admin` role) can read, update, or delete.

3.5. Example—Using Permissions

It is important to consider document permissions when you load content into a database, whether you load data using the built-in functions (for example, `xdmp:document-load()` or `xdmp:document-insert()`), WebDAV (for example, dragging and dropping files into a WebDAV folder), the REST API, the Java API, or a custom program. In each case, setting permissions is necessary, whether explicitly or by taking advantage of default permissions. 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 with the `engineering` role 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. To implement this security model, the document should be created with `read` and `insert` permissions for the `engineering` role and `read` and `update` permissions for the `engineering-manager` role.

This section describes the two ways to apply permissions to documents at creation time.

3.5.1. Setting Permissions Explicitly

Assume that the following code snippet is executed as user `Ron` of the `engineering` role. The code inserts a document with the following permissions:

- `read` and `insert` permissions for the `engineering` role
- `update`, `node-update`, and `read` permissions for the `engineering-manager` role

```
...
xdmp:document-insert("/widget.com/engineering/features/2017-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"),
        xdmp:permission("engineering-manager", "node-update"))
...
```

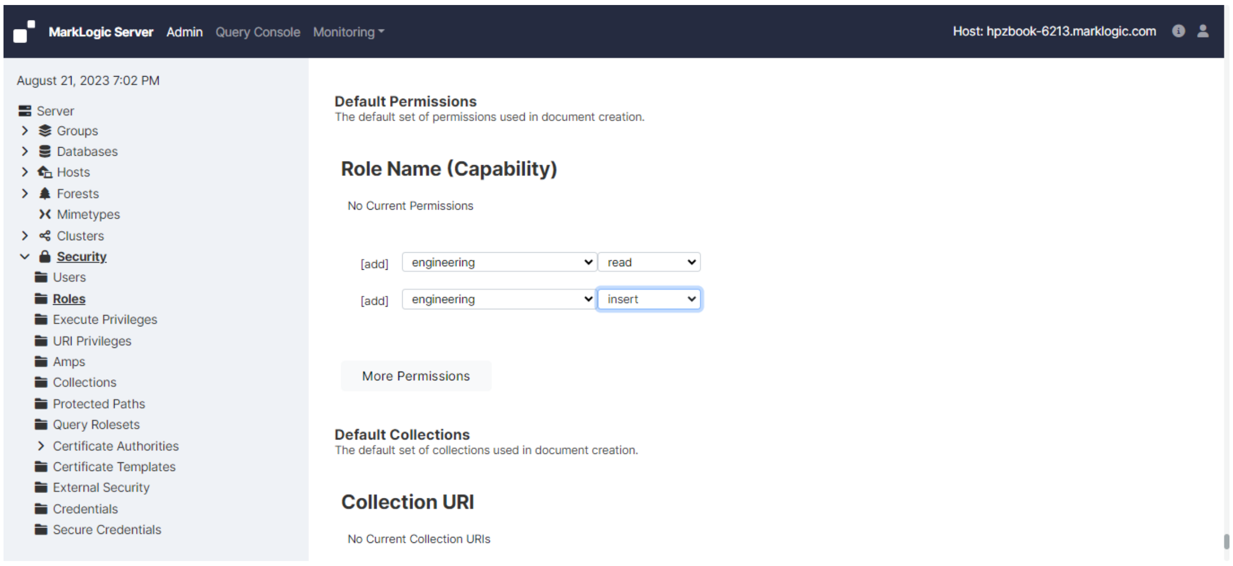
If you specify permissions to the function call explicitly, as shown above, those permissions override any default permission settings associated with the user (through user settings and role inheritance).

3.5.2. Default Permission Settings

If there is a set of permission requirements that meets the needs of most application scenarios, MarkLogic Server recommends 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:document-load` or `xdmp:document-insert`.

Default permission settings that apply to a user, either through a role or through the user definition, are important if you are loading documents using a WebDAV client. When you drag and drop files into a WebDAV folder, the permissions are automatically set based on the default permissions of the user logged into the WebDAV client. For more information about WebDAV servers, see WebDAV Servers in *Administrating MarkLogic Server*.

The following screenshot shows a portion of the Admin Interface for the `engineering` role. It shows `read` and `insert` capabilities being added to the `engineering` role's default permissions.



Default Permissions
The default set of permissions used in document creation.

Role Name (Capability)

No Current Permissions

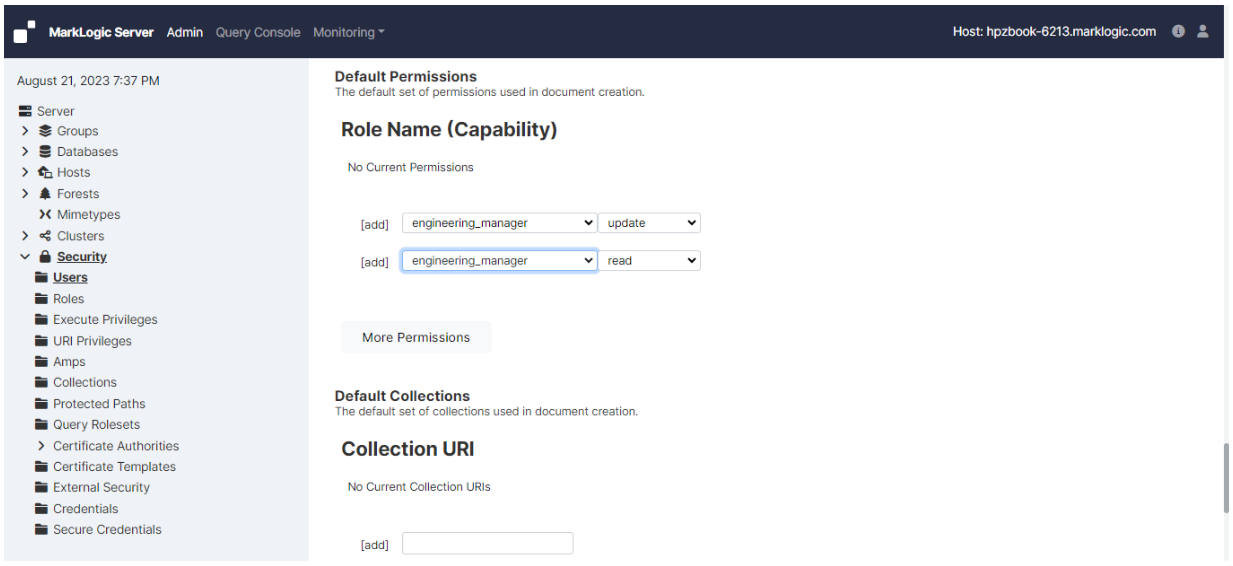
[add] engineering read
[add] engineering insert

Default Collections
The default set of collections used in document creation.

Collection URI

No Current Collection URIs

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 portion of a User configuration screen for Ron. It shows read and update capabilities being added to the `engineering-manager` role.



Default Permissions
The default set of permissions used in document creation.

Role Name (Capability)

No Current Permissions

[add] engineering_manager update

[add] engineering_manager read

Default Collections
The default set of collections used in document creation.

Collection URI

No Current Collection URIs

[add]

NOTE
A user does not need to have a certain role in order to specify that role in the 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 for the `engineering` role 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 achieves the desired objective of giving the `engineering-manager` role `read`, `update`, and `node-update` capabilities on the document, and the `engineering` role `read` and `insert` capabilities.

```
...
xdmp:document-insert("/widget.com/engineering/features/2017-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:permission("engineering-manager", "node-update"))
...
```

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

NOTE

The `xdmp:document-insert()` function performs 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 fails 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/2017-q1.xml",
  xdmp:permission("sales", "read"))
...
```

The `update` capability is needed to add permissions to a document, and the `node-update` capability is needed to update a portion of a document (or node). Therefore, the code snippet only succeeds if it is 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.

But what if the Emily is now the person in sales assigned to the project? Ian has the `node-update` capability, so he can call `xdmp:node-replace()` and `xdmp:node-delete()` to modify nodes in a document. Ian changes the “assigned-to” element in the document using `xdmp:node-update`.

```
...
xdmp:node-update("/widget.com/engineering/features/2017-q1.xml",
       <new-features>
         <feature>
           <name>blue whistle</name>
           <assigned-to>Emily</assigned-to>
           ...
          </feature>
         ...
       </new-features>,
```

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 documentation for the MarkLogic Server Built-In Functions in the *XQuery and XSLT Functions by Category* reference for more details.

4. Authenticating Users

MarkLogic Server authenticates users when they access an application. This section describes users and the available authentication schemes.

When you create or edit your app server, you choose the authentication scheme through the Authentication field. See the relevant section in *Administrating MarkLogic Server*:

- Creating a new HTTP Server.
- Creating a new XDBC Server.
- Creating a new WebDAV Server.
- Creating a new ODBC Server.

4.1. Users

A *user* in MarkLogic Server is the basis for authenticating requests to a MarkLogic Server application server. Users are assigned to roles. Roles carry security attributes, such as privileges and default permissions. Permissions assigned to documents pair a role with a capability. Therefore, roles are central to document permissions. Users derive authorization to perform actions from their roles.

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 the security attributes associated with a given user, click the `User:username` link on the Admin Interface screen for the given user. For details on configuring users in the Admin Interface, see Security Administration in *Administrating MarkLogic Server*.

During the initial installation of MarkLogic Server, four users are created:

- `Authorized administrator` - has the `admin` role. During the installation, you are prompted to specify the username and password for this user.
- `nobody` - this user is created with the `rest-reader`, `rest-extension-user`, `app-user`, and `harmonized-reader` roles. A password is randomly generated.
- `healthcheck`
- `infostudio-admin`

For details about installing MarkLogic Server, see *Installing MarkLogic Server*.

4.2. Types of Authentication

You can control the authentication scheme for HTTP, WebDAV, ODBC, and XDBC App Servers. This section describes these authentication schemes.

NOTE
Since the browser does not provide a way to clear a user's authentication information in basic, basic-digest, or digest, 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.

4.2.1. Basic

Basic authentication is available on all server types.

Basic is the typical authentication scheme for web applications. A user accessing an application page is prompted for a username and password. With basic, the password is obfuscated but not encrypted.

Basic can be used with internal security, LDAP, and SAML as authorization schemes.

NOTE
Since the browser does not provide a way to clear a user's authentication information in basic, basic-digest, or digest, 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.

4.2.2. Digest

Digest authentication is available on all server types.

Digest works the same way as basic, but it also offers encryption of passwords sent over the network. A user accessing an application page is prompted for a username and password.

Digest can be used with internal security, LDAP, and SAML as authorization schemes.

NOTE
If you change an App Server from basic to digest authentication, it invalidates existing sessions. 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 computes the digest password by extracting the relevant information from the credentials supplied in basic mode.

NOTE
Since the browser does not provide a way to clear a user's authentication information in basic, basic-digest, or digest, 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.

4.2.3. Digest-Basic

Digest-basic authentication is available on all server types.

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

Digest-basic can be used with internal security, LDAP, and SAML as authorization schemes.

NOTE
Since the browser does not provide a way to clear a user's authentication information in basic, basic-digest, or digest, 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.

4.2.4. Certificate-Based

Certificate-based authentication is available on all server types.

Certificate-based authentication requires internal and external users and HTTPS clients to authenticate themselves to MarkLogic Server through a client certificate, either in addition to, or instead of a password.

Certificate-based authentication can take the following forms:

- MarkLogic Server authenticates an internal user through the common name in a certificate.
- MarkLogic Server authenticates an internal user through the distinguished name in a certificate by matching the distinguished name to an external name configured for an internal user.
- MarkLogic Server authenticates an external LDAP user through a certificate subject name with internal authorization.
- MarkLogic Server authenticates an external user through a certificate subject name with external authorization. The user is entirely defined externally to MarkLogic Server.
- MarkLogic Server authenticates through both a client certificate and a username and password. This method provides a greater level of security by requiring that the user provide a client certificate that matches the specified user.

Certificate-based authentication can be used with internal security, LDAP, and SAML as authorization schemes.

For details on Certificate-based authentication, see Certificate-Based Authentication.

4.2.5. Application-Level

Application-level authentication is available on all server types except XDBC.

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.

The default user should have the 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 holds. For an example of this type of configuration, see Using Custom Login Pages.

Application-level can be used with internal security, LDAP, and SAML as authorization schemes.

4.2.6. Kerberos Ticket

Kerberos ticket authentication is available on all server types except ODBC.

The user is authenticated by Kerberos, then a Kerberos session ticket is used to authenticate the user to access MarkLogic Server.

Kerberos ticket can be used with internal security and LDAP as authorization schemes.

4.2.7. SAML

SAML authentication is available on all server types except ODBC.

When SAML authentication is used, a client requests a resource from MarkLogic Server with no security context. MarkLogic Server redirects the authentication request to an Identity Provider. The Identity Provider prompts the user to log in, if necessary, then sends the authentication request back to MarkLogic Server (the Service Provider) for validation.

There are two major components in SAML:

- The Identity Provider (IDP) authenticates a subject and provides security assertion to the Service Provider.
- The Service Provider (SP) provides access to the resource for a client. MarkLogic Server is a Service Provider.

MarkLogic Server sends a redirect to the resource. The client requests the resource again with a security context. MarkLogic Server then authenticates the user using the information from the authentication request to grant the user access to the requested resource.

SAML can be used only with the SAML authorization scheme.

4.2.8. OAuth

OAuth authentication is available on all server types.

There are three major components to OAuth:

- The Authorization Server, which authenticates a client and provides an Access Token.
- The Access Token, which is included in requests to the Resource Server.
- The Resource Server, which validates the Access Token and sends the requested resources to the client. MarkLogic Server is a Resource Server.

This is the OAuth authentication workflow:

1. The user sends their credentials to the client.
2. The client sends the user credentials to the Authorization Server: the OAuth vendor acting as the external agent.
3. The Authorization Server validates the user credentials.
4. The Authorization Server sends an Access Token to the client.
5. The client sends a resource request that includes the Access Token to the Resource Server: MarkLogic Server.
6. The Resource Server validates the Access Token.
7. The Resource Server sends the requested resources to the client.

To use OAuth, you must also configure External Security.

OAuth can be used only with the OAuth authorization scheme.

5. Compartment Security

MarkLogic Server includes an extension to the security model called compartment security. Compartment security allows you to specify more complex security rules on documents.

NOTE
An Advanced Security License is required when using compartment security. Licensing lists other security options requiring this license option. Contact your MarkLogic Server sales representative for details on purchasing the Advance Security License option.

This section describes compartment security.

5.1. Understanding Compartment Security

A *compartment* is a name associated with a role. You specify that a role is part of a compartment by adding the compartment name to each role in the compartment. When a role is *compartmented*, the compartment name is used as an additional check when determining a user's authority to access or create documents in a database. Compartments have no effect on execute privileges. Without compartment security, permissions are checked using OR semantics.

For example, if a document has `read` permission for `role1` and `read` permission for `role2`, a user who possesses *either* `role1` or `role2` can read that document. If those roles have different compartments associated with them (for example, `compartment1` and `compartment2`, respectively), then the permissions are checked using AND semantics for each compartment, as well as OR semantics for each non-compartmented role. To access the document if `role1` and `role2` are in different compartments, a user must possess both `role1` and `role2` to access the document, as well as a non-compartmented role that has a corresponding permission on the document.

If any permission on a document has a compartment, then the user must have that compartment in order to access any of the capabilities, even if the capability is not the one with the compartment.

Access to a document requires a permission in each compartment for which there is a permission on the document, regardless of the capability of the permission. So if there is a `read` permission for a role in `compartment1`, there must also be an `update` permission for some role in `compartment1` (but not necessarily the same role). If you try to add `read`, `insert`, `node-update`, or `execute` permissions that reference a compartmented role to a document for which there is no `update` permission with the corresponding compartment, the `XDMP-MUSTHAVEUPDATE` exception is thrown.

5.2. Configuring Compartment Security

You can only add a compartment for a new role. To add a compartment, use the Admin Interface and select **Security**> **Roles**> **Create** and enter a name for the compartment in the compartment field when you define each role in the compartment.

You cannot modify an existing role to use a compartment. To add a compartment to a role, you must delete the role and re-create it with a compartment. If you do re-create a role, any permissions you have on documents reference the old role (because they use the role ID, not the role name). So, if you want those document permissions to use the new role, you need to update those documents with new permissions that reference the new role.

5.3. Example—Compartment Security

This section describes a scenario that uses compartment security. The scenario is not meant to demonstrate *the* correct way to set up compartment security, as your situation is likely to be unique. However, it demonstrates how compartment security works and may give you ideas for how to implement your own security model:

For a MarkLogic Server application used by a government department, documents are classified with a security classification that dictates who may access the document. The department also restricts access to some documents based on the citizenship of the user. Additionally, some documents can only be accessed by employees with certain job functions.

To set up the compartment security for this scenario, you create the necessary roles, users, and documents with the example permissions. You will need access to both the MarkLogic Server Admin Interface and Query Console.

To run through the example, perform the steps in each of the sections.

5.3.1. Create Roles

Using the Admin Interface > **Security** > **Roles** > **Create**, create the roles and compartments as follows:

1. Create roles named `US` and `Canada` and assign each of these roles the `country` compartment name. These roles form the `country` compartment.
2. Create roles named `Executive` and `Employee` and assign each of these roles the `job-function` compartment name. These roles form the `job-function` compartment.
3. Create roles named `top-secret` and `unclassified` and assign each of these roles the `classification` compartment name. These roles form the `classification` compartment.
4. Create a role named `can-read` with no compartment.

5.3.2. Create Users

Using the Admin Interface > **Security** > **Users** > **Create**, create users and give them the roles indicated in the following table:

User	Roles
Don	`Executive`, `US`, `top-secret`, `can-read`
Ellen	`Employee`, `US`, `unclassified`, `can-read`
Frank	`Executive`, `Canada`, `top-secret`, `can-read`
Gary	`can-read`
Hannah	`unclassified`, `can-read`

5.3.3. Create the Documents and Add Permissions

Using the MarkLogic Server Query Console, add a document for each combination of permissions in the following table:

Document	Permissions (Role, Capability)	Users with Access
`doc1.xml`	(`Executive`, `read`) (`Executive`, `update`) (`US`, `read`) (`US`, `update`) (`top-secret`, `read`) (`top-secret`, `update`) (`can-read`, `read`) (`can-read`, `update`)	Don

Document	Permissions (Role, Capability)	Users with Access
doc2.xml	(US, read) (US, update) (can-read, read) (can-read, update)	Don and Ellen
doc3.xml	(can-read, read) (can-read, update)	All users
doc4.xml	(Canada, read) (US, read) (US, update) (can-read, read) (can-read, update)	Frank, Don, Ellen
doc5.xml	(unclassified, read) (unclassified, update) (can-read, read) (can-read, update)	Ellen, Hannah

1. You can use XQuery code similar to the following example to insert the sample documents into a database of your choice. This code adds a document with a URI of `doc1.xml`, containing one <a> element and a set of five permissions.

```
xquery version "1.0-ml";
declare namespace html = "http://www.w3.org/1999/xhtml";
xdmp:document-insert(
         "/doc1.xml", <a>This is document 1.</a>,
            (xdmp:permission("can-read", "read"),
             xdmp:permission("can-read", "update"),
             xdmp:permission("US", "read"),
             xdmp:permission("US", "update"),
             xdmp:permission("Executive", "read"),
             xdmp:permission("Executive", "update"),
             xdmp:permission("top-secret", "read"),
             xdmp:permission("top-secret", "update")))
```

   The `doc1.xml` document can only be read by `Don` because the permissions designate all three compartments and `Don` is the only user with a role in all three of the necessary compartmented roles `Executive`, `US`, and `top-secret`, plus the basic `can-read` role.
2. Create the rest of the sample documents changing the sample code as needed. You need to change the document URI and the text to correspond to `doc2.xml`, `doc3.xml`, `doc4.xml`, and `doc5.xml` and modify the permissions for each document as suggested in the table in Create the Documents and Add Permissions.

5.3.4. Test It Out

Using Query Console, you can execute a series of queries to verify that the users can access each document as specified in the table in Create the Documents and Add Permissions.

For simplicity, this sample query uses `xdmp:eval()` and `xdmp:user()` to execute a query in the context of each different user. Modify the document URI and the username to verify the permissions until you understand how the compartment security logic works. If you added the roles, users, and documents as described in this scenario, the query results should match the table in Create the Documents and Add Permissions.

```
xquery version "1.0-ml";
declare namespace html = "http://www.w3.org/1999/xhtml";

xdmp:eval('fn:doc("/doc1.xml")', (),
 <options xmlns="xdmp:eval">
  <user-id>{xdmp:user("Don")}</user-id>
</options>)
```

6. Element Level Security

MarkLogic Server includes element level security, an addition to the security model that allows you to specify more complex security rules on specific elements in documents. The feature also can be applied to JSON properties in a document. Using element level security, parts of a document may be concealed from users who do not have the appropriate roles to view them. Users without appropriate permissions cannot view the secured element or JSON property using XPath expressions or queries. Element level security can conceal the XML element (along with properties and attributes) or JSON property so that it does not appear in any searches, query plans, or indexes, unless accessed by a user with a role included in query roleset.

Element level security protects elements or JSON properties in a document using a protected path, where the path to an element or property within the document is protected so that only roles belonging to a specific query roleset can view the contents of that element or property. Only users with specific roles that match the specific query roleset can view the elements or properties protected by element level security. You can set protection with element level security to conceal a document's sensitive contents in real time, and also control which contents can be viewed and/or updated by other users.

NOTE
See Interactions with Other MarkLogic Server Features for details about using element level security with SQL and semantic queries.

Permissions on an element or property are similar to permissions defined on a document. Elements or properties may contain all supported datatypes. Search results and update built-ins will honor the permissions defined at the element level. Element level security is applied consistently across all areas of the MarkLogic Server, including reads, updates, query plans, and so on.

The protected paths are in the form of XPath expressions (not fields) that specify that an XML element or JSON property is part of a protected path. You will need to install or upgrade to MarkLogic 9.0-1 or later to use element level security.

6.1. Understanding Element Level Security

Elements of a document can be protected from being viewed as part of a query or XPath expression, or from being updated by a user, unless that user has the appropriate role. You specify that an element is part of a protected path by adding the path to the Security database. You also then add the appropriate role to a query roleset, which is also added to the Security database.

Element level security uses query rolesets to determine which elements will appear in query results. If a query roleset does not exist with the associated role that has permissions on the path, the role cannot view the contents of that path.

NOTE
A user with admin privileges can access documents with protected elements by using `fn:doc()` to retrieve documents (instead of using a query). To see protected elements as part of query results, however, a user needs the appropriate role(s).

6.2. Example—Element Level Security

This section describes a scenario using element level security. The scenario is not meant to demonstrate *the* correct way to set up element level security, as your situation is likely to be unique. However, it demonstrates how element level security works and may give you ideas for how to implement your own security model. You will need access to both the MarkLogic Server Admin Interface and Query Console. Install or upgrade to MarkLogic Server 9.0-x or later prior to starting this example:

For a MarkLogic Server application used by a department, certain parts of documents may be hidden so that only users with the correct role may view or update those parts of the document. Users without the proper role will not be able to see the element concealed by the protected path.

To set up the element level security for this scenario, follow the steps in each subsection.

6.2.1. Create Roles

Using the Admin Interface, create the roles as follows. You will create two roles, `els-role-1` and `els-role-2`.

1. In the Admin Interface, click **Security** in the left tree menu.
2. Click **Roles** and then click the **Create** tab.
3. On the Role Configuration page, enter the information for the first role: **role name**: `els-role-1`, **description**: `els role 1`.
4. Click **OK** to save the role.
5. Repeat these steps to create the second role (`els-role-2`, `els role 2`).

See Roles in *Administrating MarkLogic Server* for details about creating roles.

6.2.2. Create Users and Assign Roles

Now create three users (`els-user-1`, `els-user-2`, and `els-user-3`) using the Admin Interface. Assign roles to two of the users.

1. In the Admin Interface, click **Security** in the left tree menu.
2. Click **Users** and then click **Create**.
3. On the User Configuration page, enter the information for the first user: **user name**: `els-user-1`, **description**: `ELS user 1`, **password**: `<password>`.
   Enter a password of your choice.

Add this user to the first role that you created (`els-role-1`):

1. Scroll down the User Configuration page until you see the `els-role-1` role you just created.
2. Click the box next to `els-role-1` to assign the role to the user.
3. Click **OK** to save your changes.

Repeat these steps to create a second user and third user (`els-user-2`, `ELS user 2`, `els-user-3`, `ELS user 3`). Assign the matching roles to the users.

See Users in *Administrating MarkLogic Server* for details on creating users.

NOTE

Admin users must be added to a role in order to view the results of a query on protected paths that involve concealed elements.

6.2.3. Add the Documents

For our simple example, we will use three documents, two in XML and one in JSON. Use the Query Console to insert these documents into the Documents database, along with read and update permissions for `els-user-1` and `els-user-2`:

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

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

xdmp:document-insert("test1.xml",
<root>
  <bar baz="1" attr="test">abc</bar>
  <bar baz="2">def</bar>
  <bar attr="test1">ghi</bar>
</root>,
(xdmp:permission("els-role-1", "read"), xdmp:permission("els-role-2", "read"),
xdmp:permission("els-role-1", "update"),
xdmp:permission("els-role-2", "update")))
,
xdmp:document-insert("test2.xml",
<root>
  <reg expr="this is a string">1</reg>
  <reg>2</reg>
</root>,
(xdmp:permission("els-role-1", "read"), xdmp:permission("els-role-2", "read"),
xdmp:permission("els-role-1", "update"),
xdmp:permission("els-role-2", "update")))
,
xdmp:document-insert("test1.json", object-node {
"foo" : 1, "bar" : "2",  "baz" : object-node
{"bar" : array-node {3,4}, "test" : 5}
},
(xdmp:permission("els-role-1", "read"), xdmp:permission("els-role-2", "read"),
xdmp:permission("els-role-1", "update"),
xdmp:permission("els-role-2", "update")))
```

The code example adds permissions to the documents for `els-role-1` and `els-role-2` while inserting them into the database.

6.2.4. Add Protected Paths and Query Rolesets

Using the Admin Interface, add the protected paths and query rolesets to the Security database. If no query rolesets are configured, a query will only match documents by the terms that are visible to everyone.

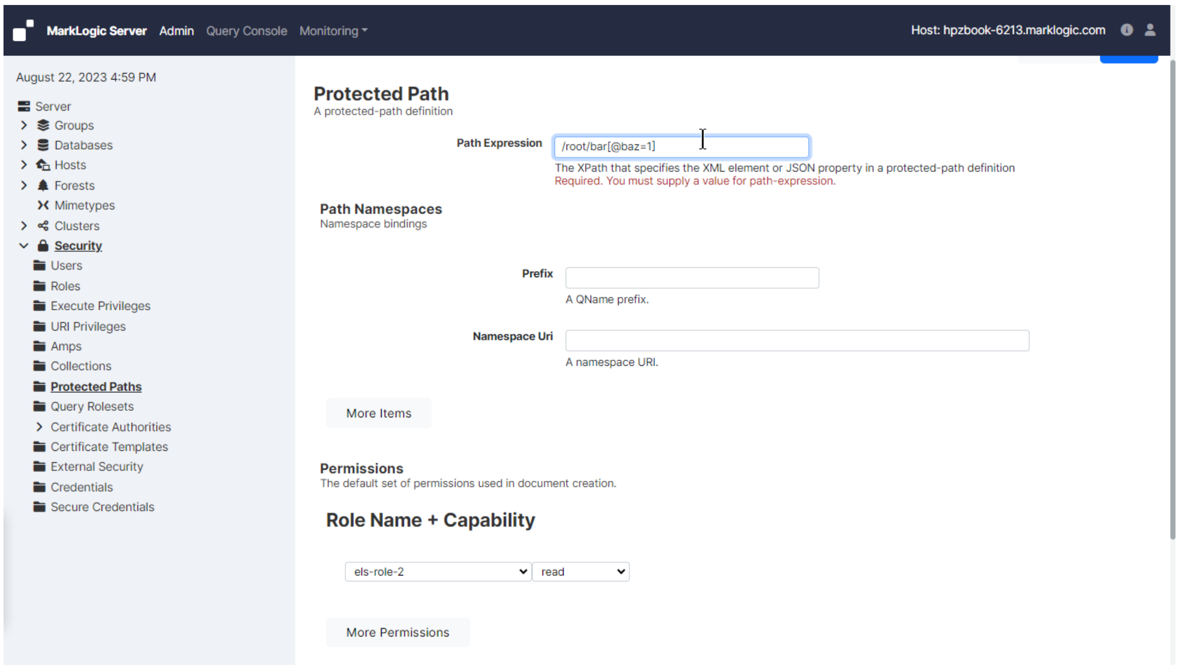
To start, check for any existing protected paths using this query in the Query Console:

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

fn:collection("http://marklogic.com/xdmp/protected-paths")
```

This will return an empty sequence if there are no protected paths. If there are protected paths, information about those protected paths will be displayed, including the path ID, the path expression, the permissions, and roles associated with that path.

Using the Admin Interface, add protected paths with permissions for `els-user-2`. To add the protected path from the Admin Interface:

1. Click **Security** in the left tree menu.
2. Click **Protected Paths** and then click the **Create** tab.
3. In the **Path Expression** field, enter the path expression for the first path (`/root/bar[@baz=1]`).



Protected Path
A protected-path definition

Path Expression: /root/bar[@baz=1]
The XPath that specifies the XML element or JSON property in a protected-path definition
Required. You must supply a value for path-expression.

Path Namespaces
Namespace bindings

Prefix:
A QName prefix.

Namespace Uri:
A namespace URI.

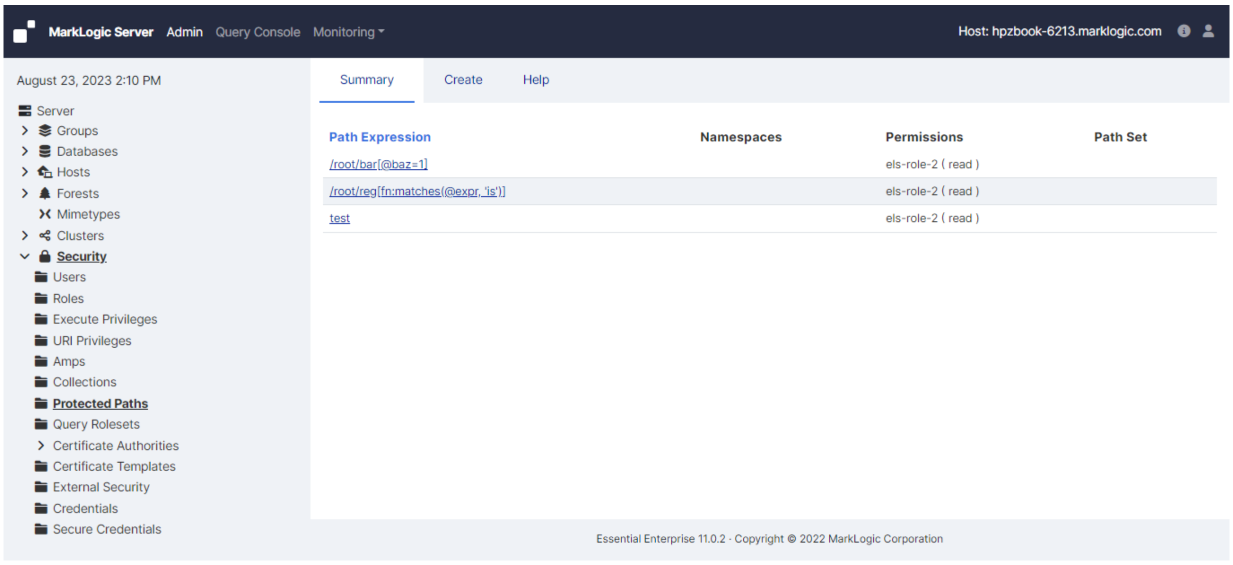
Permissions
The default set of permissions used in document creation.

Role Name + Capability
els-role-2 read

4. Grant read permissions. Under **Role Name + Capability**, from the first drop-down, select `els-role-2`, and from the second drop down, select **read**.
5. Click **OK** when you are done. Since there are no namespaces in these examples, the prefix and namespace are not required for the protected path.

For examples using namespaces and prefixes as part of a protected path, see Namespaces as Part of a Protected Path.

Repeat this for two additional protected paths, "`test`" and "`/root/reg[fn:matches(@expr, 'is')]`".



Path Expression	Namespaces	Permissions	Path Set
/root/bar[@baz=1]		els-role-2 (read)	
/root/reg[fn:matches(@expr, 'is')]		els-role-2 (read)	
test		els-role-2 (read)	

The three protected paths with read permissions for `els-role-2` are these:

```
/root/bar[@baz=1]
```

```
test
```

```
/root/reg[fn:matches(@expr, 'is')]
```

Alternatively, you can add these protected paths with the Query Console. Use this code to add these protected paths with permissions for `els-user-2` to the Security database:

```
(: add protected paths -> run against the Security database :)

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

sec:protect-path("/root/bar[@baz=1]", (), (xdmp:permission("els-role-2", "read"))),
sec:protect-path("test", (), (xdmp:permission("els-role-2", "read"))),
sec:protect-path("/root/reg[fn:matches(@expr, 'is')]", (), (xdmp:permission("els-role-2",
"read")))

=> Returns three numbers representing the protected paths
```

NOTE
Adding, unprotecting, or changing permissions on protected paths will trigger reindexing. This reindexing will only apply to documents that include or match the paths.

Now add query rolesets for these documents. In the Query Console, run this code to add query rolesets for `els-user-2`:

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

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

let $qry := 'xdmp:database-node-query-rolesets(fn:doc(), ("all"))'
let $qry-rolesets :=
xdmp:eval($qry, (),<options xmlns="xdmp:eval">
                   <database>{xdmp:database('Documents')}</database>
                 </options>)

return
sec:add-query-rolesets($qry-rolesets)
```

In most cases you will want to use the helper functions (`xdmp:database-node-query-rolesets` and `xdmp:node-query-rolesets`) to create query rolesets. The helper function automatically created the query rolesets based on the protected paths you have set. See Helper Functions for Query Rolesets for more information. To understand more about query rolesets, see Query Rolesets.

You can also add query rolesets manually with XQuery in the Query Console if you only have a few query rolesets to add. Run this code against the Security database:

```
(: add query rolesets => run against the Security database :)

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

let $roleset := sec:query-roleset("els-role-2")
return
sec:add-query-rolesets(sec:query-rolesets($roleset))
=>
Returns a unique ID representing the added query rolesets
```

NOTE
Adding query rolesets does not trigger reindexing, since it is only used by queries.

Check for query rolesets in the Security database using the Query Console:

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

fn:collection("http://marklogic.com/xdmp/query-rolesets")
=>
Returns details about query rolesets in the Security database.
```

There is also a collection for protected paths in the Security database:

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

fn:collection("http://marklogic.com/xdmp/protected-paths")
=>
Returns details about protected paths in the Security database.
```

The `els-role-2` can now see the elements in these paths, but the `els-user-1` cannot:

`test`

`/root/bar[@baz=1]`

`/root/reg[fn:matches(@expr, 'is')]`

6.2.5. Run the Example Queries

This section includes examples in both XQuery and JavaScript. Run the following queries in the Query Console. For simplicity, the sample queries use `xdmp:eval()` and `xdmp:get-current-user()` (or `xdmp.eval()` and `xdmp.getCurrentUser()`) to execute a query in the context of each user. Different elements and properties in a document are concealed for the different roles. Notice the different types of queries, using either XQuery or JavaScript, that are used to search for content.

NOTE
These examples assume that you have access permissions for both the MarkLogic Server Admin Interface and the Query Console.

XQuery Examples of Element Level Security

Run these queries on the Documents database using XQuery in Query Console. First run the queries in the context of `els-user-1`:

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

xdmp:eval(
'cts:search(fn:doc(), cts:word-query("def"), "unfiltered"),
"----------------------------------------------------",
cts:search(fn:doc(), cts:element-attribute-word-query(xs:QName("bar"), xs:QName("attr"),
"test"), "unfiltered"),
"---------------------------------------------------",
cts:search(fn:doc(), cts:json-property-value-query("bar", "2")),
"----------------------------------------------------",
cts:search(fn:doc(), cts:element-attribute-word-query(xs:QName("reg"), xs:QName("expr"),
"is"), "unfiltered")',
(),
  <options xmlns="xdmp:eval">
    <user-id>{xdmp:user("els-user-1")}</user-id>
  </options>
  )

=>
<?xml  version="1.0" encoding="UTF-8"?>
<root>
  <bar baz="2">def</bar>
  <bar attr="test1">ghi</bar>
</root>
-----------------------------------------------------


----------------------------------------------------
{
  "foo": 1,
  "bar": "2",
  "baz": {
    "bar": [
     3,
     4
    ]
  }
}
-----------------------------------------------------
```

Notice that in the first query, all of the documents are returned, but the elements with protected paths are missing from the content:

```
<bar baz="1" attr="test">abc</bar>
"test": 5
<reg expr="this is a string">1</reg>
```

In the second query, the document does not show up at all because the query is searching on a protected path that `els-user-1` is not allowed to see (protected path “`/root/bar[@baz=1]`”).

NOTE
If you are getting different results, check to see that you have set up your user roles correctly and added the query rolesets to the Security database.

Now, modify the query to use the context of the `els-user-2` and run the queries again:

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

xdmp:eval(
'cts:search(fn:doc(), cts:word-query("def"), "unfiltered"),
"-----------------------------------------------------",
cts:search(fn:doc(), cts:element-attribute-word-query(xs:QName("bar"), xs:QName("attr"),
"test1"), "unfiltered"),
"----------------------------------------------------",
cts:search(fn:doc(), cts:json-property-value-query("bar", "2")),
"-----------------------------------------------------",
cts:search(fn:doc(), cts:element-attribute-word-query(xs:QName("reg"), xs:QName("expr"),
"is"), "unfiltered")',
(),
  <options xmlns="xdmp:eval">
    <user-id>{xdmp:user("els-user-2")}</user-id>
  </options>
  )

=>
<?xml  version="1.0" encoding="UTF-8"?>
<root>
  <bar baz="1" attr="test">abc</bar>
  <bar baz="2">def</bar>
  <bar attr="test1">ghi</bar>
</root>
-----------------------------------------------------
<?xml  version="1.0" encoding="UTF-8"?>
<root>
  <bar baz="1" attr="test">abc</bar>
  <bar baz="2">def</bar>
  <bar attr="test1">ghi</bar>
</root>
----------------------------------------------------
{
  "foo": 1,
  "bar": "2",
  "baz": {
    "bar": [
      3,
      4
    ],
    "test": 5
  }
}
-----------------------------------------------------
<?xml  version="1.0" encoding="UTF-8"?>
<root>
  <reg expr="this is a string">1</reg>
  <reg>2</reg>
</root>
```

This time all of the documents are returned, along with the protected elements. Notice that the one document is returned twice; two different queries find the same document.

Run the query one more time using the `xdmp:eval()` pattern as `els-user-3` and notice that none of the documents are returned because `els-user-3` does not have the basic permissions to read the documents.

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

xdmp:eval(
'cts:search(fn:doc(), cts:word-query("def"), "unfiltered"),
"----------------------------------------------------",
cts:search(fn:doc(), cts:element-attribute-word-query(xs:QName("bar"), xs:QName("attr"),
"test1"), "unfiltered"),
"---------------------------------------------------",
cts:search(fn:doc(), cts:json-property-value-query("bar", "2")),
"----------------------------------------------------",
cts:search(fn:doc(), cts:element-attribute-word-query(xs:QName("reg"), xs:QName("expr"),
"is"), "unfiltered")',
(),
  <options xmlns="xdmp:eval">
    <user-id>{xdmp:user("els-user-3")}</user-id>
  </options>
  )

=>
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
```

Because `els-user-3` does not have document level permissions, no documents are returned. You can use document level permissions along with element level security for additional security. See Combining Document and Element Level Permissions for more information.

Now unprotect the paths and run the previous query again without the protected paths to see difference in output. First unprotect the paths:

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

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

sec:unprotect-path("/root/bar[@baz=1]", ()),
sec:unprotect-path("test", ()),
sec:unprotect-path("/root/reg[fn:matches(@expr, 'is')]", ())
```

NOTE
Adding or unprotecting protected paths will trigger reindexing. After unprotecting elements, you must wait for reindexing to finish.

Unprotecting the paths does not remove them from the database. You will still see the protected paths in the Admin Interface or when you run `fn:collection("http://marklogic.com/xdmp/protected-paths")` against the Security database. But you will be able to see the whole document once the protected paths are unprotected if you have document permissions for the document. See Unprotecting or Removing Paths for more details.

Look through the code examples and run the queries using the `xdmp:eval()` pattern to change users. Run the queries in the context of the different users to better understand how the element level security logic works.

JavaScript Examples of Element Security

You can also query the documents using Server-Side JavaScript. Run these JavaScript queries, using the previous users and documents, on the Documents database in Query Console.

First run the queries in the context of `els-user-1`:

```
// run this against the Documents database

var prog1 = `cts.search(cts.wordQuery("def"), "unfiltered")`;
var prog2 = `cts.search(cts.elementAttributeWordQuery(xs.QName("bar"), xs.QName("attr"),
"test1"), "unfiltered")`;
var prog3 = `cts.search(cts.jsonPropertyValueQuery("bar", "2"))`;
var prog4 = `cts.search(cts.elementAttributeWordQuery(xs.QName("reg"), xs.QName("expr"),
"is"), "unfiltered")`;
var res = [];
res.push(xdmp.eval(prog1, null, {userId:xdmp.user("els-user-1")}));
res.push(xdmp.eval(prog2, null, {userId:xdmp.user("els-user-1")}));
res.push(xdmp.eval(prog3, null, {userId:xdmp.user("els-user-1")}));
res.push(xdmp.eval(prog4, null, {userId:xdmp.user("els-user-1")}));
res;
=>
[
"<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n
<root><bar baz=\"2\">def</bar>
<bar attr=\"test1\">ghi</bar>
</root>",
"<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n
<root>
<bar baz=\"2\">def</bar>
<bar attr=\"test1\">ghi</bar>
</root>",
  {
  "foo": 1,
    "bar": "2",
    "baz": {
      "bar": [
        3,
        4
      ]
    }
  },
  null
]
```

Notice that all of the documents are returned, but the elements with protected paths are missing from the content:

```
<bar baz="1" attr="test">abc</bar>
"test": 5
<reg expr="this is a string">1</reg>
```

In the second query, the document does not show up at all because the query is searching on a protected path that `els-user-1` is not allowed to see (protected path “test”).

NOTE
If you are getting different results, check to see that you have set up your user roles correctly and added the query rolesets to the Security database.

Now, modify the query to use the context of the `els-user-2` and run the queries again:

```
// run this against the Documents database

var prog1 = `cts.search(cts.wordQuery("def"), "unfiltered")`;
var prog2 = `cts.search(cts.elementAttributeWordQuery(xs.QName("bar"), xs.QName("attr"),
"test1"), "unfiltered")`;
var prog3 = `cts.search(cts.jsonPropertyValueQuery("bar", "2"))`;
var prog4 = `cts.search(cts.elementAttributeWordQuery(xs.QName("reg"), xs.QName("expr"),
"is"), "unfiltered")`;
var res = [];
res.push(xdmp.eval(prog1, null, {userId:xdmp.user("els-user-2")}));
res.push(xdmp.eval(prog2, null, {userId:xdmp.user("els-user-2")}));
res.push(xdmp.eval(prog3, null, {userId:xdmp.user("els-user-2")}));
res.push(xdmp.eval(prog4, null, {userId:xdmp.user("els-user-2")}));
res;
=>
[
"<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n
<root>
<bar baz=\"1\" attr=\"test\">abc</bar>
<bar baz=\"2\">def</bar>
<bar attr=\"test1\">ghi</bar>
</root>",
"<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n
<root><bar baz=\"1\" attr=\"test\">abc</bar>
<bar baz=\"2\">def</bar>
<bar attr=\"test1\">ghi</bar>
</root>",
  {
  "foo": 1,
    "bar": "2",
    "baz": {
      "bar": [
       3,
       4
      ]
    ,
"test": 5
    }
  },
"<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n
<root>
<reg expr=\"this is a string\">1</reg>
<reg>2</reg>
</root>"
]
```

This time all of the documents are returned, along with the protected elements. Notice that the one document is returned twice; two different queries will find the same document.

Run the query one more time using the `xdmp:eval()` pattern as `els-user-3` and notice that none of the documents are returned because `els-user-3` does not have the basic permissions to read the documents.

```
// run this against the Documents database

var prog1 = `cts.search(cts.wordQuery("def"), "unfiltered")`;
var prog2 = `cts.search(cts.elementAttributeWordQuery(xs.QName("bar"), xs.QName("attr"),
"test1"), "unfiltered")`;
var prog3 = `cts.search(cts.jsonPropertyValueQuery("bar", "2"))`;
var prog4 = `cts.search(cts.elementAttributeWordQuery(xs.QName("reg"), xs.QName("expr"),
"is"), "unfiltered")`;
var res = [];
res.push(xdmp.eval(prog1, null, {userId:xdmp.user("els-user-3")}));
res.push(xdmp.eval(prog2, null, {userId:xdmp.user("els-user-3")}));
res.push(xdmp.eval(prog3, null, {userId:xdmp.user("els-user-3")}));
res.push(xdmp.eval(prog4, null, {userId:xdmp.user("els-user-3")}));
res;
=>
[
null,
null,
null,
null
]
```

Because `els-user-3` does not have document level permissions, no documents are returned. You can use document level permissions along with element level security for additional security. See Combining Document and Element Level Permissions for more information.

Now unprotect the paths and run the previous query again without the protected paths to see difference in output. Unprotect the paths :

```
//run this against the Security database

var security = require('/MarkLogic/security.xqy');
declareUpdate();

security.unprotectPath('/root/bar[@baz=1]', []);
security.unprotectPath('test', []);
security.unprotectPath('/root/reg[fn:matches(@expr, "is")]', []);
```

NOTE
Adding, unprotecting, or changing permissions on protected paths will trigger reindexing. After unprotecting elements, you must wait for reindexing to finish.

Unprotecting the paths does not remove them from the database. You will still see the protected paths in the Admin Interface or when you run `fn:collection("http://marklogic.com/xdmp/protected-paths")` against the Security database. But if you are `els-role-1` or `els-role-2`, you will be able to see the whole document once the protected paths are unprotected, if you have document permissions for the document (like `els-role-1` and `els-role-2` but not `els-role-3`). See Unprotecting or Removing Paths for more details.

Look through the code examples and run the queries using the `xdmp.eval()` pattern. Run the queries in the context of the different users to better understand how the element level security logic works.

6.2.6. Additional Examples

This section includes additional examples to try, both in XQuery and Server-Side JavaScript, that demonstrate the concealing of elements. Using `fn:doc()` instead of a CTS query to retrieve documents, different users will be able to view (or not view) protected elements. Since there is no query involved, query rolesets are not required.

These examples make use of the users and roles set up in the earlier example. (See Example—Element Level Security for details.) The first example shows hierarchies of permissions (top-secret, secret, and unclassified) in a document. The second example shows a slightly different way of protecting content with attributes. The example queries can be done in using XQuery or JavaScript.

XQuery - Query Element Hierarchies

Use this code to insert a new document (along with permissions) into the Documents database:

```
(: insert document with permissions => run against Documents database :)

xquery version "1.0-ml";

xdmp:document-insert(
"hierarchy.xml", <root>
 <title>Title of the Document</title>
 <summary>Summary of document contents</summary>
 <executive-summary>Executive summary of the document contents
   <secret>Only role having "secret" can read this
     <top-secret>Only role having "top-secret" can read this
     </top-secret>
   </secret>
</executive-summary>
<content>Contents of document
  <top-secret>Only role with "top-secret" can read this
     <secret>Only role with "secret" can read this</secret>
  </top-secret>
Unclassified content
</content>
</root>,
(xdmp:permission("els-role-1", "read"), xdmp:permission("els-role-2", "read"),
xdmp:permission("els-role-1", "update"), xdmp:permission("els-role-2", "update")))
```

Add protected paths with permissions for roles to the Security database:

```
(: add protected paths -> run against the Security database :)

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

sec:protect-path("secret", (), (xdmp:permission("els-role-2", "read"))),
sec:protect-path("top-secret", (), (xdmp:permission("els-role-1", "read")))

=>
Returns two numbers representing the protected paths
```

NOTE
Adding, unprotecting, or changing permissions on protected paths will trigger reindexing.

Test this example in the context of the different els-users. This first query uses the context of `els-user-1`:

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

xdmp:eval('fn:doc("hierarchy.xml")',(),
  <options xmlns="xdmp:eval">
    <user-id>{xdmp:user("els-user-1")}</user-id>
  </options>
)
=>
<root>
 <title>Title of the Document
 </title>
 <summary>Summary of document contents</summary>
 <executive-summary>Executive summary of contents
  </executive-summary>
 <content>Contents of document
  <top-secret>Only role with "top-secret" can read this</top-secret>
 Unclassified content</content>
</root>
```

The “top-secret” role (`els-user-1`) cannot see the elements marked with “secret”, only those that have no protected paths or marked with the protected path for “top-secret”. Next, run the query in the context of `els-user-2`:

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

xdmp:eval('fn:doc("hierarchy.xml")',(),
  <options xmlns="xdmp:eval">
    <user-id>{xdmp:user("els-user-2")}</user-id>
  </options>
)
=>
<root>
 <title>Title of the Document</title>
 <summary>Summary of document contents</summary>
 <executive-summary>Executive summary of contents
  <secret>Only role having "secret" can read this</secret></executive-summary>
 <content>Contents of document
 Unclassified content</content>
</root>
```

Notice that even though in the original document there is an element “secret” within the “top-secret” contents of the document, it is a child of the “top-secret” element and therefore hidden to users without the “top-secret” role.

The `els-user-1` (“top-secret”) cannot see the “secret” content unless you add the `els-role-2` to `els-user-1`. When you add the role, `els-user-1` will be able to see both the “secret” and “top-secret” elements.

If you run the query as `els-user-3`, the query returns an empty sequence. The `els-user-3` from the previous query does not have permission to even see the document.

XQuery - Matching by Paths or Attributes

This next example shows how protected paths can be used with `fn:contains()` and `fn:matches()`. The example uses the same roles from the previous example, adding a new role (`els-role-3`).

First unprotect the protected paths from the previous example:

```
(: unprotect the protected paths -> run against the Security database :)

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

sec:unprotect-path("secret", ()),
sec:unprotect-path("top-secret", ())
```

NOTE
Adding or unprotecting protected paths will trigger reindexing. After unprotecting elements, you must wait for reindexing to finish.

Create a new role `els-role-3` and add `els-user-3` to the role. See Create Roles and Create Users and Assign Roles for details.

Add a new document with permissions to the Documents database:

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

xquery version "1.0-ml";

xdmp:document-insert(
"attributes.xml", <root>
 <title>Document Title</title>
 <summary>Summary of document contents</summary>
 <executive-summary>Executive summary of contents
  <info attr="EU">Only role with "EU" attribute can read this summary </info>
  <info attr="UK">Only role with "UK" attribute can read this summary </info>
  <info attr="US">Only role with "US" attribute can read this summary </info>
</executive-summary>
 <content>Contents of document
  Unclassified content
  <notes>
    <info attr="EU">Only role with "EU" attribute can read this content</info>
    <info attr="UK">Only role with "UK" attribute can read this content</info>
    <info attr="US">Only role with "US" attribute can read this content</info>
  </notes>
 </content>
</root>,
(xdmp:permission("els-role-1", "read"), xdmp:permission("els-role-2", "read"),
xdmp:permission("els-role-3", "read"),
xdmp:permission("els-role-1", "update"), xdmp:permission("els-role-2", "update"),
xdmp:permission("els-role-3", "update")))
```

Add the new protected paths with permissions for roles to the Security database:

```
(: add new protected paths -> run against the Security database :)

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

sec:protect-path("//info[fn:matches(@attr, 'US')]", (), (xdmp:permission("els-role-1",
"read"))),
sec:protect-path("//info[fn:matches(@attr, 'UK')]", (), (xdmp:permission("els-role-2",
"read"),
  xdmp:permission("els-role-3", "read"))),
sec:protect-path("//info[fn:matches(@attr, 'EU')]", (), (xdmp:permission("els-role-3",
"read")))
=>
Returns three numbers representing the protected paths
```

NOTE
Adding, unprotecting, or changing permissions on protected paths will trigger reindexing.

Notice that the protected paths include attributes in the document elements. Also note that `els-role-3` has permissions for two protected paths (`@attr, 'UK'` and `@attr, 'EU'`).

Run this next query, similar to the previous queries, this time looking for the `attributes.xml` document. First query in the context of `els-user-1` who has a role that can see the “US” attribute:

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

xdmp:eval('fn:doc("attributes.xml")',(),
  <options xmlns="xdmp:eval">
    <user-id>{xdmp:user("els-user-1")}</user-id>
  </options>
)

=>
<?xml  version="1.0" encoding="UTF-8"?>
<root>
 <title>Document Title</title>
 <summary>Summary of document contents</summary>
 <executive-summary>Executive summary of contents
  <info attr="US">Only role having "US" attribute can read this summary</info>
 </executive-summary>
 <content>Contents of document
  Unclassified content
 <notes>
  <info attr="US">Only role having "US" attribute can read this content
  </info>
 </notes>
 </content>
</root>
```

Next modify the query to run in the context of `els-user-2`, who has a role that can see the “UK” attribute:

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

xdmp:eval('fn:doc("attributes.xml")',(),
  <options xmlns="xdmp:eval">
    <user-id>{xdmp:user("els-user-2")}</user-id>
  </options>
)

=>
<?xml  version="1.0" encoding="UTF-8"?>
<root>
 <title>Document Title</title>
 <summary>Summary of document contents</summary>
 <executive-summary>Executive summary of contents
  <info attr="UK">Only role having "UK" attribute can read this summary
  </info>
 </executive-summary>
 <content>Contents of document
  Unclassified content
 <notes>
  <info attr="UK">Only role having "UK" attribute can read this content</info>
 </notes>
 </content>
</root>
```

And finally modify the query to run in the context of `els-user-3`:

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

xdmp:eval('fn:doc("attributes.xml")',(),
  <options xmlns="xdmp:eval">
    <user-id>{xdmp:user("els-user-3")}</user-id>
  </options>
)

=>
<?xml  version="1.0" encoding="UTF-8"?>
<root>
 <title>Document Title</title>
 <summary>Summary of document contents</summary>
 <executive-summary>Executive summary of contents
  <info attr="EU">Only role having "EU" attribute can read this summary

  </info>
  <info attr="UK">Only role having "UK" attribute can read this summary

  </info>
 </executive-summary>
 <content>Contents of document
  Unclassified content

 <notes>
  <info attr="EU">Only role having "EU" attribute can read this content

  </info>
  <info attr="UK">Only role having "UK" attribute can read this content

  </info>
 </notes>
 </content>
</root>
```

The `els-user-3` has protected path permissions on both elements with the “EU” info attribute and the elements with the “UK” info attribute, so the `els-user-3` can see both elements. If you are getting different results, check to be sure that you created an `els-role-3` and added the `els-user-3` to that role.

NOTE
If you run the query in the context of the admin user, you will be able to see the entire document because the query is using `fn:doc`.

JavaScript - Query Element Hierarchies

You can also try these examples demonstrating concealed elements using JavaScript. Using `fn.doc()` instead of a CTS query to retrieve documents, different users will be able to view (or not view) protected elements. Since there is no query involved, query rolesets are not required.

Use this JavaScript code to insert this document (with permissions) into the Documents database:

```
// insert document with permissions -> run against Documents database

declareUpdate();
var perms = [xdmp.permission("els-role-1", "read"), xdmp.permission("els-role-2",
"read"),
xdmp.permission("els-role-1", "update"), xdmp.permission("els-role-2",
"update")
];
xdmp.documentInsert(
"hierarchy.xml", xdmp.unquote(`
<root>
 <title>Title of the Document</title>
 <summary>Summary of document contents</summary>
 <executive-summary>Executive summary of the document contents
   <secret>Only role having "secret" can read this
     <top-secret>Only role having "top-secret" can read this
     </top-secret>
   </secret>
</executive-summary>
<content>Contents of document
  <top-secret>Only role with "top-secret" can read this
     <secret>Only role with "secret" can read this</secret>
  </top-secret>
Unclassified content
</content>
</root>
`), {permissions: perms})
```

Add protected paths with permissions for roles to the Security database:

```
// add protected paths -> run against the Security database

declareUpdate();
var security = require('/MarkLogic/security.xqy');

security.protectPath('secret', [], [xdmp.permission("els-role-2", "read", "element")]),
security.protectPath('top-secret', [], [xdmp.permission("els-role-1", "read", "element")])
=>
Returns a number representing the protected paths
```

NOTE
Adding, unprotecting, or changing permissions on protected paths will trigger reindexing.

Test this example in the context of the different els-users. This query uses the context of `els-user-1`:

```
// run this query against the Documents database

xdmp.eval("fn.doc('hierarchy.xml')", null,
  {
    "userId" : xdmp.user("els-user-1")
  })
=>
<?xml  version="1.0" encoding="UTF-8"?>
<root>
  <title>Title of the Document</title>
  <summary>Summary of document contents</summary>
  <executive-summary>Executive summary of the document contents

  </executive-summary>
  <content>Contents of document

  <top-secret>Only role with "top-secret" can read this</top-secret>

  Unclassified content
  </content>
</root>
```

The “top-secret” role (`els-user-1`) cannot see the elements marked with “secret”, only those that have no protected paths or marked with the protected path for “top-secret”. Next, run the query in the context of `els-user-2`:

```
// run this query against the Documents database

xdmp.eval("fn.doc('hierarchy.xml')", null,
  {
    "userId" : xdmp.user("els-user-2")
  })
=>
<?xml  version="1.0" encoding="UTF-8"?>
<root>
  <title>Title of the Document</title>
  <summary>Summary of document contents</summary>
  <executive-summary>Executive summary of the document contents

  <secret>Only role having "secret" can read this</secret>
  </executive-summary>
  <content>Contents of document

  Unclassified content
  </content>
</root>
```

Notice that even though in the original document, there is an element “secret” within the “top-secret” contents of the document, it is a child of the “top-secret” element and therefore hidden to users without the “top-secret” role.

The `els-user-1` (“top-secret”) cannot see the “secret” content unless you add the `els-role-2` to `els-user-1`. When you add the role, `els-user-1` will be able to see both the “secret” and “top-secret” elements.

If you run the query as `els-user-3`, the query returns an empty sequence. The `els-user-3` from the previous query does not have permission to even see the document.

JavaScript - Matching by Paths or Attributes

This next example shows how protected paths can be used with `fn.contains()` and `fn.matches()`. The example uses the same roles from the previous example, adding a new role (`els-role-3`).

First unprotect the protected paths from the previous example:

```
// unprotect protected paths -> run against the Security database

declareUpdate();
var security = require('/MarkLogic/security.xqy');

security.unprotectPath('secret', []),
security.unprotectPath('top-secret', [])
```

NOTE
Adding, unprotecting, or changing permissions on protected paths will trigger reindexing.

Create a new role `els-role-3` and add `els-user-3` to the role. See Create Roles and Create Users and Assign Roles for details.

Add a new document to the Documents database:

```
// insert document and permissions -> run this against the Documents database

declareUpdate();
var perms = [xdmp.permission("els-role-1", "read"), xdmp.permission("els-role-2",
"read"),
xdmp.permission("els-role-3", "read"), xdmp.permission("els-role-1", "update"),
xdmp.permission("els-role-2", "update"), xdmp.permission("els-role-3", "update")
];
xdmp.documentInsert(
"attributes.xml", xdmp.unquote(`
<root>
  <title>Document Title</title>
  <summary>Summary of document contents</summary>
  <executive-summary>Executive summary of contents
    <info attr="EU">Only role with "EU" attribute can read this summary </info>
    <info attr="UK">Only role with "UK" attribute can read this summary </info>
    <info attr="US">Only role with "US" attribute can read this summary </info>
  </executive-summary>
  <content>Contents of document
  Unclassified content
    <notes>
    <info attr="EU">Only role with "EU" attribute can read this content</info>
    <info attr="UK">Only role with "UK" attribute can read this content</info>
    <info attr="US">Only role with "US" attribute can read this content</info>
    </notes>
  </content>
</root>
`), {permissions: perms})
```

Add the new protected paths with permissions for roles to the Security database:

```
// add new protected paths -> run against the Security database

declareUpdate();
var security = require('/MarkLogic/security.xqy');

security.protectPath("//info[fn:matches(@attr, 'US')]", [],[xdmp.permission("els-
role-1","read", "element")]),
security.protectPath("//info[fn:matches(@attr, 'UK')]", [],[xdmp.permission("els-role-2",
"read", "element"),
  xdmp.permission("els-role-3", "read", "element")]),
security.protectPath("//info[fn:matches(@attr, 'EU')]", [],
  [xdmp.permission("els-role-3", "read", "element")])

=>
Returns one number representing the protected paths
```

NOTE
Adding or changing permissions on protected paths will trigger reindexing.

Run the same queries as before, first in the context of `els-user-1`, who has a role that can see the “US” attribute:

```
// run this query against the Documents database

xdmp.eval("fn.doc('attributes.xml')", null,
  {
    "userId" : xdmp.user("els-user-1")
  });
=>
<?xml  version="1.0" encoding="UTF-8"?>
<root>
  <title>Document Title</title>
  <summary>Summary of document contents</summary>
  <executive-summary>Executive summary of contents
  <info attr="US">Only role with "US" attribute can read this summary</info>
  </executive-summary>
  <content>Contents of document
  Unclassified content

  <notes>
  <info attr="US">Only role with "US" attribute can read this content</info>
  </notes></content>
</root>
```

Next modify the query to run in the context of `els-user-2`,who has a role that can see the “UK” attribute:

```
// run this query against the Documents database

xdmp.eval("fn.doc('attributes.xml')", null,
  {
    "userId" : xdmp.user("els-user-2")
  });
=>
<?xml  version="1.0" encoding="UTF-8"?>
<root>
  <title>Document Title</title>
  <summary>Summary of document contents</summary>
  <executive-summary>Executive summary of contents
  <info attr="UK">Only role with "UK" attribute can read this summary</info></executive-
summary>
  <content>Contents of document
  Unclassified content

  <notes>
  <info attr="UK">Only role with "UK" attribute can read this content</info>
  </notes></content>
</root>
```

And finally modify the query to run in the context of `els-user-3`:

```
// run this query against the Documents database

xdmp.eval("fn.doc('attributes.xml')", null,
  {
    "userId" : xdmp.user("els-user-3")
  });
=>
<?xml  version="1.0" encoding="UTF-8"?>
<root>
  <title>Document Title</title>
  <summary>Summary of document contents</summary>
  <executive-summary>Executive summary of contents

  <info attr="EU">Only role with "EU" attribute can read this summary</info>
  <info attr="UK">Only role with "UK" attribute can read this summary</info>
  </executive-summary>
  <content>Contents of document
  Unclassified content
  <notes>
  <info attr="EU">Only role with "EU" attribute can read this content</info>
  <info attr="UK">Only role with "UK" attribute can read this   content</info>
  </notes></content>
</root>
```

The `els-user-3` has protected path permissions on both elements with the “EU” info attribute and the elements with the “UK” info attribute. So that user can see both elements.

NOTE
If you run the query in the context of the admin user, you will be able to see the entire document because the query is using `fn.doc()`.

6.3. Configuring Element Level Security

Configuring element level security includes setting up protected paths and creating query rolesets, then adding them to the Security database. This section covers the steps you will need to follow to configure element level security. As an overview, you will need to do the following:

- Set up roles.
- Create users and assign roles.
- Add or update documents with permissions for users.
- Add protected paths for elements in documents, by inserting the protected paths into the Security database.
- Add the query rolesets to the Security database.

Configuring the query rolesets is a task for the administrator. There are two helper functions to help configure query rolesets. The helper function `xdmp:database-node-query-rolesets()` is used for querying documents already in your database to discover existing query rolesets, while `xdmp:node-query-rolesets()` is used to query for protected paths in documents as they are being loaded into the database. See APIs for Element Level Security for more information. You can configure element level security using the Admin Interface, using XQuery, or by using REST.

NOTE

The number of protected paths that you set on a single element may impact performance. One or two protected paths on an element will have no discernible impact (less than 5% in our testing), 10 or so protected paths may have some impact (around 10%) but setting 100 or so protected paths on a single element will cause severe and noticeable impact on performance.

6.3.1. Protected Paths

You can define permissions on an element in the same way that you define permissions on a document. Element level security works by specifying an “indexable” path to an element (or JSON property) and configuring permissions on that path - creating a protected path.

For performance and security reasons, you can only use a subset of XPath for defining protect paths. For details, see Element Level Security in the *XQuery and XSLT Reference Guide*.

NOTE

The read, update, and insert permissions for an element are checked separately. For instance, if there are permissions for read, but no permissions for update or insert, there is no control for update or insert on that element. If there are no permissions on an element, anyone can read that element, given that they have the proper *document level permissions*.

Examples of Protected Paths

This table shows some examples of protected paths.

Protected Path	Permissions	Result
`/foo/bar`	(role1, read)	Element “bar” is readable by “role1” but concealed for all other roles. No mention of other permissions means that others can update or insert content for this element.
`/foo/bar`	(role1, read) (role2, read)	Element “bar” is readable by “role1” or “role2” but concealed for all other roles. No mention of other permissions means that others can update or insert content for this element.
`/foo/bar`	(role1, read) (role1, update)	Element “bar” is readable by “role1” but concealed for all other roles. “Role1” can update the element. No mention of insert permissions means that others can insert content for this element.
`/foo/bar[@attr= “test”]`	(role1, read) (role1, update)	Same as above except that it only applies to a bar element if the element has an attribute “attr” with the value “test”. No mention of insert permissions means that others can insert content for this element.
`bar`	(role1, read)	This is the simplest path. Element “bar” is readable by “role 1” but concealed for all other roles. This applies to all “bar” elements. No mention of other permissions means that others can update or insert content for this element.
`/root/reg[fn:matches(@expr, 'is')]`	(role1, read) (role1, update)	Elements that match the regular express for ‘is” will be readable by “role 1” but concealed for all other roles. “Role 1” can update the element. No mention of insert permissions means that others can insert content for this element.

For more about update permissions with element level security, see the table in the section Document and Element Level Permissions Summary.

WARNING
Defining element level security protection (protected paths) on “reserved” elements or properties (for example, alerting, thesaurus, and so on) may cause undefined behavior.

The path is an XPath expression, not a field.

Namespaces as Part of a Protected Path

Both namespaces and prefixes can be used as part of a protected path. For instance, this simple example uses the namespace “ex” as part of the protected path:

```
(: add protected paths -> run against the Security database :)
xquery version "1.0-ml";
import module namespace sec = "http://marklogic.com/xdmp/security"
  at "/MarkLogic/security.xqy";
declare namespace ex = "http://marklogic.com/example";
let $role := "role-4"
return
  sec:protect-path(
    "/ex:envelope/ex:instance/employee/salary",
    (let $prefix := "ex",$namespace-uri :=
      "marklogic.com/example"
    return
    sec:security-path-namespace($prefix, $namespace-uri)),
  (xdmp:permission($role, "read"))
  )
```

For simple cases, you can also specify a namespace as part of a protected path when configuring protected paths in the Admin Interface.

You can also specify a namespace when using the helper functions `xdmp:database-node-query-rolesets` and `xdmp:node-query-rolesets`. See page Helper Functions for Query Rolesets for more info.

Unprotecting or Removing Paths

Unprotecting protected paths does not remove them from the database, it removes the permissions, which disables the protection. You will still see the unprotected paths in the Admin Interface. The unprotected paths can also be seen by running `fn:collection("http://marklogic.com/xdmp/protected-paths")` against the Security database, in the Query Console.

Removing protected paths is a two-step process. First you must unprotect the path, and then you can remove it.

NOTE
You must first unprotect a path before removing it to trigger the reindexer. Since query rolesets changes don't require reindexing, there is no need for the separate step of unprotecting before removing a query roleset.

To unprotect a protected path:

1. Navigate to Protected Path Configuration by clicking **Security** and then **Protected Paths** in the left tree menu.
2. Click on the name of the protected path you want to unprotect.
3. On the Protected Path Configuration page there are two buttons; an unprotect button and a delete button (greyed out).
4. Click the **unprotect** button.
5. Click **OK** to save the changes.

When you have unprotected the protected paths, you'll see the protected paths on the Summary page, but no permissions are associated with the paths.

To remove a path, you will need to first unprotect the path. See Unprotecting or Removing Paths

1. After unprotecting the path, go back to the Protected Path Configuration page. Notice that the delete button is now available and the unprotect button is greyed out.

2. Click the **Delete** button to remove this protected path.
3. Click **OK** to confirm and save your changes.

The deleted path no longer appears on the Summary page of protected paths.

NOTE
Adding, unprotecting, or changing permissions for protected paths will trigger reindexing of the relevant databases. Having too many unprotected paths for your database can affect performance. Best practice is to delete unprotected paths when you no longer need them. Be sure to let the reindexing triggered by unprotecting finish before you delete the paths.

Performance Considerations with Protected Paths

The fewer protected paths that you have in your documents, the better performance you will have with element level security. One way to reduce the number of protected paths is to group information. If you have the ability to control the schema of your documents, you can group information that you want to protect under one element and then protect that element.

In this example, an insurance company has a schema that groups policy information to control access to the information, making it easier to protect client information and policy information by role (`US Read`, `ID_Read`, `Compliance`, and `Risk`):

```
"policy": {
"access": "US Read",
"client": {
   "access": "ID_Read",
   "name": "Paul",
   "address": "999 Broadway St",
   "phone": "323-344-1555",
   "country": "US",
   "ssn4digits": "5664"
     }
,
"clientSSN": {
   "access": "Compliance",
   "ssn": "999-999-5664"
     }
,
"clientIncome": {
   "access": "Risk",
   "income": "44,4444"
     }
,
"info": {
   "access": "Risk",
   "propertyType": "Home",
   "premium": 432,
   "assetValue": 750000,
   "currency": "Dollar"
     }
}
```

Different users would be able to see different parts of the data: the Call Center might have the `ID_Read` role, the Financial Risk Researcher might have the `Risk` role, and a Compliance Admin might have the `ID_Read`, `Risk`, and `Compliance` roles. Each of these would all need to have the `US Read` role as well.

If you don't have control of the schema and your document data is in various formats, you can leverage Entity Services as a way to improve performance. You can use entity services to create an entity that groups multiple elements under a single node and then use a single protected path on that node. See Introduction to Entity Services in the *Entity Services Developer's Guide* for information about creating an entity that links to the source document and protecting both.

6.3.2. Query Rolesets

What are query rolesets and what do they do? This section describes query rolesets and how they are used with element level security.

How Query Rolesets Work

When you add a document into MarkLogic Server, it parses the document and puts "terms" (or keys) into the universal index. Later when you run a query, the query side needs to know what terms to find in the universal index. In element level security, the terms are combined with permissions in the index. Existing query rolesets are automatically used by the query to figure out which terms to use, based on the role(s) of the user running the query. Each query can include multiple query rolesets. If no query rolesets are configured, a query will only match documents using the terms that are visible to everyone.

Let's use an example. Say you have a protected path defined as the following:

```
sec:protect-path("/root/bar[@baz=1]", (), (xdmp:permission("els-role-2",
"read")))
```

And then you ingest a document like this:

```
<root>
  <bar baz=1>Hello</bar>
</root>
```

When MarkLogic Server parses the document, it sees that the word "`Hello`" is inside the element `<bar>` that matches the protected path definition (since `bar` is under `root` and has an attribute `baz=1`). So instead of simply putting the term "`Hello`" into the universal index, it combines the term "`Hello`" and the permission information in the protected path (in this case, basically the role name "`els-role-2`") into one term and puts this new term into the universal index.

Suppose then you run a search with a query `cts:word-query("Hello")` with a user that has the `els-role-2` role. The query must know this new term to find the document. The query already knows the word "`Hello`" but how would it know the permission information in the protected path?

This is where the query rolesets are used. You configure query rolesets (with just `els-role-2` in this example) and then the query compares that query roleset with the caller's role. If the caller's role "matches" the query rolesets, the query will combine that information with the word "`Hello`" to generate the term, which matches the term put into the universal index by MarkLogic Server.

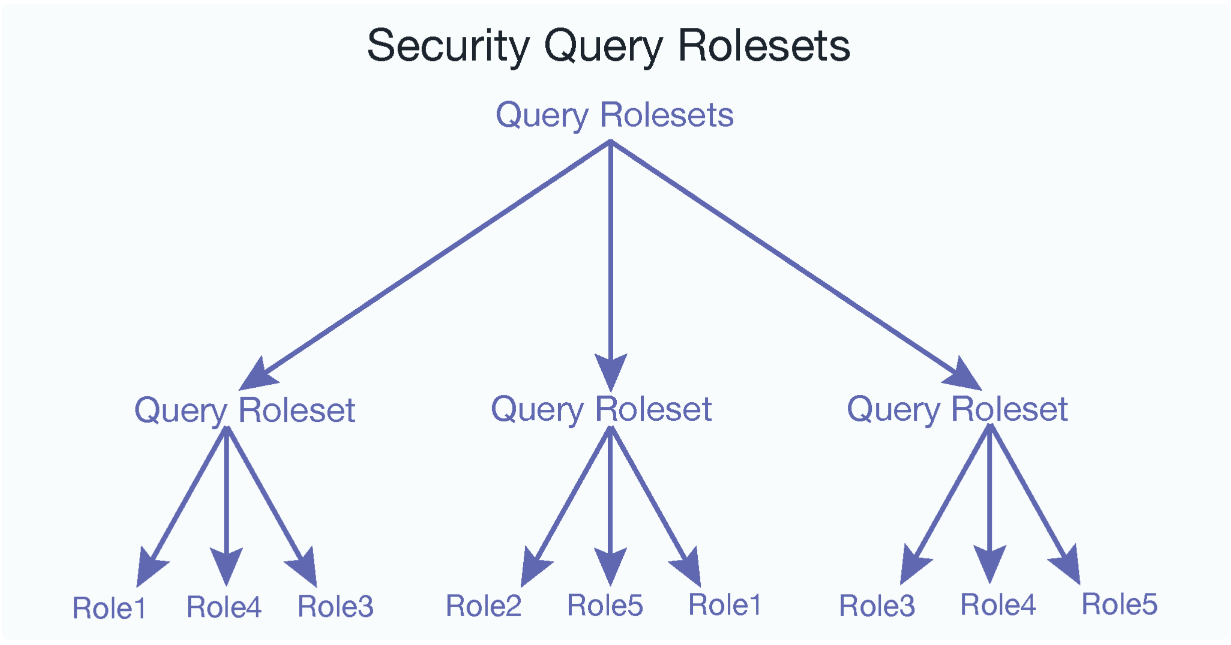
There are three ways to configure query rolesets:

- Use `xdmp:database-node-query-rolesets()` for documents with protected paths that are already in MarkLogic Server. See Helper Functions for Query Rolesets for information.
- Use `xdmp:node-query-rolesets()` to configure query rolesets as documents are being loaded into MarkLogic Server. See Helper Functions for Query Rolesets for information.
- Use `sec:add-query-rolesets()` to manually create the query rolesets on a case-by-case basis.

This last method of manually creating query rolesets works for simple examples and cases where there are not many protected paths. If you have a single protected path that matches an element like one in the examples above (with no overlaps), use a simple rule to create the query roleset in the Admin Interface. See Add Protected Paths and Query Rolesets for details.

The two helper functions; `xdmp:database-node-query-rolesets()` and `xdmp:node-query-rolesets()`, can help with configuring more complex query rolesets, either for documents already stored in MarkLogic Server or while documents are being added. MarkLogic Server leaves query rolesets configuration (creating and inserting the query rolesets into the Security database) to the administrator.

Query rolesets are made up of roles. There can be any number of roles in a roleset, as long as there are no duplicates. There can be multiple query rolesets in a database:



Query rolesets are required for element level security to work. You may ask why not just get the query rolesets information automatically from the protected paths when you configure `sec:protect-path()` to avoid the manual configuration of query rolesets. For this simple example this seems practical, but in the real world it is not uncommon to have multiple protected paths that match the same node or element. Some use cases will have 1000s of protected paths but only 100s of query rolesets. The indexer side of MarkLogic Server often needs to combine multiple query rolesets to create the term.

There is no way for the query side to derive that information from the protected path configuration, since whether a node element matches a protected path is based on the “value” of the node. And the query side doesn’t know the value of a node. There is no way for the query side to know what subsets of all the configured protected paths need to be taken into consideration when creating the query term. Since enumerating all possible combinations of the roles used in all protected paths is not practical, MarkLogic Server leaves query rolesets configuration (creating and inserting the query rolesets into the Security database) to the administrator.

Parent/Child Relationships in Query Rolesets

You might have a document where one user has permissions for an element that is the child of a parent element, for which that user does not have permissions. For example, there might be a simple document like this:

```
<root>
 <content>Contents of document
  <top-secret>Only role with "top-secret" can read this
    <secret>Only role with "secret" can read this</secret>
  </top-secret>
 Unclassified content
 </content>
</root>
```

This document might have these protected paths:

```
sec:protect-path("secret", (), (xdmp:permission("els-role-2", "read"))),
sec:protect-path("top-secret", (), (xdmp:permission("els-role-1", "read")))
```

A user with permissions on only the protected path for “secret” can’t see “secret” content unless the user also had permissions for the protected path for “top-secret” because the “secret” node is a child of the “top-secret” parent node.

Overlapping Protected Paths

Consider a more complex case with multiple paths matching the same node. Suppose you have a document like this:

```
<root>
  <foo a=1 b=2 c=3>Hello</foo>
</root>
```

It is possible to define three different protected paths that all match the `foo` element, overlapping each other:

```
sec:protect-path("/root/foo[@a=1]", (), (xdmp:permission("els-role-1", "read")))
sec:protect-path("/root/foo[@b=2]", (), (xdmp:permission("els-role-2", "read")))
sec:protect-path("/root/foo[@c=3]", (), (xdmp:permission("els-role-3", "read")))
```

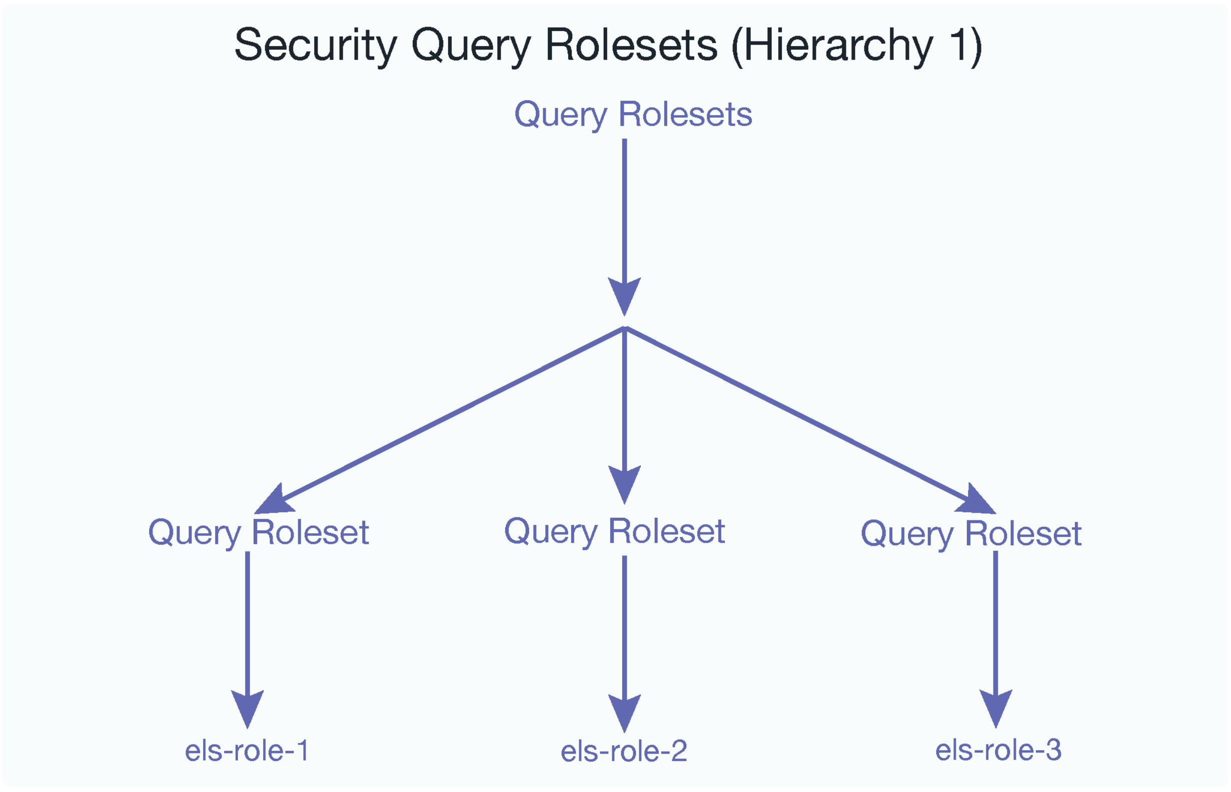
MarkLogic Server will still create just one term for “`Hello`”, which is the combination of the word and the query rolesets ((“`els-role-1`”),(“`els-role-2`”),(“`els-role-3`”)).

As a side note, in the above example the query rolesets is ((“`els-role-1`”),(“`els-role-2`”),(“`els-role-3`”)), which is different from simply (“`els-role-1`”,“`els-role-2`”,“`els-role-3`”).

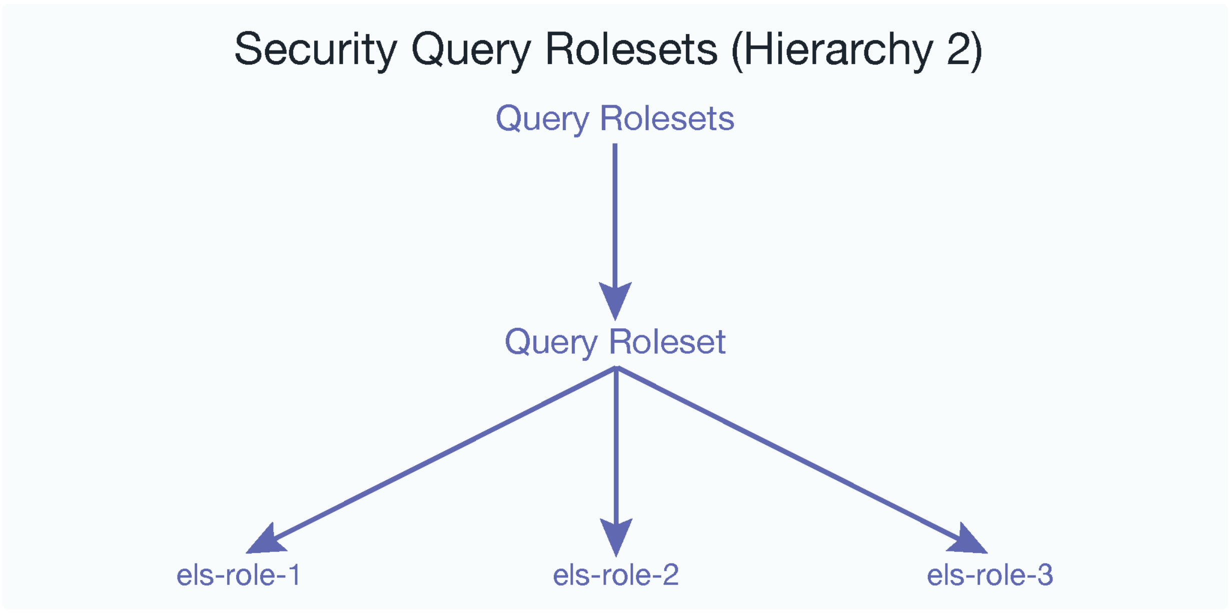
NOTE

In MarkLogic Server 9.0-2 query rolesets have been simplified and optimized. Existing documents with query rolesets configured in 9.0-1 will still be protected in 9.0-2. To take advantage of the optimization however, you need to reindex your documents and regenerate your query rolesets using the helper functions (APIs for Element Level Security). It is *highly recommended* that you reindex any protected documents already in your database and regenerate your query rolesets, since documents may be reindexed by another operation, which may cause a mismatch between the documents and the query rolesets. See Algorithm That Determines Which Query Rolesets to Use for examples and more details.

This is what the query rolesets hierarchy looks like for ((“`els-role-1`”),(“`els-role-2`”),(“`els-role-3`”)); three query rolesets and three roles:



This is what the query rolesets hierarchy looks like for (“`els-role-1`”,“`els-role-2`”,“`els-role-3`”); one query roleset and three roles:



If you only have one protected path that matches `foo` in the above example but with three roles, like this:

```
sec:protect-path("//foo", (), (
xdmp:permission("els-role-1", "read"),
xdmp:permission("els-role-2", "read"),
xdmp:permission("els-role-3", "read")))
```

Then (“`els-role-1`”,“`els-role-2`”,“`els-role-3`”) would be the proper query roleset to use. To configure the former ((“`els-role-1`”),(“`els-role-2`”),(“`els-role-3`”)), you would call:

```
(:run against the Security database :)
xquery version "1.0-ml";
import module namespace sec="http://marklogic.com/xdmp/security"
  at "/MarkLogic/security.xqy";

let $roleset1 := sec:query-roleset(("els-role-1"))
let $roleset2 := sec:query-roleset(("els-role-2"))
let $roleset3 := sec:query-roleset(("els-role-3"))
return sec:add-query-rolesets(sec:query-rolesets(($roleset1,$roleset2,$roleset3)))
```

To configure the latter (“`els-role-1`”,“`els-role-2`”,“`els-role-3`”), you can simply call:

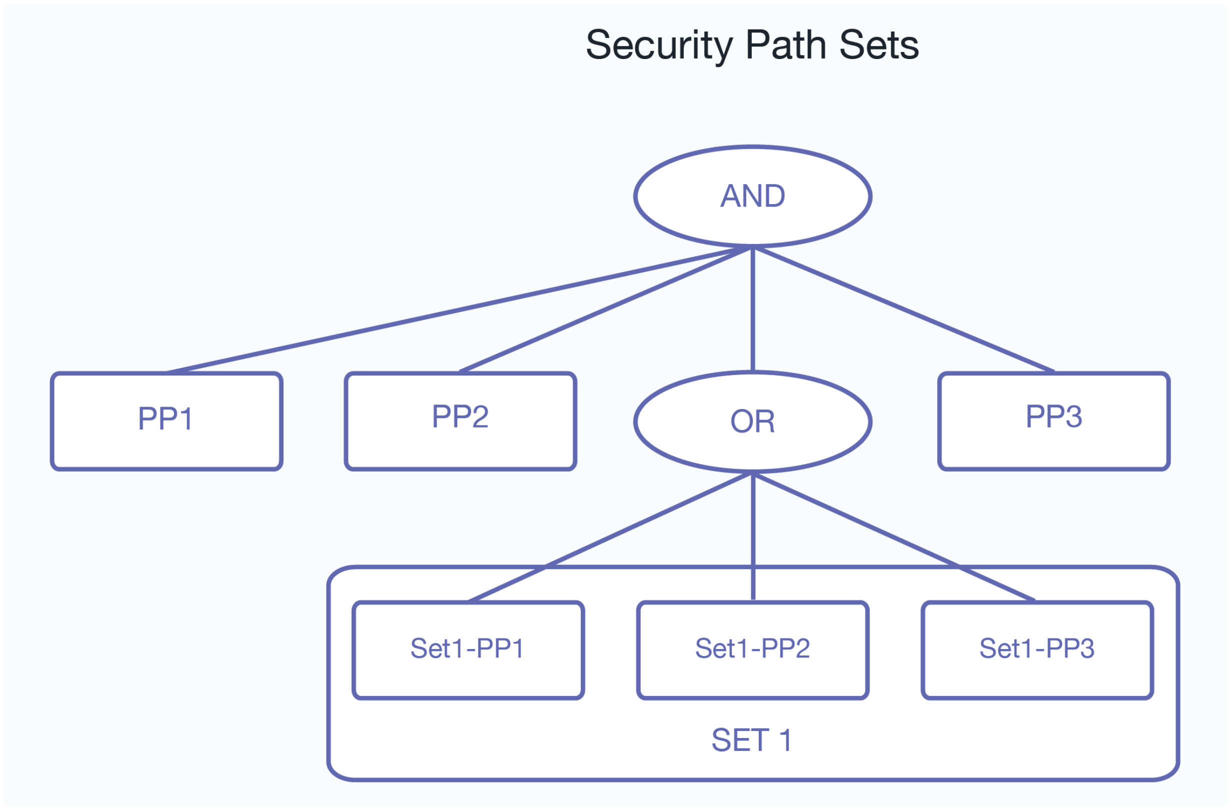
```
(:run against the Security database :)
xquery version "1.0-ml";
import module namespace sec="http://marklogic.com/xdmp/security"
  at "/MarkLogic/security.xqy";

let $roleset1 := sec:query-roleset(("els-role-1","els-role-2","els-role-3"))
return
sec:add-query-rolesets(sec:query-rolesets($roleset1))
```

When you are starting to configure and use element level security, the two query rolesets helper functions, `xdmp:database-node-query-rolesets()` and `xdmp:node-query-rolesets()` can simplify the process of setting up your query rolesets. These functions can be used for configuring query rolesets either for documents in the database, or for documents during ingestion. See Helper Functions for Query Rolesets for more information.

Protected Path Sets

A protected path set is a way to allow multiple protected paths covering the same element, with both AND and OR relationships between the permissions. This enables multiple arbitrary security marking for an element.

A protected path set is an optional string that represents the name of a set is associated with a protected path. A path that has no “set name” can be seen as a “degenerated form” of a set. The diagram below shows how permissions from paths in the same set are ORed, while permissions between sets are ANDed.



The set information (the name) is simply a “tag” on the protected path definition, not a separate document in the Security database.

Consider the following element:

```
<foo classification="TS" releasableTo="USA GBR AUS">
```

Using protected paths, MarkLogic Server element level security allows multiple protected paths covering the same element with an AND relationship among their permissions. This models a multiple security markings (for example `@classification` and `@releasableTo`) situation well. For the element above, two protected paths may be defined:

```
//foo[@classification="TS"] ("Role_TS", "read")
```

```
//foo[@releasableTo="USA GBR AUS"] (("Role_USA", "read"),
("Role_GBR","read"), ("Role_AUS","read"))
```

Note that the value of `@releasableTo` is a list of country codes, with each mapping to a role. A user with any of the “country roles” is allowed to read the element. The challenge is that a list can contain an arbitrary combination of country codes (total 200+). The above approach would require a user to define one protected path for each of the possible combinations, which may lead to a very large number of protected paths:

```
//foo[fn:contains(@releasableTo, "USA")] ("Role_USA", "read")
```

```
//foo[fn:contains(@releasableTo, "GBR")] ("Role_GBR", "read")
```

```
//foo[fn:contains(@releasableTo, "AUS")] ("Role_AUS", "read")
```

NOTE
Defining the preceding protected paths won't satisfy the requirement because the permissions among the paths are ANDed, not ORed.

The following example shows the benefit of the path set concept more clearly. Consider the following elements to be protected:

```
<foo classification="TS" releasableTo="USA">

<foo classification="TS" releasableTo="GBR">

<foo classification="TS" releasableTo="AUS">

<foo classification="TS" releasableTo="USA GBR">

<foo classification="TS" releasableTo="GBR AUS">

<foo classification="TS" releasableTo="USA AUS">

<foo classification="TS" releasableTo="USA GBR AUS">
```

Without using protected path sets, the following protected paths would need to be defined to protect the elements above:

```
//foo[@classification="TS"] ("Role_TS", "read")

//foo[@releasableTo="USA"] ("Role_USA", "read")

//foo[@releasableTo="GBR"] ("Role_GBR","read")

//foo[@releasableTo="AUS"] ("Role_AUS","read")

//foo[@releasableTo="USA GBR"] (("Role_USA", "read"), ("Role_GBR","read"))

//foo[@releasableTo="GBR AUS"] (("Role_GBR","read"), ("Role_AUS","read"))

//foo[@releasableTo="USA AUS"] (("Role_USA", "read"), ("Role_AUS","read"))

//foo[@releasableTo="USA GBR AUS"] (("Role_USA", "read"),
("Role_GBR","read"), ("Role_AUS","read"))
```

With protected path sets, only these protected paths are needed:

```
//foo[@classification="TS"] ("Role_TS", "read")

//foo[fn:contains(@releasableTo, "USA")] ("Role_USA", "read")
"SetReleasableTo"

//foo[fn:contains(@releasableTo, "GBR")] ("Role_GBR", "read")
"SetReleasableTo"

//foo[fn:contains(@releasableTo, "AUS")] ("Role_AUS", "read")
"SetReleasableTo"
```

The total number of protected paths required for the `@releasableTo` attribute is reduced from 7 to 3 using the `SetReleasableTo` protected path set.

In real world systems, the total number of possible country codes for these examples are more than 200, which leads to millions of possible combinations. So, with protected path sets, the number of required protected paths can be reduced from millions to just a couple of hundred for the `@releasableTo` use case.

Helper Functions for Query Rolesets

In order to search for query rolesets, you find out which query rolesets are configured for protected paths for a document already in the database. You can also discover if query rolesets are required for proper querying of a document being loaded into the database. Element level security includes two built-ins that can be used to discover existing protected paths in documents. The `xdmp:database-node-query-rolesets()` built-in is used for querying documents already in the database, while `xdmp:node-query-rolesets()` is used to query for protected paths in documents that are being loaded into the database. Given a node, these functions will return a list of the query rolesets for any protected paths, as long as the user of the built-ins has sufficient privileges and permissions. Usually, these function are called by an admin user.

For `xdmp:database-node-query-rolesets()`, the built-in returns a sequence of query rolesets that are required for proper querying of any given database nodes where element level security is in place on a document already in the database.

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

(: run this against the Security database :)

let $qry := 'xdmp:database-node-query-rolesets(fn:doc("/example.xml"), ("all"))'
let $qry-rolesets :=
xdmp:eval($qry, (),<options xmlns="xdmp:eval">
                   <database>{xdmp:database(YOUR_DB_NAME)}</database>
                 </options>)
return
sec:add-query-rolesets($qry-rolesets)

=>
<query-rolesets xml:lang="zxx" xmlns="http://marklogic.com/xdmp/security">
 <query-roleset>
  <role-id>12006351629398052509
  </role-id>
 </query-roleset>
</query-rolesets>
```

To find the name of this role ID, use this query in the Query Console:

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

sec:get-role-names((12006351629398052509))
=>
<sec:role-name xmlns:sec="http://marklogic.com/xdmp/security">els-role-2</sec:role-
name>
```

The `unconfigured` option for `xdmp:database-node-query-rolesets()` will return only those query rolesets that are not configured, meaning these query rolesets are not in the Security database yet (you have not configured them yet). The `all` option returns all query rolesets, even if they are already configured.

You can find existing or yet-to-be-configured query rolesets for documents being loaded into the database using `xdmp:node-query-rolesets()`. This built-in returns a sequence of query rolesets that are required for proper querying with element level security if the node is inserted into the database with the given document-insert options. This built-in also comes with the `unconfigured` option and the `all` option, and works the same as the `xdmp:database-node-query-rolesets()` built-in.

A typical workflow would call this function and add each query rolesets through the `sec:add-query-rolesets()` function before inserting the document into the database, so that the document can be correctly queried with element level security as soon as the document is inserted.

```
xdmp:node-query-rolesets(
    "/example.xml",
    <foo>aaa</foo>,
    <options xmlns="xdmp:document-insert">
      <permissions>
{xdmp:permission("role1","read"),xdmp:permission("role2","read")}
      </permissions>
    </options>)
```

To run this built-in you need to have the security role privileges.

Query for Protected Paths on a Document

You can use this XQuery code as a model to customize. The code sample searches for the protected paths associated with `foo.xml`.

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

declare function local:get-role-name($p as element(sec:permission)) {
  element sec:permission {
    $p/*,
    sec:get-role-names($p/sec:role-id)
  }
};

let $doc := xdmp:eval('fn:doc("foo.xml")', (), <options
xmlns="xdmp:eval"><database>{xdmp:database("Documents")}</database></options>)
for $p in fn:collection(sec:protected-paths-collection())/sec:protected-path
let $path :=
   xdmp:with-namespaces(
       for $ns in $p//sec:path-namespace
       return ($ns/sec:prefix/fn:string(.), $ns/sec:namespace-uri/fn:string(.)),
       xdmp:value("$doc" || $p/sec:path-expression/fn:string()))
return
  if (fn:exists($path)) then
    element sec:protected-path {
      $p/* except $p/sec:permissions,
      element sec:permissions {
        $p/sec:permissions/sec:permission ! local:get-role-name(.)
      }
    }
  else
    ()
```

You will only be able to see the protected paths for elements that you as the user would have permission to see. For example, if you had `role1` and the protected path was associated with `role2`, `role1` would not be able to see those paths.

Related functionality is the `all-query-rolesets-fragment-count` element returned from `xdmp:forest-counts()`. This number tells the caller how many fragments are indexed with a certain query-rolesets. If the number is 0 (across all databases), then `query-rolesets` is no longer in use.

6.4. Configure Element Level Security in the Admin Interface

Protected paths and query rolesets for element level security can be configured from the Admin Interface. The steps to configure users and roles for element level security are the same as described in Create Roles and Create Users and Assign Roles. To test the examples, add the sample documents using Query Console, as described in Add the Documents.

6.4.1. Add a Protected Path

To add a protected path for element level security:

1. Click **Security** in the left tree menu.
2. Under **Security**, click **Protected Paths**.
3. Click the **Create** tab.
4. Enter the information for the protected path: the path expression, the prefix and namespace, and the role name and capabilities for the permissions.
5. Click **More Permissions** to add additional permissions to this protected path.
6. Click **OK** when you are done.

6.4.2. Add a Query Roleset

To add a query roleset for element level security, using the Admin Interface:

1. Click **Security** in the left tree menu.
2. Under **Security**, click **Query Rolesets**.
3. Click **Create**.
4. In the Query Roleset field, add the roles (`els-role-1` and `els-role-2`) for the query roleset, separated by commas.
5. Click **more items** to add additional comma-separated query rolesets.
6. Click **OK** when you are done.

NOTE
An administrator must define query rolesets.

6.5. Configure Element Level Security with XQuery

To configure element level security, you would follow the same series of steps that you used for the earlier example. (See Example—Compartment Security.)

- Set up roles.
- Create users and assign roles.
- Insert documents with permissions.
- Add the query rolesets to the Security database.
- Add protected paths for elements in documents, by inserting the protected paths into the Security database.

6.5.1. Using XQuery for Query Rolesets

Use the `xdmp:database-node-query-rolesets()` helper function with the `sec:add-query-rolesets()` command to set up query rolesets using XQuery.

For example:

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

(: run this against the Security database :)

let $qry := 'xdmp:database-node-query-rolesets(fn:doc("/example.xml"), ("all"))'
let $qry-rolesets :=
xdmp:eval($qry, (),<options xmlns="xdmp:eval">
                   <database>{xdmp:database('Documents')}</database>
                 </options>)
return
sec:add-query-rolesets($qry-rolesets)
```

To manually set up just a few query rolesets, use the `sec:add-query-rolesets()` command using XQuery.

```
(: add a few query rolesets => run against the Security database :)

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

let $roleset := sec:query-roleset("new-role")
return
sec:add-query-rolesets(sec:query-rolesets(($roleset))
```

6.5.2. Using XQuery for Protected Paths

Use the `sec:protect-path()` command to set up your protected paths.

For example:

```
(: add protected paths -> run against the Security database :)

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

sec:protect-path("secret", (), (xdmp:permission("els-role-2", "read"))),
sec:protect-path("top-secret", (), (xdmp:permission("els-role-1", "read")))
```

This example uses a second parameter to set a protected path on the example path namespace.

```
(: add protected paths -> run against the Security database :)

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

declare namespace ex = "http://marklogic.com/example";

let $role := "executive"
return
  sec:protect-path(
    "/ex:envelope/ex:instance/employee/salary",
    (let $prefix := "ex",$namespace-uri :=
      "marklogic.com/example"
    return
    sec:security-path-namespace($prefix, $namespace-uri),
  (xdmp:permission($role, "read"))
  )
```

6.6. Configure Element Level Security with REST

You can also use the REST Management APIs to configure element level security. The REST properties endpoint is available to create query rolesets and protected paths:

`GET:/manage/v2/security/properties`

6.6.1. Using REST for Query Rolesets

The following XML and JSON examples show what is returned from `GET` (or used as payload to `PUT`) when using REST for query rolesets.

This example uses a `GET` with the response payload in XML:

```
$ curl -GET --anyauth -u admin:admin
  -H "Accept:application/xml,Content-Type:application/xml"
  http://localhost:8002/manage/v2/security/properties
```

This returns

```
<security-properties xmlns="http://marklogic.com/manage">
    <query-rolesets>
      <query-roleset>
             <role>432432534053458236326</role>
             <role>454643243253405823326</role>
      </query-roleset>
      <query-roleset>
             <role>124325333458236346123</role>
             <role>124233432432534058213</role>
      </query-roleset>
     </query-rolesets>
</security-properties>
```

Here is the same example with a JSON response payload:

```
$ curl -GET --anyauth -u admin:admin
  -H "Accept:application/json,Content-Type:application/json"
  GET:/manage/v2/security/properties
```

This returns

```
{
    "queryRoleset": [
          [
               432232321212123100000,
               432432534053458200000
          ],
          [
               124325333458236346123,
               124233432432534058213
          ]
     ]
}
```

NOTE

The REST Management APIs will accept both role names and role IDs in configuring query rolesets with `PUT`.

The following are example payloads for `POST` or `PUT` calls for managing query rolesets.

JSON:

```
{
  "role-name": ["manage-admin","rest-writer"]
}
```

XML:

```
<query-roleset-properties xmlns="http://marklogic.com/manage/query-roleset/properties">
  <query-roleset>
    <role-name>rest-reader</role-name>
  </query-roleset>
</query-roleset-properties>
```

6.6.2. Using REST for Protected Paths

The following XML and JSON examples show what is returned from GET (or used as payload to PUT) when using REST for query rolesets.

This example uses a GET with the response payload in XML:

```
$ curl -GET --anyauth -u admin:admin \
  -H "Accept:application/xml,Content-Type:application/xml" \
  http://localhost:8002/manage/v2/security/properties
```

This returns

```
<security-properties xmlns="http://marklogic.com/manage">
  <protected-paths>
<protect-path>
  <path-namespaces>
      <path-namespace>
          <prefix>ml</prefix>
          <namespace-uri>marklogic.com</namespace-uri>
      </path-namespace>
  </path-namespaces>
      <path-expression>/ml:foo/ml:bar</path-expression>
      <permissions>
        <permission>
          <role-name>user1</role-name>
          <capability>read</capability>
        </permission>
      </permissions>
    </protected-path>
  </protect-paths>
</security-properties>
```

Here is the same example with a JSON response payload:

```
$ curl -GET --anyauth -u admin:admin \
  -H "Accept:application/json,Content-Type:application/json" \
  http://localhost:8002/manage/v2/security/properties
```

This returns

```
"protected-path": [
{
  "path-namespace": [
      {
        "prefix" : "ml",
        "namespace-uri":  "marklogic.com"
      }
   ]
      "path-expression": "/some/path",
      "permissions": [
        {
          "role-name": "user1",
          "capability": "read"
        }
      ]
    }
  ]
}
```

NOTE

- When DELETE is used, a `force=true url param` will force deletion of “in use” protected paths.
- To specify an options element namespace in a JSON REST payload, you will need to define an `options-ns` key to set the namespace.

6.7. Combining Document and Element Level Permissions

This section describes how document level and element level permissions interact when both are applied to a document. At the element level `read`, `insert`, and `node-update` permissions can be used as part of the protected path definition.

NOTE

At the element level, the `update` and `node-update` capabilities are equivalent.

6.7.1. Document Level Security and Indexing

The document level security (document permissions with read capability) interacts with the element level security and affects

- The indexing of protected elements and whether index keys are combined with query rolesets.
- Whether protected embedded triples are indexed.

During indexing, the element level security of every node in the document is compared to the document’s protection. For a given node in the document, the permissions on every matching protected path are compared to the document’s permissions. When all matching protected paths are determined to be *weaker* than the document’s protection, the element’s protection is considered to be *weaker*. In this case, the query rolesets for the matching protected paths are **not** used when indexing the current node. An embedded triple with weaker protection on all of its nodes (subject, predicate and object), is extracted.

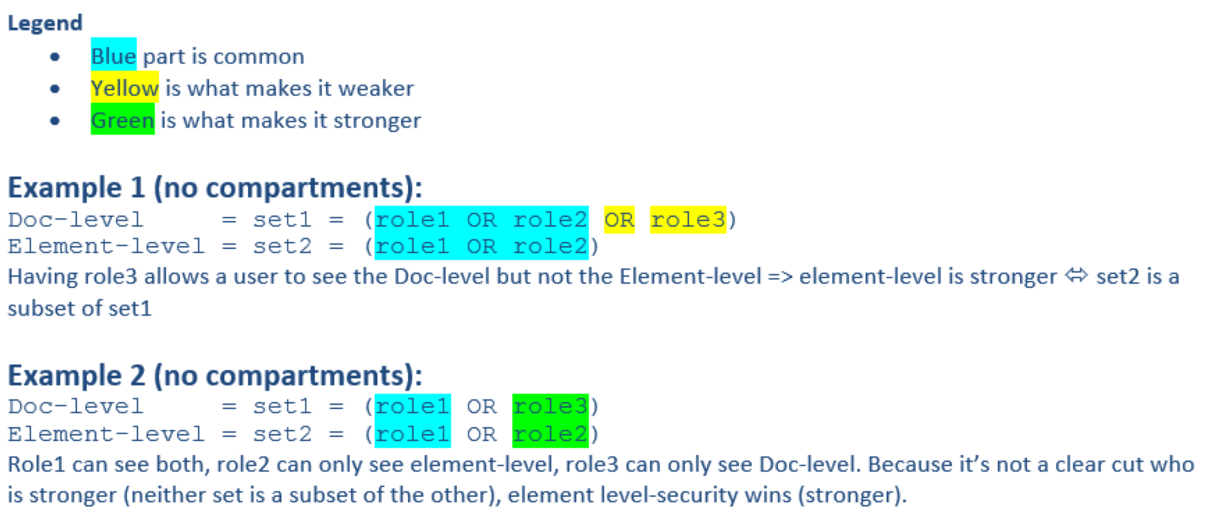
How is the element level protection determined to be weaker? In the absence of compartment security, a higher number of roles implies weaker permission because it means more accessibility. More roles in this case doesn't mean the total number of roles. It means that one set of roles is a superset of the other. The smaller set (the subset) is considered stronger because it is more restrictive. Roles are ORed by default. If the document is permitted to be accessed by more roles than the element (the element is more restrictive because there are more limitations on access), then the element security is considered to be *stronger* than the document security. In such a case, the element security is given higher precedence, and the element is protected (that is, the element is more restrictive). The fewer the number of contained or embedded roles, the more restrictive the permissions.

In situations where neither is stronger or it is unclear whether the document security or element security is stronger, the element level is always considered stronger. Only "Read" capability is checked when comparing the document's permissions to the element's permissions.

Note that there is no "flattening" of roles (inheritance of permissions) with element level security. Using the helper functions described in APIs for Element Level Security can facilitate both discovering existing query rolesets and applying them as part of ingestion.

6.7.2. Combination Security Example

More roles does not mean the total number of roles. It means that one set of roles is a superset of the other. The smaller set of roles is considered stronger. Consider the following examples:



Legend

- Blue part is common
- Yellow is what makes it weaker
- Green is what makes it stronger

Example 1 (no compartments):

```
Doc-level     = set1 = (role1 OR role2 OR role3)
Element-level = set2 = (role1 OR role2)
```

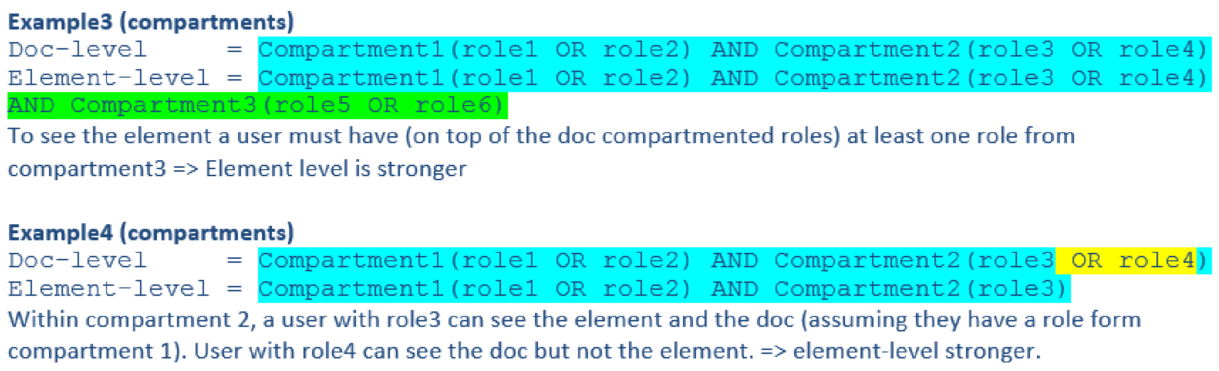
Having role3 allows a user to see the Doc-level but not the Element-level => element-level is stronger ⇔ set2 is a subset of set1

Example 2 (no compartments):

```
Doc-level     = set1 = (role1 OR role3)
Element-level = set2 = (role1 OR role2)
```

Role1 can see both, role2 can only see element-level, role3 can only see Doc-level. Because it's not a clear cut who is stronger (neither set is a subset of the other), element level-security wins (stronger).

Note that in example 1, element level protection is more restrictive than the document level protection. With compartment security, it's more complicated. The security level that has the most compartments wins, because more compartments means that access is more restrictive.



Example3 (compartments)

```
Doc-level     = Compartment1(role1 OR role2) AND Compartment2(role3 OR role4)
Element-level = Compartment1(role1 OR role2) AND Compartment2(role3 OR role4)
AND Compartment3(role5 OR role6)
```

To see the element a user must have (on top of the doc compartmented roles) at least one role from compartment3 => Element level is stronger

Example4 (compartments)

```
Doc-level     = Compartment1(role1 OR role2) AND Compartment2(role3 OR role4)
Element-level = Compartment1(role1 OR role2) AND Compartment2(role3)
```

Within compartment 2, a user with role3 can see the element and the doc (assuming they have a role form compartment 1). User with role4 can see the doc but not the element. => element-level stronger.

When element security is weaker than the document security, MarkLogic Server will index the content based on the document level security. MarkLogic Server lets the document level security protect it.

If the element is considered stronger, then content won't be visible without the correct query rolesets. If the element is weaker, then MarkLogic Server will return the element as part of a query (with the correct document level permissions).

6.8. Node Update Capabilities

Node update capabilities allow you to update document content by node. At the document level `xdmp:document-delete()` and `xdmp:document-insert()` can still be used if you have `update` capabilities, but `node-update` provides a finer control when combined with element level security. The `node-update` capability exists at the document level *and* at the element level. At the document level, if you have the `node-update` capability you can call `xdmp:node-replace()` and `xdmp:node-delete()` to modify nodes in a document, but not `xdmp:document-delete()` or `xdmp:document-insert()`. All of the node update built-ins take element level permissions into consideration.

Note that `node-update`, just like `insert`, can be seen as a *subset* of `update`, meaning that if a role has the `update` capability, it *automatically* gets the `node-update` capability as well.

If you have the `update` capability at the document level, you can call `xdmp:document-insert()`, `xdmp:document-delete()`, and *all* node-update functions. When you have the `update` capability at the document level, the element level security for `update` will not be checked, it is effectively "turned off". If you have the `node-update` capability, you can only call all `node-update` functions for that node.

6.8.1. Updates with Element Level Security

You can update content in documents when protected paths have been defined with element level security. Both document level and element level permissions will apply to the content (compartment level permissions may apply as well - see Interactions with Compartment Security for details). With the appropriate permissions, you can use insert and node-update at the element level to modify content containing protected paths. These capabilities take all element level permissions into consideration.

You can also protect document property nodes with element level security. With the `node-update/insert` capability, you can call `xdmp:document-add-properties()`, `xdmp:document-remove-properties()`, `xdmp:document-set-property()`, or `xdmp:document-set-properties()`. See Document and Element Level Permissions Summary for details.

6.8.2. Node Update and Node Insert at the Element Level

The `node-update` capability at the element level enables to you replace and delete nodes with `xdmp:node-replace()` and `xdmp:node-delete()`. The `insert` capability enables you to call `xdmp:node-insert-before()`, `xdmp:node-insert-after()`, and `xdmp:node-insert-child()`.

NOTE

At the element level, the `update` and `node-update` capabilities are equivalent.

Here are some simple examples using the `xdmp:node-insert-before()`, `xdmp:insert-node-after()`, and `xdmp:node-replace()` functions at the element level. These examples assume that both roles have document `insert/node-update` permissions as well as `read` permissions for the document and that the query rolesets are configured correctly.

Say that you have a document with these nodes:

```
<root>
  <foo>hello</foo>
  <bar>World</bar>
</root>
```

There are two roles; `role1` with both `read` and `update` permissions on the `<foo>` node, and `role2` with `read` and `node-insert` permissions on the `<root>` node:

```
<foo>,("role1", "read"),("role2", "read"),("role1", "update")
<root>,("role1", "read"),("role2", "read"),("role2", "insert")
```

The protected paths look like this:

```
sec:protect-path("//foo", (), (
xdmp:permission("role1", "read"),("role1", "update"),"role2", "read"))
sec:protect-path("//root", (), (
xdmp:permission("role1", "read"),("role2", "read"),("role2", "insert"))
```

The insert and update permissions check the ancestors of a node as well. See Document and Element Level Permissions Summary for details.

```
(: insert a new document :)
xdmp:document-insert("/example.xml",
<root>
  <foo>hello</foo>
  <bar>World</bar>
</root>
(xdmp:permission("role1", "read"), xdmp:permission("role2", "read"),
xdmp:permission("role1", "node-update"),("role1", "insert"), xdmp:permission("role2",
"node-update"),("role2", "insert")));
```

As `role2`, use `xdmp:node-insert-before()` to add a node to the document:

```
(: add a baz node before the foo node :)
xdmp:node-insert-before(fn:doc("/example.xml")/root/foo,
    <baz>Greetings</baz>);
(: view the revised document :)
fn:doc("/example.xml")

=>
<root>
  <baz>Greetings</baz>
  <foo>hello</foo>
  <bar>World</bar>
</root>
```

As `role1` you can use `xdmp:node-replace()` to change the `<bar>` node.

```
xdmp:node-replace(doc("/example.xml")/root/foo,<foo>Hello</foo>));
doc("/example.xml");
fn:doc("/example.xml")
=>
<root>
  <baz>Greetings</baz>
  <foo>Hello</foo>
  <bar>World</bar>
</root>
```

If you are using a user other than `role1` to do these same operations, a permission denied exception will be thrown.

6.9. Document and Element Level Permissions Summary

This table describes the permissions required to add, remove, or modify content at the document and element level.

Function Signature	Document and Element Level Permissions
`xdmp:node-replace($old,$new)`	Document: `node-update` is required Element: `$old` and all its ancestors, as well as descendants are checked for `update`/`node-update`
`xdmp:node-delete($old)`	Document: `node-update` is required Element: `$old` and all its ancestors as well as descendants are checked for `update`/`node-update`
`xdmp:node-insert-before($sibling,$new)`	Document: `insert` is required Element: all ancestors of `$sibling` are checked for `insert`
`xdmp:node-insert-after($sibling,$new)`	Document: `insert` is required Element: all ancestors of `$sibling` are checked for `insert`
`xdmp:node-insert-child($parent,$new)`	Document: `insert` is required Element `$parent` and all its ancestors are checked for `insert`
`xdmp:document-add-properties($uri,$props)`	Document: `node-update` is required Element: the properties root* is checked for `insert`
`xdmp:document-set-property($uri,$prop)`	Document: `node-update` is required Element: IF the property to be set doesn't exist, THEN the properties root is checked for `insert`; ELSE a.) the properties root* is checked for `update/node-update` b.) the property nodes) and all their descendants are checked for `update/node-update`
`xdmp:document-set-properties($uri,$props)`	Document: `node-update` is required Element: IF there is no properties fragment THEN the properties root is checked for `insert`; ELSE a.) the properties root* is checked for `update/node-update` b.) all existing property nodes and all their descendants are checked for `update/node-update`
`xdmp:document-remove-properties($uri,$property-names)`	Document: `node-update` is required Element: a.) the properties root* is checked for `update/node-update` b.) all property nodes to be removed and all their descendants are checked for `update/node-update`

* The properties root is the root of the properties node of a document, not the individual properties contained in the properties node. The properties root is the first line in this diagram:

```
<prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
   <prop1>. . .</prop1>
   <prop2>. . .</prop2>
   .
   .
   .
   <propN>. . .</propN>
</prop:properties>
```

See Interactions with Compartment Security for more about combining element level security with compartment security.

6.10. Node Update and Document Permissions Expanded

The examples in this section expand upon the interactions of element level security and document permissions.

6.10.1. Unexpected Behavior with Permissions

In this example, the role has the necessary document-level permissions. The example has to do with the element level, protected path permissions. Say you have a document (`example.xml`) with these nodes:

```
<foo>
  <bar>
</foo>
```

For this example, `role1` has both `read` and `update` permissions on the `<foo>` node, and `update` permissions on the `<bar>` node, but *no* `read` permissions on the `<bar>` node:

```
<foo>, ("role1", "read"), ("role1", "update")
<bar>, ("role1", "update")
```

It is assumed for these examples that all of the query rolesets are already configured correctly.

If `role1` calls this `xdmp:node-replace()` query:

```
xquery version "1.0-ml";

xdmp:node-replace(doc("/example.xml")/foo, <foo><baz>Hello</baz></foo>);
```

The query will succeed, because `role1` has update permissions on `/foo.`

If `role1` calls this `xdmp:node-replace()` query on `/bar`:

```
xquery version "1.0-ml";

xdmp:node-replace(doc("/example.xml")/foo/bar, <baz>Hello</baz>);
```

The expression `/foo/bar` will return an empty sequence because `role1` cannot read the bar element. Hence the `node-replace` call will effectively be a no-op, because `xdmp:node-replace()` was asked to replace nothing with something.

6.10.2. Different Permissions on the Same Node

Multiple roles can have different permissions on the same node. Some interactions between roles may be unexpected. For example, if you have a document with two nodes `<foo>` and `<bar>`. The `<bar>` node is a child of the `<foo>` node.

```
<foo>
  <bar>
```

You have two roles; `role1` with both `read` and `update` permissions on the `<foo>` node, and `role2` with read permissions on the `<bar>` node:

```
<foo>, ("role1", "read"), ("role1", "node-update")
<bar>, ("role2", "read")
```

NOTE

At the element level, the `update` and `node-update` functions are equivalent.

The protected paths for this document would look like this:

```
sec:protect-path("//foo", (), (
xdmp:permission("els-role-1", "read"),("role1", "node-update"))

sec:protect-path("//foo/bar", (), (
xdmp:permission("role2", "read"))
```

With these protected paths, `role1` cannot read the `<bar>` node. But because `role1` has update permissions on the parent node (`<foo>`), `role1` can overwrite the `<bar>` node, even though it cannot read it.

To prevent this, add node-update permissions to the `<bar>` node. The permissions would now look like this:

```
<foo>, ("role1", "read"), ("role1", "node-update")
<bar>, ("role2", "read"), ("role2", "node-update")
```

The presence of the "node-update" permission on the `<bar>` node prevents `role1` from being able to update and overwrite the `<bar>` node (the child node of the `<foo>` node).

This happens because node permissions are checked separately; first there's a check for protected paths for `read`. Then there is a check for protected paths for `update`. If no update is found for `/foo/bar`, then `role1` is allowed to update `<bar>`. If there is a protected path for updating `<bar>`, then `role1` is not allowed to update `<bar>`.

6.10.3. A More Complex Example

To expand even more on the node-update example with added document permissions, you could have roles with both protected paths and document permissions.

Say you have a document with these nodes:

```
<foo>
  <bar>
<baz>
```

At the document level, there are these permissions:

```
("role1", "read"), ("role1", "node-update")
("role2", "read"), ("role2", "node-update")
("role3", "read"), ("role3", "update")
```

At the element level, there are these permissions for protected paths:

```
<foo>, ("role1", "read"), ("role1", "node-update")
<bar>, ("role2", "read"), ("role2", "node-update")
```

In this example,

- `role1` cannot update (or override) `<bar>` because at the element level `role2` has `<bar>` protected path permissions.
- `role3` can override everything because at the document level it has `update` capability but can only read `<baz>` which has no protected paths.

6.11. APIs for Element Level Security

This section describes the element-level security APIs.

6.11.1. XQuery APIs

These built-in functions are available to help manage element level security:

- `sec:protect-path()`
- `sec:unprotect-path()`
- `sec:remove-path()`
- `sec:path-set-permissions()`
- `sec:path-add-permissions()`
- `sec:path-get-permissions()`
- `sec:path-remove-permissions()`
- `sec:query-rolesets-collection()`
- `sec:security-path-namespace()`
- `sec:query-roleset()`
- `sec:query-rolesets()`
- `sec:query-rolesets-id()`
- `sec:add-query-rolesets()`
- `sec:remove-query-rolesets()`
- `sec:protected-paths-collection()`

With the appropriate permissions, protected path content can be modified using these node update APIs:

- `xdmp:node-replace()`
- `xdmp:node-delete()`
- `xdmp:node-insert-after()`
- `xdmp:node-insert-before()`
- `xdmp:node-insert-child()`

These two helper functions can be used to search for protected paths:

- `xdmp:node-query-rolesets()`
- `xdmp:database-node-query-rolesets()`

6.11.2. REST Management APIs

The REST Management APIs provide the same functionality as the XQuery APIs covered in XQuery APIs for both protected paths and query rolesets.

REST Management APIs for Protected Paths

These REST Management APIs can be used for adding, modifying, or deleting protected paths.

`GET /manage/v2/protected-paths`

`POST /manage/v2/protected-paths`

`GET /manage/v2/protected-paths/{id|name}`

`DELETE/ manage/v2/protected-paths/{id|name}`

`GET /manage/v2/protected-paths/{id}/properties`

`PUT /manage/v2/protected-paths/{id}/properties`

REST Management APIs for Query Rolesets

These REST Management APIs are available for managing query rolesets:

`GET /manage/v2/query-rolesets`

`POST /manage/v2/query-rolesets`

`GET /manage/v2/query-rolesets/{id}`

```
DELETE /manage/v2/query-rolesets/{id}
```

```
GET /manage/v2/query-rolesets/{id}/properties
```

6.12. Algorithm That Determines Which Query Rolesets to Use

In MarkLogic Server 9.0-1, if the path permissions on a node are “weaker” (as defined in Document Level Security and Indexing) than the document level permissions or its parent node’s permissions, the path level permissions will be ignored as far as query rolesets definition is concerned.

NOTE
A child node will still inherit its parent’s query rolesets.

In MarkLogic Server 9.0-2, the set of query rolesets for a given node (after inheritance from ancestors) will be “compacted” based on the “weaker” permissions defined in Document Level Security and Indexing. If a query roleset in the set is “weaker” than any other query rolesets in the set, that “weaker” roleset will be “removed”.

For example:

Roles: `role-1`, `role-2`, `role-3`

Document:

```
<foo>Hello<bar>World</bar>,</foo>
```

with `((role-1, read), (role-2, read), (role-3, read))`

Protected Paths:

```
//foo (role-1, read), (role-2, read)
//bar (role-1, read)
```

In MarkLogic Server 9.0-1, the query rolesets for the “bar” node is ((`role-1`, `role-2`), (`role-1`)), but in 9.0-2 it is simplified (“compacted”) to ((`role-1`)).

NOTE
If any query roleset in the above set is “weaker” than the document level permissions, it will be omitted too.

Here is another example:

Roles: `role-1`, `role-2`, `role-3`

Document:

```
<foo><bar>Hello</bar></foo>
```

with `(role-1, read)`

Protected Paths:

```
/foo/bar (role-1, read), (role-2, read)
//bar (role-3, read)
```

In 9.0-1, the query rolesets for the “bar” node is ((`role-1`, `role-2`), (`role-3`)), but in 9.0-2 it is simplified (“compacted”) to ((`role-3`)) because (`role-1`, `role-2`) is “weaker” than the document level permissions.

6.13. Interactions with Compartment Security

You can add an extra level of protection to any content concealed by protected paths by using compartment security in conjunction with element level security. Compartment security adds a finer granularity of protection for content because a user must have the appropriate role *and* belong to the appropriate compartment to view the concealed content. For more about compartment security see Compartment Security.

A *compartment* is a name associated with a role. The compartment name is used as an additional check when determining a user’s authority to access, modify, or create documents. If compartment security is not used, permissions are checked using OR semantics. For example, if a document has `read` permissions for `role1` and `read` permissions for `role2`, without compartment security, a user who has *either* `role1` or `role2` can read that document.

If any permission on a document has a compartment, then the user must have that compartment in order to access any of the capabilities, even if the capability is not the one with the compartment. Access to a document requires a permission in each compartment for which there is a permission on the document, regardless of the capability of the permission. So, if there is read permission for role `compartment1`, there must also be an update permission for some role in `compartment1` (but not necessarily the same role).

If compartment security is used, then the permissions are checked using AND semantics for each compartment. If the document has compartment permissions for both `compartment1` and `compartment2`, a role must be associated with both compartments to view the document. If two roles have different compartments associated with them (for example `compartment1` and `compartment2`) , a user must have `role1` and `role2` access the document.

This is in addition to checking the OR semantics for each non-compartmented role, as well as a non-compartmented role that has a corresponding permission on the document. If compartment security is used along with element level security, a user must have both the appropriate compartment security and the appropriate role to view protected content.

Because element level security follows the same role-based authorization model, compartment security checks are be done in the same way at the element level. The only difference is that when calculating “compartments needed” at the element level, only those permissions with the capability being requested (for example “read”) are checked.

Here is an example using these three roles:

- `role0` (with no compartment)
- `role1` (with `compartment1`)
- `role2` (with `compartment2`)

These permissions have been set on the document:

`(role0, read)`, `(role1, read)`, and `(role2, update)`

With these permissions set on the *document*, a user with both `role1` and `role0` cannot perform a `read` operation. This is because one of the permissions mentions `role2`, even though it is not for `read`. In fact, with these permissions at the document level, no one (except for admin) would be able to read the document.

If the above permissions are set for an *element*, a user with both `role1` and `role0` will be able to read the element, because element level security checks `read`, `update`, and `insert` permissions separately, based on the operation requested.

NOTE
Permission checks at the document and element levels are performed independently.

6.13.1. Compartment Security and Indexing

Using more compartments means stronger security because compartments are ANDed. The roles within the same compartment are ORed. When a document or element is protected by more compartments, this implies stricter access. Roles without compartments are ORed amongst themselves and then ANDed with compartment roles. The general rules are:

- If an element is protected by more compartments than the document's, the element level protection is considered *stronger*.
- Within the same compartment, if the element is protected for fewer roles, the element level protection is *stronger*.
- There are situations where the weaker/stronger protection cannot be clearly determined. In this case, element level security is always considered to be *stronger*.

See Node Update and Document Permissions Expanded and Combination Security Example for more about security protection and indexing. For more information about compartment security, see Compartment Security.

6.14. Interactions with Other MarkLogic Server Features

The element level security feature is an index-level feature that is implemented in the universal index, the geospatial index, the bitemporal index, the range index, and the triple index. Features that use a single lexicon (values, elements, element values, sum-aggregation, and so forth.) will work with element level security, as well as operations that make use of the triple index.

Query operations that rely on the triple index (such as SPARQL, SQL, the new version of MarkLogic Server ODBC, and the Optic API) are supported by element level security. Element-level security can be leveraged by semantic graphs and SQL. In semantics, individual triples can be protected. In SQL, this allows you to enable column-level security by protecting specific columns in a Template (TDE). See Node Update and Document Permissions Expanded for details.

This section describes interactions with various MarkLogic Server features.

6.14.1. Lexicon Calls

For simple lexicons like values or words, this feature is similar to CTS queries (see Others). However, lexicon calls that involve co-occurrences will only work with unprotected values (range-index based SQL implementation has the same problem).

6.14.2. Fragmentation

The indexer in MarkLogic Server doesn't know the full path when working on child fragments of a parent document, because the indexer indexes the child fragments first before it indexes the parent. Because of this element level security and fragmentation don't work well together, although fragmentation will still work on documents that don't have any protected elements.

Any new document with matching fragmentation and protected elements will be rejected. Either an `XDMP-PARENTLINK` or an `XDMP-FRAGMENTPROTECTEDPATH` error will be thrown. When element level security and fragmentation both apply simultaneously to an existing document (already in the database), a reindexing error will be thrown, causing reindexing to stop. User must either remove/fix the matching element level security path or the matching fragmentation element.

For example, if a protected path that ends with `baz` is added (`/foo/bar/baz`) and if a fragment root is configured for `baz`, any document containing node `baz` (even under a different path `/A/B/C/baz`) will error out with `XDMP-PARENTLINK` when the document is inserted or reindexed.

6.14.3. SQL on Range-Index Based Views

SQL that is based on Range-Index views will only work with values that are not protected by element level security.

6.14.4. UDFs (Including UDF-Based Aggregate Built-ins)

UDFs that operate on a single range index will work with element level security. This includes the most commonly used aggregate functions like `cts:sum-aggregate()`, `cts:stddev()`, and so on. UDFs that apply to more than one range index will only work with unprotected values.

6.14.5. Reverse Indexes

Similar to the case for triples (see SPARQL), if an element that contains a `cts:query()` matches a protected path of *any role*, or any part of the `cts:query()` matches *any role*, the query won't be added into the reverse index unless the document's security is stronger than the element security on the element. See Node Update and Document Permissions Expanded for details. A `cts:reverse-query()` that would normally find a document containing a matching `cts:query()` will no longer match once the embedded `cts:query()` (or its children) is protected by element level security that is stronger than the document's security.

6.14.6. SPARQL

If a `sem:triple()` is inside an element that is concealed for any role and the element level security is stronger than the document security, it will not be put into the triple index. If the triple itself or its subject, predicate, or object is protected, it will not be put into the triple index, unless the document security is stronger than the element level security protection. In some scenarios, where the document's security is *stronger* than the element security on a triple, the protected triple will be added to the triple index. This is because the document's security already covers the protected element. The information will be protected at the document level. See Node Update and Document Permissions Expanded for details.

6.14.7. Alerting and QBFR

Each target in a QBFR (Query Based Flexible Replication) configuration is associated with a user and a query. A target can only get documents that match the query and that the user is allowed to access. In QBFR, some flows must use the privileged user to run queries because the process needs to figure out what documents will be deleted from a target. Internally, alerting uses reverse queries to determine the set of matching rules for a given document or node. The matching rules are then used to trigger the appropriate action for the target user of each matching rule.

There is a two-pass rule matching approach; first the rule matching runs against the full version of the document, then for each matching rule, a second match test is performed using the version of the document that the target user of the rule is allowed to see.

Now, a rule that matches “hello” will not trigger the action if the target user cannot see “hello” due to element level security protection. Using element level security, MarkLogic Server will deliver a redacted version of the document, based on element level security configuration of protected paths and the user's role.

NOTE
When using element level security with Alerting and QBFR, if a query contains a “NOT” clause, you may see false negatives. What this means is documents might not be replicated when the alerting rule contains a `cts:not-query()` due to the false negatives.

6.14.8. mlcp

When you use mlcp to ingest files from MarkLogic Server 9 or later to another MarkLogic Server 9 or later instance, the protected paths and `node-update` permissions will be preserved.

6.14.9. XCC

If you use XCC to insert a document with the `node-update` permission into MarkLogic Server 8.0-6 or earlier, the behavior is undefined.

If you use XCC to insert a document with the `node-update` permission into MarkLogic Server 8.0-7 or a later version of MarkLogic Server 8, the `node-update` permission is silently discarded.

These restrictions apply to using `Session.insertContent` with a `Content` object whose `ContentCreateOptions` include the `ContentCapability.NODE_UPDATE` capability.

6.14.10. Bitemporal

Do not protect system axis for bitemporal queries when using element level security.

6.14.11. Others

A key concept to support CTS queries with element level security is query rolesets. A query roleset is simply a list of roles. When indexing, MarkLogic Server takes query roleset information into consideration and essentially “partitions” indexes based on query rolesets. All queries (except for composite ones like `and-query`) will look into indexes for different query rolesets based on the caller’s role and logically “OR” the results. See Query Rolesets for more about query rolesets.

There are special rules for CTS queries, phrase breaks, field values, geo element pairs, auditing and term-queries when the elements involved are protected.

- CTS queries - Positions are always calculated based on the original (full) document, prior to any concealing. This implies that the distances calculated based on indexes will be larger than what appears in the concealed document.
- Phrase breaks - When indexing, any element that is protected is considered a phrase break. Consider this example: `<foo>1<bar>2 3</bar>4</foo>`. If “bar” is protected by any protected path, then it is considered a phrase break regardless of whether a phrase through is defined on it. So, in the example, “2 3” is still a phrase, but “1 2” or “3 4” is not. “1 4” is not a phrase either.
- Fields - For an XML document, field values or field range values are sometimes calculated by concatenating elements included in the field. If those elements don’t have the same rolesets (permissions), concatenating can cause leaking of information. MarkLogic Server will treat this as a misconfiguration and log a warning. The query result on such a field is undefined.
- Geo element pair with inconsistent permissions - Similar to the field case above, if permissions on the two elements (or JSON properties) of the geo pair are not consistent (or either of the two elements has different permissions from the parent node), MarkLogic Server will treat it as a misconfiguration and log a warning. The query result is undefined in this case.
- Auditing -

For the “document-read” event, if the node involved has any element concealed, the string “concealed” will be reported in the event. Here is an example:

```
2016-10-18 15:45:29.886 event=document-read; type=concealed; uri=foo.json;
database=Documents; success=true;
```

When a node or properties update built-in call is rejected due to the lack of element-level permissions, the “no-permission” event will be reported. This is very similar to how the event is used when such a call is rejected due to the lack of document-level permissions.

- term-query - Element level security won’t prevent a “malicious” user from getting a term key through `xdmp:plan()` from a different MarkLogic Server deployment, then passing that to a `cts:term-query` to find out information she is not supposed to see on the current MarkLogic Server deployment. The solution is to add a new execute privilege “term-query” to “protect” `cts:term-query`. For backward compatibility, this privilege will only be checked when element level security is in use (for example, when at least one protected path is configured).

6.14.12. Rolling Upgrades

For rolling upgrades, configuration and Admin Interface API calls will throw an error when a rolling upgrade (from a release that does not support element level security) has not yet completed and been committed across the cluster. Document inserts (or set-permissions) with the new node-update capability will be rejected if the effective version is not 9.0-1 or above.

7. Protecting XQuery and JavaScript Functions with Privileges

Execute privileges provide authorization control for executing XQuery and JavaScript functions. MarkLogic Server provides three ways to protect XQuery functions:

- Built-in execute privileges, created by MarkLogic Server, control access to protected functions such as `xdmp:document-load()`.
- Custom execute privileges, which you create using the Admin Interface or the security function in the `security.xqy` module, control access to functions you write.
- Amps, which temporarily amplify a user's authority by granting the authority to execute a single, specific function. You can only amp a function in a library module that is stored in the MarkLogic Server modules database.

This section describes how to use these protections.

7.1. Built-In MarkLogic Server Execute Privileges

Every installation of MarkLogic Server includes a set of pre-defined execute privileges. You can view this list either in the Admin Interface or in Appendix B: Pre-defined Execute Privileges of *Administrating MarkLogic Server*.

7.2. Protecting Your XQuery and JavaScript Code with Execute Privileges

To protect the execution of an individual XQuery or JavaScript function that you have written, you can use an *execute privilege*. When a function is protected with an execute privilege, a user must have that specific privilege to run the protected XQuery or JavaScript function.

NOTE
Execute privileges operate at the function level. To protect an entire XQuery or JavaScript document that is stored in a modules database, you can use execute permissions. For details, see Document Permissions.

This section describes how to protect your code with execute privileges.

7.2.1. Using Execute Privileges

The basic steps for using execute privileges are:

- 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.

To test for a privilege, use `xdmp:security-assert()` (XQuery) or `xdmp.securityAssert()` (JavaScript). This function tests to determine 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 can catch and handle.

For example, to create an execute privilege to control the access to an XQuery function called `display-salary`, use the following steps:

1. Use the Admin Interface to create an execute privilege named `allow-display-salary`.
2. Assign any URI (for example, `http://my/privs/allow-display-salary`) to the execute privilege.
3. Assign a role to the privilege. You may want to create a specific role for this privilege depending on your security requirements.
4. Finally, in your `display-salary()` XQuery function, include an `xdmp.securityAssert()` call to test for the `allow-display-salary` execute privilege as follows:

```
xquery version "1.0-ml";
declare function display-salary (
     $employee-id as xs:unsignedLong)
as xs:decimal
{
xdmp:security-assert("http://my/privs/allow-display-salary", "execute"),
...
} ;
```

7.2.2. Execute Privileges and App Servers

You can also control access to specific HTTP, WebDAV, ODBC, or XDBC servers using an execute privilege. Using the Admin Interface, you can specify that a privilege is 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.

7.2.3. Creating and Updating Collections

To create or update a document and add it to a collection, the `unprotected-collections` privilege is required. You also need a role corresponding to an insert or update permission on the document. For a *protected collection* (a protected collection is created using the Admin Interface), 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 `xdmp:document-load()`, `xdmp:document-insert()`, and `xdmp:document-add-collections()` in the *XQuery/XSLT Functions by Name* reference.

7.3. Temporarily Increasing Privileges with Amps

Amps provide users with additional authorization to execute a specific function. Assigning the user this authorization permanently could compromise the security of the system. When executing an *amped* function, the user is part of an amped role, which temporarily grants the user additional privileges and permissions of that role. 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. If the user does not have read permissions on all the documents in the database, queries run by the user do not “see” all the documents in the database. If you want anyone to be able to know how many documents are in the database, regardless of whether they have permissions to see those documents, you can create a function named `document-count()` and use an amp on the function to elevate the user to a role with read permission for all documents. When the user executes the amped function, she temporarily has the necessary read permissions that enable the function to complete accurately. The administrator has in effect decided that, in the context of that `document-count()` function, it is safe to let anyone execute it.

Amps are security objects that you use the Admin Interface or Management API to create. Amps are specific to a single function in a library module, which you specify by URI and local name when creating the amp. You can only amp a function that resides in a library module that is stored in a trusted directory on the filesystem, such as in the `Modules` directory (`<install_dir>/Modules`), or in the modules database configured for the server in which the function is executed. The recommended best practice is to put your library module code into the modules database. You cannot amp functions in XQuery modules or JavaScript modules stored in other locations. For example, you cannot amp a function in a module installed under the filesystem root of an HTTP server, and you cannot amp functions that reside in a main module. Functions must reside in the `Modules` database or in the `Modules` directory because these locations are trusted. Allowing amped functions from under a server root or from functions submitted by a client could compromise security. For details on creating amps, see Security Administration in *Administrating MarkLogic Server*.

For an example that uses an amp, see Access Control Based on Client IP Address. For details on amps in JavaScript modules, see Amps and the module.amp Function in the *JavaScript Reference Guide*.

8. Query-Based Access Control

Query-based access control (QBAC) is a mechanism to provide policy enforcement for access to resources based on security markings, metadata, or data in the records themselves. It works by associating queries with roles and users, adding them automatically to security queries to constrain access. It integrates with the existing MarkLogic Server security model, which is a role-based security model.

NOTE
Advanced Security License option is required when using Query-Based Access Control.

8.1. What is QBAC?

Prior to the addition of this feature, a secure data access query is formed solely based on permissions from the effective user roles. QBAC augments this security query with more general CTS queries to provide more flexible data access rules. These queries are associated with roles and users and are added to the security queries to constrain and check access permissions. This allows you to define access policies based on document contents or metadata, and to change those policies without re-processing the document permissions, and without having to write triggers or code to monitor when document contents change.

There are two types of QBAC queries: queries on roles and queries on users. Queries on roles are *definitional*: a document that passes the role query is treated as having the corresponding permission for that role, so a user with that role may also see the document. Queries on users are *restrictive*: the user may only see the documents that pass the query. When the server checks queries, the queries on uncompartmented roles are ORed, and the queries on users are ANDed. So, queries on roles expand the scope what is authorized for that role, while queries on users restrict the scope of what is authorized.

Secure data access at the fundamental level in MarkLogic Server is constrained by a security query. Unsecured data access is used only for the admin user or for certain internal lookups or fetches. All user-facing APIs that access data stored in the database are secured in this fashion, whether using a cts:search, a lexicon call such as cts:values, a SQL or SPARQL query that accesses triples, an update operation such as xdmp:node-replace, or the execution of a module.

As a result, QBAC can integrate with all the existing MarkLogic Server security features, such as Compartment Security, Element Level Security (ELS), triples and protected collections. For example, when a path is protected by ELS, QBAC will not leak information about the contents. However, extra care should be taken when setting up security model combining both queries and other security features.

NOTE
Users with QBAC document access are not able to read document properties. This is a design limitation. Users with QBAC document access do not have properties access by default, unless the QBAC query explicitly matches document properties through a CTS query. However, QBAC access to document properties gives access to the document itself by default.

8.2. Example QBAC Applications

This section describes several scenarios that use QBAC. They are not meant to demonstrate the correct way to set up QBAC, as your situation is likely to be unique. However, it demonstrates how QBAC works and may give you some ideas on how to implement your own security model.

8.2.1. Scenario 1: Region Restrictions

Description: A security architect Sammy from company ABC wishes to enforce up a policy that people in each of the regions can see documents relevant to their region by inspecting metadata in each document to determine if someone can access it.

To setup QBAC for this scenario, you need to create the necessary roles and users, and insert documents through Query Console or REST Management APIs.

To run through the example, follow the steps in each subsection.

Create Roles

Sammy sets up some roles: `region-APAC`, `region-EMEA`, and `region-NA`.

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

sec:create-role("can-read", "General read", (), (), ()),
sec:create-role("region-APAC", "Can see APAC documents.",
   (), (), (), (), (),
   map:map()=>map:with(
     "read", cts:element-query(xs:QName("metadata"), cts:element-word-
query(xs:QName("region"), "APAC")))
),
sec:create-role("region-EMEA", "Can see EMEA documents.",
   (  ), (), (), (), (),
   map:map()=>map:with(
     "read", cts:element-query(xs:QName("metadata"), cts:element-word-
query(xs:QName("region"), "EMEA")))
),
sec:create-role("region-NA", "Can see NA documents.",
   (), (), (), (), (),
   map:map()=>map:with(
     "read", cts:element-query(xs:QName("metadata"), cts:element-word-
query(xs:QName("region"), "NA")))
)
```

Create Users

Using the Admin Interface > Security > Users > Create, Query Console, or REST Management APIs (RMAs), Sammy creates the users and assign them the roles indicated in the following table:

User	Roles
Edna	`region-NA`, `can-read`

User	Roles
Fred	`region-EMEA`, `can-read`
Peter	`region-APAC`, `can-read`

Insert the Documents and Add Permissions

Using Query Console, insert the following documents to the database. `/doc5.xml` and `/doc6.xml` are added with read permissions for `can-read`, so that they are visible to anyone that has `can-read` role.

```
xquery version "1.0-ml";
xdmp:document-insert("/doc1.xml",
   <root>
     <metadata>
     <region>region-NA</region>
     <group>group-engineering</group>
     </metadata>
     <email>jane@companyabc.com</email>
     <feature>New feature</feature>
   </root>),
xdmp:document-insert("/doc2.xml",
   <root>
     <metadata>
       <region>region-NA</region>
       <group>group-finance</group>
     </metadata>
     <email>matt@companyabc.com</email>
     <price>100</price>
   </root>),
xdmp:document-insert("/doc3.xml",
   <root>
     <metadata>
       <region>region-EMEA</region>
       <group>group-engineering</group>
     </metadata>
     <email>jim@companyabc.com</email>
<feature>Another new feature</feature>
   </root>),
xdmp:document-insert("/doc4.xml",
   <root>
     <metadata>
       <region>region-APAC</region>
       <group>group-finance</group>
     </metadata>
     <email>jeff@companyabc.com</email>
     <price>10</price>
   </root>),
xdmp:document-insert("/doc5.xml",
   <root>
     <metadata>
       <region>region-all</region>
       <group>group-all</group>
     </metadata>
     <email>dummy@companyabc.com</email>
   </root>),
xdmp:document-insert("/doc6.xml",
   <root>
     <metadata>
      <region>region-all</region>
      <group>group-finance</group>
     </metadata>
     <email>dummy@companyabc.com</email>
   </root>),
xdmp:document-add-permissions("/doc5.xml", xdmp:permission("can-read","read")),
xdmp:document-add-permissions("/doc6.xml", xdmp:permission("can-read","read"))
```

Test It Out

The definitional queries on the roles will effectively treat documents as having permissions for that role. As a result, when Edna, Fred and Peter perform a search (read) against the database, they are able to read the following documents:

Document	Metadata	User with read Access
/doc1.xml	region-NA	Edna

Document	Metadata	User with `read` Access
`/doc2.xml`	`region-NA`	Edna
`/doc3.xml`	`region-EMEA`	Fred
`/doc4.xml`	`region-APAC`	Peter
`/doc5.xml`	`can-read` permission key	Edna, Fred, Peter
`/doc6.xml`	`can-read` permission key	Edna, Fred, Peter

8.2.2. Scenario 2: Group Restrictions

Description: Another security architect Carly from Company XYZ now wants to enforce a policy that only folks in the engineering group should be able to see feature design specifications, and that only folks in the finance group should be able to read and update documents with pricing information. This scenario will show the interaction between QBAC and Compartment Security. For more information about Compartment Security, see Compartment Security.

Carly didn't need to use compartment security here because there is only one dimension of access, but she thinks she may have others and wants them to be intersectional. Since the update policy is of the form `if (query) then Deny`, we need to also put the negated queries on the roles that we want to exclude, so the implementation is a little more complicated.

Mike is a contractor who works for Company XYZ. He is only able to read the documents marked with "group-all" in the metadata. He cannot see any other documents in the database. Carly sets up a user for him and grants permissions through user queries, which are restrictive.

To run through the example, follow the steps in each subsection.

Create Roles

Carly sets up some roles, `can-update`, `can-read`, `group-all`, `group-engineering`, and `group-finance`, by running this code against the Security database:

```
xquery version "1.0-ml";
import module namespace sec="http://marklogic.com/xdmp/security"
  at "/MarkLogic/security.xqy";
(: Uncompartmented roles can-read and can-update for compartment setup :)
sec:create-role("can-read", "General read", (), (), ()),
sec:create-role("can-update", "General update", (), (), ()),

(: Compartment role group-all for compartment permissions :)
sec:create-role("group-all", "All groups.", (), (), (), "compartment-group"),

sec:create-role("group-engineering", "Engineering.",
   (), (), (),"compartment-group", (),
     map:map()=>map:with(
     "node-update", cts:not-query(cts:element-query(xs:QName("price"), cts:true-query()))
   )=>map:with(
     "read", cts:element-query(xs:QName("feature"),cts:true-query())
   )
),
sec:create-role("group-finance", "Finance.",
   (), (), (), "compartment-group", (),
     map:map()=>map:with(
     "node-update", cts:element-query(xs:QName("price"), cts:true-query())
   )=>map:with(
     "read", cts:element-query(xs:QName("price"), cts:true-query())
   )
);
xquery version "1.0-ml";

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

sec:create-user("Mike", "Contractor", "Mike",
   ("can-read"), (), (), (),
   map:map()=>map:with(
     "read",cts:element-query(xs:QName("metadata"),
cts:element-word-query(xs:QName("group"), "group-all"))
    )
)
```

Create Users

Using the Admin Interface > Security > Users > Create, Query Console or RMAs, Carly creates the users and assign them the roles indicated in the following table:

User	Roles

John	`group-engineering`
	`can-read`
	`can-update`

Pari	`group-finance`
	`can-read`
	`can-update`

Mike	none

Insert the Documents and Add Permissions

For simplicity, we will reuse the 6 documents in Scenario 1: Region Restrictions. Here are the new permissions that need to be added:

```
xquery version "1.0-ml";
(: Doc 1 to 4 have compartment compartment-group. Doc 5 and 6 don't :)

let $permissions := (xdmp:permission("can-read","read"),
                     xdmp:permission("can-update","node-update"),
                     xdmp:permission("group-all","read"))
for $i in 1 to 4
return xdmp:document-add-permissions("/doc"||$i||".xml", $permissions),

xdmp:document-add-permissions("/doc5.xml",
                     (xdmp:permission("can-read","read"),
                     xdmp:permission("can-update","node-update"))),

xdmp:document-add-permissions("/doc6.xml",
                     (xdmp:permission("can-read","read"),
                     xdmp:permission("can-update","node-update")))
```

Test It Out

When John and Pari perform a read or a node-update against the database, the results are shown in the following table:

Document	Metadata	User with `read` Access	User with `node-update` Access
`/doc1.xml`	feature	John	
`/doc2.xml`	price	Pari	Pari
`/doc3.xml`	feature	John	
`/doc4.xml`	price	Pari	Pari
`/doc5.xml`	`can-read` and `can-update` permission keys, `group-all`	John, Pari, Mike	John, Pari
`/doc6.xml`	`can-read` and `can-update` permission keys	John, Pari	John, Pari

Mike is not able to read `/doc6.xml` although he has the `can-read` role. The access to `/doc6.xml` is further restricted by the user queries.

8.3. Interfaces to Support QBAC

Some existing Security APIs have been modified to support query-based access control. In addition, several new APIs have been added.

8.3.1. Changes to Security Module APIs

The following security APIs are updated to allow for queries to be added to users and roles, `sec:create-user()` and `sec:create-role()`:

```
sec:create-user(
  $user-name as xs:string,
  $description as xs:string?,
  $password as xs:string,
  $role-names as xs:string*
  $permissions as element(sec:permission)*,
  $collections as xs:string*,
  [$external-names as xs:string*],
  [$queries as map:map]
  ) as xs:unsignedLong
sec:create-role(
  $role-name as xs:string,
  $description as xs:string?,
  $role-names as xs:string*,
  $permissions as element(sec:permission)*,
  $collections as xs:string*,
  [$compartment as xs:string?],
  [$external-names as xs:string*],
  [$queries as map:map]
)   as xs:unsignedLong
```

Queries are a mapping from capabilities to CTS queries.

Capabilities associated through permissions are `read`, `insert`, `update`, `node-update`, and `execute`. For more information about Document Permissions, see Capabilities Associated through Permissions. Please note that, in terms of QBAC queries, operations that need a `node-update` capability will use the `node-update` query, and those that need `update` capability will use `update` query to reduce complexity. The `node-update` capability does not serve as a subset of the `update` capability.

These new APIs are added to support QBAC:

```
sec:role-get-queries($role-name as xs:string) as map:map
```

The `sec:role-get-queries()` function requires the privilege `http://marklogic.com/xdmp/privileges/role-get-queries`.

```
sec:role-set-queries(
  $role-name as xs:string,
  $queries as map:map
  ) as empty-sequence()
```

The `sec:role-set-queries()` functions requires the privilege `http://marklogic.com/xdmp/privileges/role-set-queries`.

```
sec:role-set-query(
  $role-name as xs:string,
  $capability as xs:string,
  $query as cts:query?
  ) as empty-sequence()
```

The `sec:role-set-query()` function requires the privilege `http://marklogic.com/xdmp/privileges/role-set-queries`.

```
sec:user-get-queries($user-name as xs:string) as map:map
```

The `sec:user-get-queries()` requires the privilege `http://marklogic.com/xdmp/privileges/user-get-queries`.

```
sec:user-set-queries(
  $user-name as xs:string,
  $queries as map:map
  ) as empty-sequence()
```

The sec:user-set-queries() function requires the privilege http://marklogic.com/xdmp/privileges/user-set-queries.

```
sec:user-set-query(
  $user-name as xs:string,
  $capability as xs:string,
  $query as cts:query?
  ) as empty-sequence()
```

The sec:user-set-query() function requires the privilege http://marklogic.com/xdmp/privileges/user-set-queries.

8.3.2. Admin Interface

Roles and users will get an additional query map. There is a read-only property on the corresponding roles and users page on the Admin Interface that shows the queries and capabilities for debugging purpose.

8.4. Errors

A query may raise an error. For example, if a range index is referenced but not available, or stemmed searches are used but not enabled. A failed query will lead to denial of access.

8.5. Limitations

- Users with QBAC document access are not able to read document properties. This is a design limitation. Users with QBAC document access do not have properties access by default, unless the QBAC query explicitly matches document properties through a CTS query. However, QBAC access to document properties gives access to the document itself by default.
- Queries run unfiltered. If a query has false positives that means that access may be granted where it is not intended to.
- It is not recommended to use expensive QBAC queries (for example, wildcards with lexicon expansion), since they run on every database request.
- Queries may depend on specific indexes (for example, range queries). If those indexes are deleted, the queries will fail and will lead to denial of access.
- Configuration of QBAC queries is through security APIs and RMAs only. See the RMAs for configuring roles and users at https://docs.marklogic.com/REST/POST/manage/v2/roles and https://docs.marklogic.com/REST/POST/manage/v2/users.

9. Granular Privileges

Granular privileges extend MarkLogic Server security model by allowing finer granularity access control over configuration and various administration abilities. *Granular privileges* is a subtype of *execute privileges* type. These are the purposes of granular privileges:

- Allow different applications to coexist in a single cluster, with some users having authority over some parts of the cluster and other users having authority over other parts of the cluster.
- Support separation of concerns between different administrative users, constraining control to just the layers they are concerned with.

This section describes granular privileges.

9.1. Understanding Granular Privileges

The MarkLogic Server security model includes execute privileges. Execute privileges are identified with URIs and can be assigned to roles. For detail on execute privileges, see Protecting XQuery and JavaScript Functions with Privileges.

For example, the following privilege allows a user to restart any forest:

`http://marklogic.com/xdmp/privileges/xdmp-forest-restart`

Granular privileges allow more fine-grained approach to execute privileges. When assigning privileges to roles, you may not only specify a privilege to perform a specific action but also identify a specific resource to which this privilege applies.

For example, you may allow a user to restart a specific forest by assigning one of the following privileges to this user's role:

`http://marklogic.com/xdmp/privileges/xdmp-forest-restart/forest/`*`forest-ID`*

`http://marklogic.com/xdmp/privileges/xdmp-forest-restart/database/`*`database-ID`*

where *`forest-ID`* is the forest identifier and *`database-ID`* is the identifier of the database using the forest.

You can create an appropriate fine-grained privilege, assign it to some role, and assign that role to a user. Then the user will be able to restart the specified forest, or forests in the specified database.

9.2. Categories of Granularity

You can use various categories of granular privileges to limit access to privileged operations.

9.2.1. Privileges to Read, Write, or Delete Any Configuration File

A privilege in this category grants a user the ability to read, write, or delete any configuration file as specified (for example, call to `xdmp:write-cluster-config-file()`). This privilege is specific to the operation (for example, `"write"`) and the scope (for example, `"cluster"`). The combination of the two values is a specific privilege (for example, `http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file`).

The following granular privileges belong to this category:

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file`

`http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file` `http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file`

9.2.2. Privileges to Read, Write, or Delete a Specific Configuration File

A privilege in this category grants a user the ability to read, write, or delete a specific configuration file (for example, `databases.xml`). This privilege is specific to the operation (for example, `"write"`), scope (for example, `"cluster"`), and the configuration file (for example, `"databases.xml"`). The combination of the three values is a specific privilege (for example, `http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/databases.xml`).

The following privileges belong to this category:

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/assignments.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/calendars.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/clusters.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/countries.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/databases.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/groups.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/hosts.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/languages.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/mimetypes.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/security.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/server.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/tokenizer.xml`

`http://marklogic.com/xdmp/privileges/xdmp-read-cluster-config-file/user-languages.xml`

`http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/assignments.xml`

`http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/calendars.xml`

`http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/clusters.xml`

`http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/countries.xml`

```
http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/
databases.xml

http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/
groups.xml

http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/
hosts.xml

http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/
languages.xml

http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/
mimetypes.xml

http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/
security.xml

http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/
server.xml

http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/
tokenizer.xml

http://marklogic.com/xdmp/privileges/xdmp-write-cluster-config-file/user-
languages.xml

http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/
assignments.xml

http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/
calendars.xml

http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/
clusters.xml

http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/
countries.xml

http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/
databases.xml

http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/
groups.xml

http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/
hosts.xml

http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/
languages.xml

http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/
mimetypes.xml

http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/
security.xml

http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/
server.xml
```

`http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/tokenizer.xml`

`http://marklogic.com/xdmp/privileges/xdmp-delete-cluster-config-file/user-languages.xml`

9.2.3. Privileges to Administer a Set of Resources

A privilege of this category grants a user the ability to administer a specific set of resources (for example, databases). This privilege is specific to the resource set (for example, `"databases"`), which defines the specific privilege (for example, `http://marklogic.com/xdmp/privileges/admin/database`). This privilege may imply the privilege to read and write a specific configuration file.

The following privileges belong to this category:

`http://marklogic.com/xdmp/privileges/admin/database`

`http://marklogic.com/xdmp/privileges/admin/forest`

`http://marklogic.com/xdmp/privileges/admin/host`

`http://marklogic.com/xdmp/privileges/admin/app-server`

`http://marklogic.com/xdmp/privileges/admin/app-server-security`

`http://marklogic.com/xdmp/privileges/admin/group`

`http://marklogic.com/xdmp/privileges/admin/group-security`

`http://marklogic.com/xdmp/privileges/admin/cluster`

`http://marklogic.com/xdmp/privileges/admin/mimetypes`

NOTE

Privileges of this category are pre-defined and included with every installation of MarkLogic Server. You can view them in the Execute Privileges Summary page of the Admin Interface (see instructions in Viewing an Execute Privilege in *Administrating MarkLogic Server*).

9.2.4. Privileges to Administer a Specific Resource

A privilege of this category grants a user an ability to administer a specific resource (for example, a database with the specified identifier). This privilege is granted by suffixing the administrator privilege for that kind of resource (for example, `"database"`) with the specific identifier (for example, `database-`*`ID`*), which results in the specific privilege (for example, `http://marklogic.com/xdmp/privileges/admin/database/`*`database-ID`*). This privilege may imply the privilege to read and write a portion of a configuration file. It also grants the ability to call various built-in functions for specific resources (for example, `http://marklogic.com/xdmp/privileges/xdmp-forest-clear/forest/`*`forest-ID`* privilege allows calls to `xdmp:forest-clear()` for that forest identifier).

The following privileges belong to this category:

`http://marklogic.com/xdmp/privileges/admin/database/`*`database-ID`*

`http://marklogic.com/xdmp/privileges/admin/forest/`*`forest-ID`* `http://marklogic.com/xdmp/privileges/admin/host/`*`host-ID`*

`http://marklogic.com/xdmp/privileges/admin/app-server/`*`server-ID`*

`http://marklogic.com/xdmp/privileges/admin/app-server-security/`*`server-ID`*

`http://marklogic.com/xdmp/privileges/admin/group/`*`group-ID`*

`http://marklogic.com/xdmp/privileges/admin/group-security/`*`group-ID`*

`http://marklogic.com/xdmp/privileges/admin/cluster/`*`cluster-ID`*

9.2.5. Privileges to Administer a Specific Aspect of a Set of Resources

A privilege of this category grants a user an ability to administer a specific aspect (for example, backup) of a set of resources (for example, databases). This privilege is granted by suffixing the administrator privilege for that kind of resource (for example, `"database"`) with the specific aspect (for example, `"backup"`), which results in the specific privilege (for example, `http://marklogic.com/xdmp/privileges/admin/database/backup`). This privilege may imply the privilege to read and write a portion of a configuration file.

The following privileges belong to this category:

`http://marklogic.com/xdmp/privileges/admin/database/forests`

`http://marklogic.com/xdmp/privileges/admin/database/backup`

`http://marklogic.com/xdmp/privileges/admin/database/index`

`http://marklogic.com/xdmp/privileges/admin/database/replication`

`http://marklogic.com/xdmp/privileges/admin/database/forest-backup`

`http://marklogic.com/xdmp/privileges/admin/forest/backup`

`http://marklogic.com/xdmp/privileges/admin/group/scheduled-task`

9.2.6. Privileges to Administer a Specific Aspect of a Specific Resource

A privilege of this category grants a user an ability to administer a specific aspect (for example, backup) of a specific resource (for example, the database with identifier *`database-ID`*). This privilege is granted by suffixing the privilege for the specific aspect (for example, `"backup"`) of that kind of resource (for example, `"database"`) with the specific identifier (for example, `"`*`database-ID`*`"`), which results in the specific privilege (for example, `http://marklogic.com/xdmp/privileges/admin/database/backup/`*`database-ID`*). This privilege may imply the privilege to read and write a portion of a configuration file.

The following privileges belong to this category:

`http://marklogic.com/xdmp/privileges/admin/database/forests/`*`database-ID`*

`http://marklogic.com/xdmp/privileges/admin/database/backup/`*`database-ID`*

`http://marklogic.com/xdmp/privileges/admin/database/index/`*`database-ID`*

`http://marklogic.com/xdmp/privileges/admin/database/index/`*`database-name`*

`http://marklogic.com/xdmp/privileges/admin/database/replication/`*`database-ID`*

`http://marklogic.com/xdmp/privileges/admin/database/forest-backup/`*`database-ID`* `http://marklogic.com/xdmp/privileges/admin/forest/backup/`*`forest-ID`*

`http://marklogic.com/xdmp/privileges/admin/group/scheduled-task/`*`group-ID`*

A user with any of the following privileges

`http://marklogic.com/xdmp/privileges/admin/database/index`

`http://marklogic.com/xdmp/privileges/admin/database/index/`*`database-ID`*

`http://marklogic.com/xdmp/privileges/admin/database/index/`*`database-name`*

can alter the following properties:

Property	Description
`attribute-value-positions`	Index attribute value positions for faster near searches involving element-attribute-value-query (slower document loads and larger database files).
`collection-lexicon`	Maintain a lexicon of collection URIs (slower document loads and larger database files).
`default-rulesets`	The default rulesets configuration.
`element-attribute-word-lexicons`	Maintain lexicons of words in elements.
`element-value-positions`	Index element value positions for faster near searches involving element-value-query (slower document loads and larger database files).
`element-word-lexicons`	Maintain lexicons of words in XML elements or JSON properties.
`element-word-positions`	Index element word positions for faster element-based phrase and near searches (slower document loads and larger database files).
`element-word-query-throughs`	The element-word-query-through specifications.
`fast-case-sensitive-searches`	Enable faster case sensitive searches (slower document loads and larger database files).
`fast-diacritic-sensitive-searches`	Enable faster diacritic sensitive searches (slower document loads and larger database files).
`fast-element-character-searches`	Enable element wildcard searches and element-character-based XQuery predicates (slower document loads and larger database files).
`fast-element-phrase-searches`	Enable faster element phrase searches (slower document loads and larger database files).
`fast-element-trailing-wildcard-searches`	Enable element trailing wildcard searches (slower document loads and larger database files).
`fast-element-word-searches`	Enable faster element-word searches (slower document loads and larger database files).
`fast-phrase-searches`	Enable faster phrase searches (slower document loads and larger database files).
`fast-reverse-searches`	Enable faster reverse searches (slower document loads and larger database files).
`field-value-positions`	Index field value positions for faster near searches involving field-value-query (slower document loads and larger database files).
`field-value-searches`	Index field values for faster searches involving field-value-query (slower document loads and larger database files).
`fields`	The fields specifications.
`geospatial-element-attribute-pair-indexes`	Indexes for fast geospatial element comparisons.
`geospatial-element-child-indexes`	Indexes for fast geospatial element comparisons.
`geospatial-element-indexes`	Indexes for fast geospatial element comparisons.
`geospatial-element-pair-indexes`	Indexes for fast geospatial element comparisons.
`geospatial-path-indexes`	Indexes for fast geospatial path-based comparisons.
`geospatial-region-path-indexes`	Indexes for fast geospatial region comparisons.
`language`	The default language assumed for content (if xml:lang encoding is absent)
`path-namespaces`	The namespace binding specifications for Path indexes.
`phrase-arounds`	The phrase-around specifications.
`phrase-throughs`	The phrase-through specifications.

Property	Description
`range-element-attribute-indexes`	Indexes for fast element-attribute inequality comparisons.
`range-element-indexes`	Indexes for fast inequality comparisons.
`range-index-optimize`	Specifies how to optimize range indexes.
`range-path-indexes`	Indexes for fast inequality comparisons.
`stemmed-searches`	Enable stemmed word searches (slower document loads and larger database files).
`tf-normalization`	What kind of TF normalization to apply.
`three-character-searches`	Enable wildcard searches and faster character-based XQuery predicates using three or more characters (slower document loads and larger database files).
`three-character-word-positions`	Index word positions for three-character searches only when three-character-searches are enabled (slower document loads and larger database files).
`trailing-wildcard-searches`	Enable trailing wildcard searches (slower document loads and larger database files).
`trailing-wildcard-word-positions`	Index word positions for trailing-wildcard searches only when trailing-wildcard-searches are enabled (slower document loads and larger database files).
`triple-index`	Enable the RDF triple index (slower document loads and larger database files).
`triple-positions`	Index triple positions for faster near searches involving cts:triple-range-query (slower document loads and larger database files).
`uri-lexicon`	Maintain a lexicon of document URIs (slower document loads and larger database files).
`word-lexicons`	A list of word lexicons. Each lexicon is defined by its collation URI.
`word-positions`	Index word positions for faster phrase and near searches (slower document loads and larger database files).
`word-searches`	Enable unstemmed word searches (slower document loads and larger database files).

9.3. Configuring Granular Privileges

You can configure granular privileges either via the Admin Interface or via the functions of XQuery API security module.

This section describes both mechanisms.

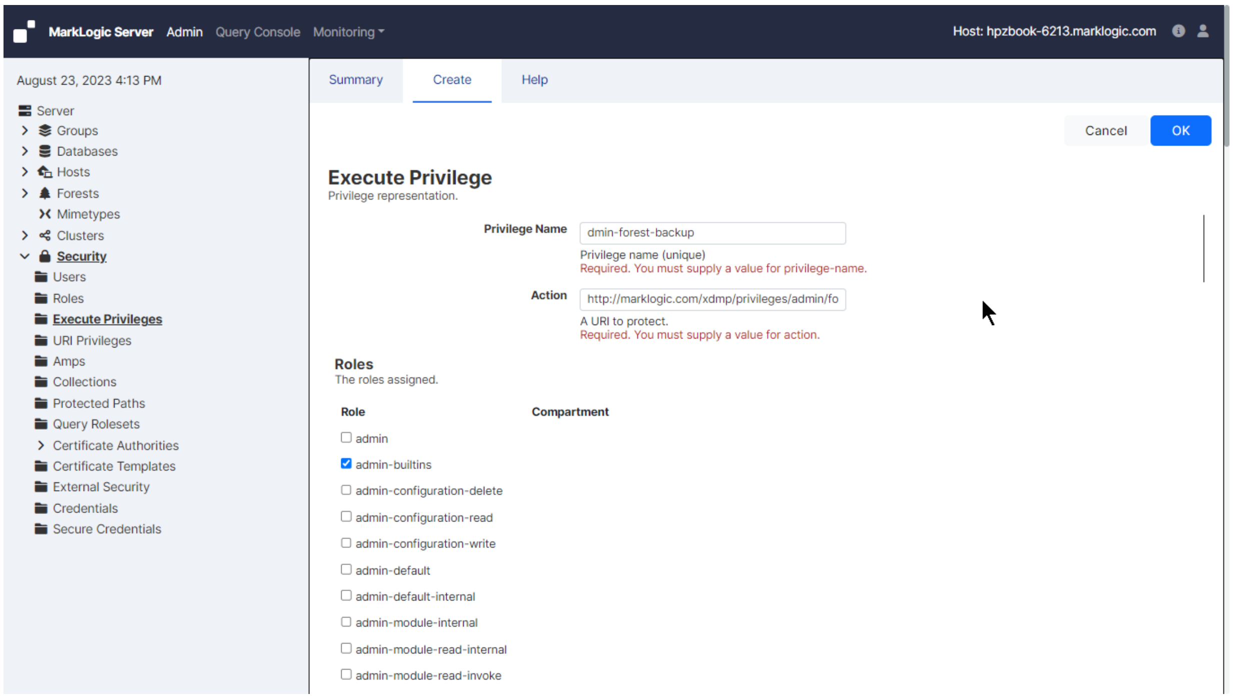
9.3.1. Configure Granular Privileges via the Admin Interface

To create a new granular privilege via the Admin Interface, follow steps for creating an execute privilege described at Creating an Execute Privilege in *Administering MarkLogic Server*.

For example, to create a granular privilege that grants a user an ability to administer a specific aspect (for example, backup) of a set of resources (for example, forests), perform the following steps:

1. Use the Admin Interface to create an execute privilege named `admin-forest-backup.`
2. Assign the action URI `http://marklogic.com/xdmp/privileges/admin/forest/backup` to the privilege.
3. Assign the privilege to the desired role or roles. You may want to create a specific role for this privilege depending on your security requirements.

The following screenshot depicts the **Execute Privilege** page with these parameters:



August 23, 2023 4:13 PM

- Server
- Groups
- Databases
- Hosts
- Forests
- Mimetypes
- Clusters
- Security
  - Users
  - Roles
  - Execute Privileges
  - URI Privileges
  - Amps
  - Collections
  - Protected Paths
  - Query Rolesets
  - Certificate Authorities
  - Certificate Templates
  - External Security
  - Credentials
  - Secure Credentials

Execute Privilege
Privilege representation.

Privilege Name dmin-forest-backup
Privilege name (unique)
Required. You must supply a value for privilege-name.

Action http://marklogic.com/xdmp/privileges/admin/fo
A URI to protect.
Required. You must supply a value for action.

Roles
The roles assigned.

Role	Compartment
☐ admin	
☑ admin-builtins	
☐ admin-configuration-delete	
☐ admin-configuration-read	
☐ admin-configuration-write	
☐ admin-default	
☐ admin-default-internal	
☐ admin-module-internal	
☐ admin-module-read-internal	
☐ admin-module-read-invoke	

NOTE
You cannot create a granular privilege that grants a user the ability to administer a specific resource (such as a forest with the specified identifier) in the manner described here because resource identifiers are not exposed in the Admin Interface. To create a granular privilege of this type (for example, `http://marklogic.com/xdmp/privileges/admin/forest/`*forest-ID*), you need to use the functions of the XQuery API security module, as described in the following section Configure Granular Privileges via the XQuery API Security Module.

9.3.2. Configure Granular Privileges via the XQuery API Security Module

You can use the XQuery API security module to create and assign granular privileges.

Creating and Assigning Granular Privileges

To create a new granular privilege programmatically, use the following function of the XQuery API security module:

```
sec:create-privilege(
   $privilege-name as xs:string,
   $action as xs:string,
   $kind as xs:string,
   $role-names as xs:string*
) as xs:unsignedLong
```

To assign an existing granular privilege to an additional role, use the following function of the XQuery API security module:

```
sec:privilege-set-roles(
   $action as xs:string,
   $kind as xs:string,
   $role-names as xs:string*
) as empty-sequence()
```

For detailed descriptions of `sec:create-privilege()` and `sec:privilege-set-roles()` functions of the `security.xqy` library module, see the *MarkLogic XQuery and XSLT Function Reference*.

Using Pseudo-functions with Granular Privileges

When you have a payload that creates a database and a granular privilege for that database, you need to substitute a variable of some sort for the ID of the database because the database has yet to be created. MarkLogic Server has the following pseudo-functions that can be used when creating and assigning granular privileges:

Pseudo-Function and Parameters	Replaced By...
`$$group-id(group-name)`	The group ID of the named group.
`$$database-id(database-name)`	The database ID of the named database.
`$$host-id()`	The host ID of the host running the query.
`$$host-id(host-name)`	The host ID of the named host.
`$$forest-id(forest-name)`	The forest ID of the named forest.
`$$cluster-id()`	The cluster ID of the cluster to which the host running the query belongs.
`$$cluster-id(cluster-name)`	The cluster ID of the named cluster.
`$$role-id(role-name)`	The role ID of the named role.
`$$user-id(user-name)`	The user ID of the named user.
`$$server-id(server-name)`	The server ID of the named server in the group to which the host running the query belongs.
`$$server-id("server-name", group-id)`	The server ID of the named server in the specified group. Note that `group-id` is an unsigned long. To refer to the group by name as well, nest the calls: `$$server-id(server-name, $$group-id(group-name))`
`$$privilege-id("privilege-name")`	The privilege ID of the named /execute/ privilege.
`$$privilege-id("privilege-name", "execute")`	The privilege ID of the named execute privilege.
`$$privilege-id("privilege-name", "uri")`	The privilege ID of the named URI privilege.

For example, to create the privilege `finalDbName-index-editor` for a not-yet-created database represented by the variable `FinalDbName`, execute the following code:

```
{
    "privilege-name": "finalDbName-index-editor",
    "action": "http://marklogic.com/xdmp/privileges/admin/database/index/$$database-
id(FinalDbName)",
    "role": ["firstEditorRole","secondEditorRole"],
    "kind": "execute"
  }
```

Examples of Creating and Assigning Granular Privileges

The following are examples of creating and assigning granular privileges via the XQuery API. They must be run against the Security database.

Example 1: Assign a privilege to perform index operations on any database to `role1`

Suppose you previously created `http://marklogic.com/xdmp/privileges/admin/database/index` privilege via the Admin Interface, as described in the previous section, Configure Granular Privileges via the Admin Interface. Assign this privilege to `role1` as follows:

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

sec:privilege-set-roles(
   "http://marklogic.com/xdmp/privileges/admin/database/index",
   "execute",
   ("admin","role1")
)
```

Example 2: Create a privilege to perform any operations on database `db1` for `role2`

Create a privilege to perform any operations on database `db1` for `role2` as follows (note the use of function `xdmp:database("db1")` to convert from the database name to the database identifier):

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

sec:create-privilege(
   "admin-database-db1",
   fn:concat("http://marklogic.com/xdmp/privileges/admin/database/",
xdmp:database("db1")),
   "execute",
   "role2"
)
```

Example 3: Create a privilege to perform index operations on database `db1` for `role3`

Create a privilege to perform index operations on database `db1` for `role3` as follows (note the use of function `xdmp:database("db1")` to convert from the database name to the database identifier):

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

sec:create-privilege(
   "admin-index-database-db1",
   fn:concat("http://marklogic.com/xdmp/privileges/admin/database/index/",
xdmp:database("db1")),
   "execute",
   "role3"
)
```

9.4. Examples of Granular Privileges Usage

This section describes several scenarios that use granular privileges.

9.4.1. Prerequisites - Create Databases, Roles, Users, and Privileges

To execute the scenarios discussed in this section, you need to perform the following preparation steps:

1. Using the Admin Interface, create databases `db1` and `db2`. For details on creating databases, see Creating a New Database in *Administering MarkLogic Server*.
2. Using the Admin Interface, create roles `role1`, `role2`, and `role3`. For details on creating roles, see Creating a Role in *Administering MarkLogic Server*.
3. Using the Admin Interface, create users `user1`, `user2`, and `user3` with roles `role1`, `role2`, and `role3` correspondingly. For details on creating users and assigning roles to them, see Creating a User in *Administering MarkLogic Server*.
4. Create and assign granular privileges to roles `role1`, `role2`, and `role3` as described in Example 1, Example 2, and Example 3 correspondingly of the previous section Configure Granular Privileges via the XQuery API Security Module.

As the result, you will have the users with roles and privileges as described in the following table:

User	Role	Privilege
user1	role1	http://marklogic.com/xdmp/privileges/admin/database/index
user2	role2	http://marklogic.com/xdmp/privileges/admin/database/*db1_identifier*
user3	role3	http://marklogic.com/xdmp/privileges/admin/database/index/*db1_identifier*

9.4.2. Scenarios That Use Granular Privileges

This section includes examples in XQuery that you may run for user1, user2, and user3 from the Query Console and observe different results depending on the user's privileges. The results are discussed in detail in the next section, Test It Out.

Scenario 1: Add range index to database db1

Execute the following XQuery code to add a range index to database db1:

```
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("db1")
let $rangespec := admin:database-range-element-index("int", "http://marklogic.com/qa",
"column1", (), fn:false())
let $config := admin:database-add-range-element-index($config, $dbid, $rangespec)
return admin:save-configuration($config)
```

Scenario 2: Add range index to database db2

Execute the following XQuery code to add a range index to database db2:

```
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("db2")
let $rangespec := admin:database-range-element-index("int", "http://marklogic.com/qa",
"column1", (), fn:false())
let $config := admin:database-add-range-element-index($config, $dbid, $rangespec)
return admin:save-configuration($config)
```

Scenario 3: Add backup for database db1

Execute the following XQuery code to add a backup for database db1:

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin" at "/MarkLogic/
admin.xqy";
let $config := admin:get-configuration()
let $backup := admin:database-monthly-backup("/space/backup", 2, 1, xs:time("09:45:00"),
2, true(), true(), true())
return admin:save-configuration(admin:database-add-backup($config, xdmp:database("db1"),
$backup))
```

9.4.3. Test It Out

Using the Query Console, you can execute Scenario 1, Scenario 2, and Scenario 3 for each one of the users user1, user2, and user3. The results of the execution are presented in the following table:

User	Role	Scenario	Result
user1	role1	Add range index to database db1	Success
user1	role1	Add range index to database db2	Success

User	Role	Scenario	Result
`user1`	`role1`	Add backup for database db1	Failure
`user2`	`role2`	Add range index to database db1	Success
`user2`	`role2`	Add range index to database db2	Failure
`user2`	`role2`	Add backup for database db1	Success
`user3`	`role3`	Add range index to database db1	Success
`user3`	`role3`	Add range index to database db2	Failure
`user3`	`role3`	Add backup for database db1	Failure

The following analysis explains these results:

- The user `user1` successfully adds indexes to both databases `db1` and `db2`, but fails to add backup to database `db1`, because the user's `role1` has granular privilege `http://marklogic.com/xdmp/privileges/admin/database/index` that allows to add indexes to any database but does not allow other operations on databases.
- The user `user2` successfully adds both the index and backup to database `db1`, but fails to add index to database `db2`, because the user's `role2` has granular privilege `http://marklogic.com/xdmp/privileges/admin/database/`*db1_identifier* that allows this user to perform any operation on database `db1` but does not allow operations on other databases.
- The user `user3` successfully adds index to database `db1` but fails to add index to database `db2` and to add backup to database `db1`, because the user's `role3` has granular privilege `http://marklogic.com/xdmp/privileges/admin/database/index/`*db1_identifier* that allows to add indexes to database `db1` but does not allow any other operation on database `db1` and does not allow any operation on other databases.

9.5. Enabling Non-privileged Users to Create Privileges, Roles, and Users

Non-privileged users can use granular privileges to create and manage user privileges, create and manage roles, and create users.

9.5.1. Enabling Non-privileged Users to Assign Roles

The `create-user-privilege` privilege enables otherwise non-privileged users to create and manage user-defined privileges.

If a user has a role with this privilege set, they do not need the `grant-my-privileges` privilege to assign specific privileges.

The general form of this granular privilege is:

`http://marklogic.com/xdmp/privileges/admin/create-user-privilege/`*DOMAIN/PRIVILEGE-PATH/*

Note that the *PRIVILEGE-PATH* can contain more than one slash ("/") and must end with a slash.

For example, given a user with a role that has the following privilege:

```
http://marklogic.com/xdmp/privileges/admin/create-user-privilege/acme.com/publishing/
```

This user can manage the following execute or URI privileges:

- `http://acme.com/publishing/`
- `http://acme.com/publishing/updates/`
- `http://acme.com/publishing/updates/weekly/`

This user can also create roles that use these privileges, as long as the role name is unique to the entire system, including someone else’s set of roles.

As another example, if you only want this user to be able to publish weekly updates, you would assign them a role with the following privilege:

```
http://marklogic.com/xdmp/privileges/admin/create-user-privilege/acme.com/
publishing/updates/weekly
```

9.5.2. Enabling Non-privileged Users to Create and Manage Roles (Data Roles)

The `http://marklogic.com/xdmp/privileges/create-data-role` allows non-admin users (with the `manage` role) to create and manage roles.

- data role: created by a data manage (non-admin) user
- data manage user for data roles:
  - non-admin to create and manage roles
  - can only manage (edit, delete and grant) roles own created or granted
  - requires one role to include `create-data-role` privilege and `manage` role (or privilege)
  - user self can be created by `admin` or another data manage user
  - optional `grant-my-role` privilege to grant roles or create another data manage user
  - can grant own created or granted data roles to other data roles
- created data roles are attached to the roles (with `create-data-role` privilege) data manage user owned
  - tracked by internal `data-role-edit-<ROLEID>` and `data-role-inherit-<ROLEID>` privileges created for every data role
- every data manage user granted (new or existed) with above roles can also manage these data roles
  - to share responsibility for managing data roles through a common data role
- An optional privilege - `http://marklogic.com/xdmp/privileges/role-set-queries` - is required to create data roles with query-based access control (QBAC) queries. The `http://marklogic.com/xdmp/privileges/role-get-queries` privilege is needed for reading the QBAC queries on the data roles. For more information on QBAC, please see Query-Based Access Control.

For example:

Create role (`demo-data-role`), grant that role the `create-data-role` privilege.

```
curl -s --anyauth -u admin:admin -H "content-type:application/json" \
     -X POST -d "{\"role-name\": \"demo-data-role\",
                  \"description\": \
                     \"A role for demonstrating the create-data-role privilege\", \
                  \"privilege\": [ { \
                  \"privilege-name\": \"create-data-role\", \
                  \"action\": \"http://marklogic.com/xdmp/privileges/create-data-role\", \
                  \"kind\": \"execute\"}]}" \
     http://localhost:8002/manage/v2/roles
```

Create a user and grant that user (`demo-user`) the `demo-data-role` and the `manage` role.

```
curl -s --anyauth -u admin:admin -H "content-type:application/json" \
     -X POST -d "{\"user-name\": \"demo-user\", \"password\": \"password\", \
               \"description\": \"A demo user\", \
                \"role\": [ \"demo-data-role\", \"manage\" ] }" \
     http://localhost:8002/manage/v2/users
```

Now that user can create new roles, `demo-role-one`:

```
curl -s --anyauth -u "demo-user:password" -H "content-type:application/json" \
     -X POST -d "{\"role-name\": \"demo-role-one\",
                  \"description\": \"First demo role\" }" \
     http://localhost:8002/manage/v2/roles
```

And demo-role-two:

```
curl -s --anyauth -u "demo-user:password" -H "content-type:application/json" \
     -X POST -d "{\"role-name\": \"demo-role-two\",
                   \"description\": \"Second demo role\" }" \
     http://localhost:8002/manage/v2/roles
```

The users can assign roles they have created to each other:

```
curl -s --anyauth -u "demo-user:password" -H "content-type:application/json" \
     -X PUT -d "{\"role\": [\"demo-role-two\"]}" \
     http://localhost:8002/manage/v2/roles/demo-role-one/properties
```

But they cannot assign roles that they did not create. To allow a user to assign existing roles, you can grant this `demo-data-role` to another user or role, so that user can manage both `demo-role-one` and `demo-role-two`.

A user with the ability to edit a role may also delete it. When the role is deleted, the extra `data-role-edit` and `data-role-inherit` privileges associated with it are also removed.

9.5.3. Enabling Non-privileged Users to Create and Manage Users (Data Users)

The `http://marklogic.com/xdmp/privileges/create-data-user` allows non-admin users (with the `manage` role) to create and manage users.

- data user: created by a data manage (non-admin) user
- data role: created by a data manage (non-admin) user
- data manage user for data users:
  - non-admin to create and manage users
  - can only manage (edit and delete) users own created or granted
  - might be the same data manage user to create data roles and data users
  - requires one role to include `create-data-user` privilege and `manage` role (or privilege)
  - user self can be created by `admin` or another data manage user
  - optional `grant-my-role` privilege to grant roles or create another data manage user
  - can grant data users own created or granted to other data roles
- created data users are attached to the roles (with `create-data-user` privilege) data manage user owned
  - tracked by an internal `data-user-edit-<USERID>` privilege created for every data user
- every data manage user granted (new or existed) with above roles can also manage these data users
  - to share responsibility for managing data users through a common data role
- An optional privilege - `http://marklogic.com/xdmp/privileges/user-set-queries` - is required to create data users with query-based access control (QBAC) queries. The `http://marklogic.com/xdmp/privileges/user-get-queries` privilege is needed for reading the QBAC queries on the data users. For more information on QBAC, please see Query-Based Access Control.

For example:

Create a role (`demo-data-user-role-one`) and grant that role the `create-data-user` privilege.

```
curl -s --anyauth -u admin:admin -H "content-type:application/json" \
     -X POST -d "{\"role-name\": \"demo-data-user-role-one\", \
                  \"description\": \
                     \"A role for demonstrating the create-data-user privilege\", \
                  \"privilege\": [ { \
                     \"privilege-name\": \"create-data-user\", \
                     \"action\": \
                        \"http://marklogic.com/xdmp/privileges/create-data-user\", \
                     \"kind\": \"execute\"}]}" \
     http://localhost:8002/manage/v2/roles
```

Create another role (`demo-data-user-role-two`) and grant that role the `create-data-user` privilege.

```
curl -s --anyauth -u admin:admin -H "content-type:application/json" \
     -X POST -d "{\"role-name\": \"demo-data-user-role-two\", \
                  \"description\": \
                     \"Second role for demonstrating the create-data-user privilege\", \
                  \"privilege\": [ { \
                     \"privilege-name\": \"create-data-user\", \
                     \"action\": \
                        \"http://marklogic.com/xdmp/privileges/create-data-user\", \
                      \"kind\": \"execute\"}]}" \
     http://localhost:8002/manage/v2/roles
```

Create user `demo-user-one`, and grant two roles: the `manage` role, the new created `demo-data-user-role-one` role.

```
curl -s --anyauth -u admin:admin -H "content-type:application/json" \
     -X POST -d "{\"user-name\": \"demo-user-one\", \
                  \"password\":                   \"password\", \
                  \"description\": \"A demo user one\", \
                  \"role\": [ \"demo-data-user-role-one\", \"manage\" ] }" \
     http://localhost:8002/manage/v2/users
```

Also create another user `demo-user-two` and grant `demo-data-user-role-two` and `manage` role.

```
curl -s --anyauth -u admin:admin -H "content-type:application/json" \
     -X POST -d "{\"user-name\": \"demo-user-two\", \"password\": \"password\", \
                  \"description\": \"A demo user two\", \
                  \"role\": [ \"demo-data-user-role-two\", \"manage\" ] }" \
     http://localhost:8002/manage/v2/users
```

Now that user `demo-user-one` can create new users, `demo-one-created-user`:

```
curl -s --anyauth -u "demo-user-one:password" -H "content-type:application/json" \
     -X POST -d "{\"user-name\": \" demo-one-created-user\", \
                  \"description\": \"user created by demo-user-one\" }" \
     http://localhost:8002/manage/v2/users
```

And user `demo-user-two` can create new users, `demo-two-created-user`:

```
curl -s --anyauth -u "demo-user-two:password" -H "content-type:application/json" \
     -X POST -d "{\"user-name\": \" demo-two-created-user\", \
                   \"description\": \"user created by demo-user-two\" }" \
     http://localhost:8002/manage/v2/users
```

The user `demo-one-created-user` can be updated (and also deleted) by user `demo-user-one` who created this user:

```
curl -s --anyauth -u "demo-user-one:password" -H "content-type:application/json" \
     -X PUT -d "{\"description\": \"demo-user-one updated this\"}" \
     http://localhost:8002/manage/v2/users/demo-one-created-user/properties
```

And user `demo-user-two` can update `demo-two-created-user`:

```
curl -s --anyauth -u "demo-user-two:password" -H "content-type:application/json" \
     -X PUT -d "{\"description\": \"demo-user-two updated this\"}" \
     http://localhost:8002/manage/v2/users/demo-two-created-user/properties
```

But these users cannot update users they did not create.

```
curl -s --anyauth -u "demo-user-two:password" -H "content-type:application/json" \
     -X PUT -d "{\"description\": \"demo-user-two updating demo-one-created-user\"}" \
     http://localhost:8002/manage/v2/users/demo-one-created-user/properties
```

This request fails:

```
{
  "errorResponse": {
    "statusCode": "404",
    "status": "Not Found",
    "messageCode": "SEC-USERDNE",
    "message": "SEC-USERDNE: (err:FOER0000) User does not exist: demo-one-created-user =
%2"
  }
}
```

All users created by `demo-user-two` are attached to `demo-data-user-role-two` role. They can be added to `demo-user-one` directly, so `demo-user-one` can edit them.

```
curl -s --anyauth -u "admin:admin" -H "content-type:application/json" \
     -X PUT -d "{\"role\": [ \"demo-data-user-role-one\", \"demo-data-user-role-two\",
\"manage\" ] }" \
     http://localhost:8002/manage/v2/users/demo-user-one/properties
```

Now, user `demo-user-two` with role `demo-data-user-role-two` has the appropriate privilege to edit `demo-one-created-user` directly. So, `demo-user-two` can edit them, and the previous request will succeed.

9.6. Using Granular Privileges with MarkLogic Data Hub Service

MarkLogic Data Hub Service (DHS) provides a managed instance in which to deploy an operational data hub created using MarkLogic Data Hub.

The following roles are built into DHS:

Amazon Web Services (AWS)

- Service Roles
- Portal Roles

Microsoft Azure

- Service Roles
- Portal Roles

The following rules apply to granular privileges on a data hub:

- A user assigned the Security Admin service role cannot delete or modify privileges for these or any other pre-built roles, and these pre-built roles cannot inherit privileges.
- When a user assigned the Security Admin service role creates a DHS custom role, that role initially has no pre-built roles associated with it.
- Custom roles in DHS can inherit functionality from the pre-built DHS roles, from other DHS custom roles, or they can be created to have no inheritance, but you cannot assign any privileges to DHS custom roles.
- DHS custom roles cannot inherit privileges from any other (non-DHS) pre-built MarkLogic Server roles.

- You can change the external name for a DHS custom role, but the internal name stays constant.

10. Configuring SSL on App Servers

This section describes how to use the Admin Interface to configure SSL on App Servers. For details on how to configure SSL programmatically, see Enabling SSL on an App Server in the *Scripting Administrative Tasks Guide*.

10.1. Understanding SSL

SSL (Secure Sockets Layer) is a transaction security standard that provides encrypted protection between browsers and App Servers. When SSL is enabled for an App Server, browsers communicate with the App Server by means of an HTTPS connection, which is HTTP over an encrypted Secure Sockets Layer. HTTPS connections are widely used by banks and web vendors for secure transactions over the web.

A browser and App Server create a secure HTTPS connection by using a handshaking procedure. When browser connects to an SSL-enabled App Server, the App Server sends back its identification in the form of a digital certificate that contains the server name, the trusted certificate authority, and the server's public encryption key. The browser uses the server's public encryption key from the digital certificate to encrypt a random number and sends the result to the server. From the random number, both the browser and App Server generate a *session key*. The session key is used for the rest of the session to encrypt/decrypt all transmissions between the browser and App Server, enabling them to verify that the data didn't change in route.

The end result of the handshaking procedure described above is that only the server is authenticated. The client can trust the server, but the client remains unauthenticated. MarkLogic Server supports mutual authentication, in which the client also holds a digital certificate that it sends to the server. When mutual authentication is enabled, both the client and the server are authenticated and mutually trusted.

MarkLogic Server uses OpenSSL to implement the Secure Sockets Layer (SSL v3) and Transport Layer Security (TLS v1) protocols.

The following are the definitions for the SSL terms used in this section:

- A *certificate,* or more precisely, a *public key certificate*, is an electronic document that incorporates a digital signature to bind together a public key with identity information, such as the name of a person or an organization, address, and so on. The certificate can be used to verify that a public key belongs to an individual or organization. In a typical public key infrastructure (PKI) scheme, the signature will be that of a certificate authority.
- A *certificate authority* (CA) is a trusted third party that certifies the identity of entities, such as users, databases, administrators, clients, and servers. When an entity requests certification, the CA verifies its identity and grants a certificate, which is signed with the CA's private key. If the CA is trusted, then any certificate it issues is trusted unless it has been revoked.
- A *certificate chain* is a group of interdependent CAs. A certificate chain consists of a single trusted *root CA*, one or more *intermediate CA*, and one or more *end CA*. The intermediate and end certificates must be imported into MarkLogic Server.

NOTE
MarkLogic Server supports only one intermediate CA per host.

- A *certificate request* is a request data structure containing a subset of the information that will ultimately end up in the certificate. A certificate request is sent to a *certificate authority* for certification.
- A *key* is a piece of information that determines the output of a cipher. SSL/TLS communications begin with a public/private key pair that allow the client and server to securely agree on a session key. The public/private key pair is also used to validate the identity of the server and can optionally be used to verify the identity of the client.
- A *certificate template* is a MarkLogic Server construct that is used to generate certificate requests for the various hosts in a cluster. The template defines the name of the certificate, a description, and identity information about the owner of the certificate.
- A *cipher* is an algorithm for encrypting information so that it's only readable by someone with a key. A cipher can be either symmetric or asymmetric. Symmetric ciphers use the same key for both encryption and decryption. Asymmetric ciphers use a public and private key.

NOTE
Signed certificates are imported via the Certificate Templates import page, as described in Importing a Signed Certificate into MarkLogic Server. Certificate Authority certificates are imported via the Certificate Authorities import page, as described in CA Certificate (User Cert Signer) Import from Admin Interface.

10.2. General Procedure for Setting Up SSL for an App Server

This section describes the general procedure for setting up SSL on an App Server. These are the general steps:

- Create a certificate template, as described in Creating a Certificate Template.
- Enable SSL for the App Server, as described in Enabling SSL for an App Server.
- Access the SSL-enabled server from a browser, as described in Accessing an SSL-Enabled Server from a Browser or WebDAV Client.
- Generate a certificate request and send it off to a certificate authority, as described in Generating and Downloading Certificate Requests.
- When you receive the signed certificate from the certificate authority, import it into MarkLogic Server for use by your App Server, as described in Importing a Signed Certificate into MarkLogic Server.

NOTE
Certificate templates, requests, and the resulting signed certificates are only valid within a single cluster.

10.3. Procedures for Enabling SSL on App Servers

This section describes how to enable SSL for an App Server.

10.3.1. Creating a Certificate Template

Access to an SSL-enabled server is managed by a public key in a signed certificate obtained from a certificate authority. The first step in producing a request for a signed certificate is to define a certificate template. This procedure will produce a self-signed certificate that your browser can temporarily use to access an SSL-enabled server until you receive a signed certificate from a certificate authority.

1. Click **Security** in the left tree menu.
2. Click **Certificate Templates** in the left tree menu.
3. Click the **Create** tab.
4. In the **Template Name** field, enter a shorthand name for this certificate template. MarkLogic Server will use this name to refer to this template on display screens in the Admin Interface.
5. You can enter an optional description for the certificate template.
6. You can optionally fill in subject information, such as your country, state, locale, and email address. Country Name must be two characters, such as US, UK, DE, FR, ES, and so on.
7. Enter the name of your company or organization in the **OrganizationName** field.
8. When you have finished filling in the fields, click **OK**. MarkLogic Server automatically generates a Self-Signed Certificate Authority, which in turn automatically creates a signed certificate from the certificate template for each host. For details on how to view the Certificate Authority and signed certificate, see Viewing Trusted Certificate Authorities.

10.3.2. Enabling SSL for an App Server

After creating a certificate template, you can enable SSL for an HTTP, ODBC, WebDAV, or XDBC server.

1. Click the **Groups** icon in the left tree menu.
2. Click the group in which you want to define the HTTP server (for example, **Default**).
3. Click the **App Servers** icon on the left tree menu.
4. Either create a new server by clicking on one of the Create *server_type* tabs or select an existing server from the left tree menu.
   The SSL fields are located at the bottom of the server specification page.
5. In the SSL Certificate Template field, select the certificate template you created in Creating a Certificate Template. Selecting a certificate template implicitly enables SSL for the App Server.
6. (Optional) The SSL Hostname field should only be filled in when a proxy or load balancer is used to represent multiple servers. In this case, you can specify an SSL hostname here and all instances of the application server will identify themselves as that host.
7. (Optional) In the SSL Ciphers field, you can either use the default (`ALL:!LOW:@STRENGTH`) or one or more of the SSL ciphers defined in `https://www.openssl.org/docs/man1.0.2/man1/ciphers.html`.
8. (Optional) If you want SSL to require clients to provide a certificate, select True for SSL Require Client Certificate. Then select Show under SSL Client Certificate Authorities and which certificate authority is to be used to sign client certificates for the server.
9. (Optional) Set SSL Client Issuer Authority Verification to `True` to ensure that the App Server will accept client certificates only signed directly by a selected CA from the SSL Client Certificate Authorities list. A setting of `False` enables the App Server to accept client certificates that have a parent CA that is indirectly signed by one or more ancestor CAs selected in the Admin Interface (same as prior to MarkLogic Server 9.0-8).

10.3.3. Setting Response Headers for HTTPS-Enabled App Servers

App Servers that use HTTPS do not set strict-transport-security in the response header by default. MarkLogic Server has options to control HSTS (HTTP Strict-Transport-Security) headers.

NOTE
These options are only effective when the app server is configured with HTTPS.

The max age value can be set for the HSTS response headers. If the max age value for the HSTS is set to 0 (over an HTTPS connection) it immediately expires the Strict-Transport-Security header, allowing access via HTTP. The typical value used for HSTS is one year, expressed as 31536000.

These options can be set in three different ways.

Using the MarkLogic Server Admin Interface

In the Admin Interface, follow these steps:

1. In the left nav menu, click **Groups** > **Default** > **App Servers**. The App Server Summary page appears.
2. Click **App-Services** on the App Server Summary page. The HTTP Server configuration page appears.
3. Scroll to the SSL Client Certificate Authorities section at the bottom of the HTTP Server configuration page.
4. Set Enable HSTS Header to true.
5. Set the maximum age for the HSTS header in HSTS Header Max Age in seconds (31,536,000s = 1y).
6. Click **OK**.

Using Admin Functions

You can enable and configure HSTS headers through certain admin functions:

Use `appserver-get-enable-hsts-header` to get information about the HSTS header:

```
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:appserver-get-enable-hsts-header($config,
    admin:appserver-get-id($config, $groupid, "test"))
```

Use `appserver-get-hsts-header-max-age` to get information about the current HSTS header max age amount:

```
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:appserver-get-hsts-header-max-age($config,
  admin:appserver-get-id($config, $groupid, "test"))
```

Use `appserver-set-enable-hsts-header` to enable HSTS header:

```
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:appserver-set-enable-hsts-header($config,
  admin:appserver-get-id($config, $groupid, "test"),true())
```

Use `appserver-set-hsts-header-max-age` to set the HSTS max age amount:

```
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:appserver-set-hsts-header-max-age($config,
  admin:appserver-get-id($config, $groupid, "test"),31536000)
```

The max age amount is being set to 31,536,000 or one year.

Using REST APIs

These REST APIs can be used to set HSTS headers.

This REST call will get the current properties.

```
curl -X GET --digest -u admin:admin \
"http://localhost:8002/manage/v2/servers/test/properties?group-id=Default"
```

This REST call enables the HSTS header and sets the HSTS max age.

```
curl -X PUT --digest -u admin:admin -H "Content-type: application/json"\
-d '{"enable-hsts-header":true, "hsts-header-max-age":31536000}'\
"http://localhost:8002/manage/v2/servers/test/properties?group-id=Default"
```

The max age amount is being set to 31,536,000 or one year.

10.4. Accessing an SSL-Enabled Server from a Browser or WebDAV Client

When you create a certificate template and set it in your App Server, MarkLogic Server automatically generates a temporary self-signed MarkLogic Server certificate authority that signs host certificates. If you have not yet received a signed certificate for your SSL-enabled App Server from a certificate authority, your browser must accept the temporary self-signed certificate authority before it can access the App Server. There are two alternative ways to do this, both of which are browser-dependent and described below.

To enable WebDAV clients to access an SSL-enabled App Server, you must follow the procedure described in Importing a Self-Signed Certificate Authority into Windows.

To enable a single browser to access the SSL-enabled App Server, you can create a security exception for the self-signed certificate in your browser, as described in the following sections:

- Creating a Security Exception in Internet Explorer
- Creating a Security Exception in Google Chrome
- Importing a Self-Signed Certificate Authority into Windows

If you need to enable a number of browsers to access the SSL-enabled App Server, you might want each browser to import the self-signed certificate authority for the certificate template. Once this is done, all certificates signed by the certificate authority will be trusted by the browser, so you can distribute new certificates without requiring each browser to create new security exceptions. The following sections describe how to import the self-signed MarkLogic Server certificate authority:

- Importing a Self-Signed Certificate Authority into Windows
- Procedures for Obtaining a Signed Certificate

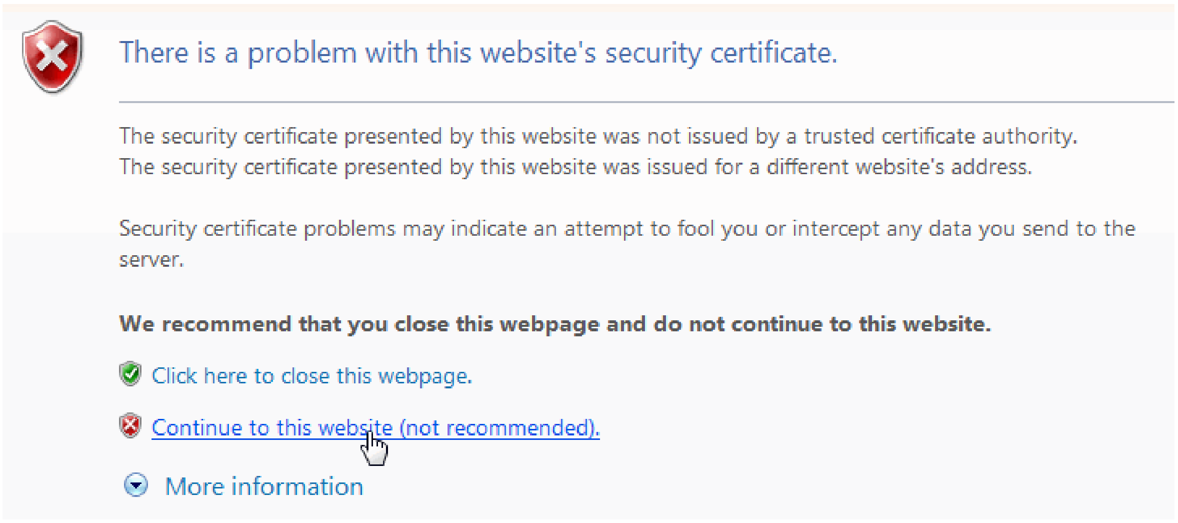
10.4.1. Creating a Security Exception in Internet Explorer

If you have not imported the certificate authority for the certificate template into Windows, when you first access an SSL-enabled server with your IE browser, you will receive an error notifying you that there is a problem with this website's security certificate. You can bypass this security exception by accepting the certificate. For example, if you enable SSL on the HTTP server, App-Services, each host can accept the self-signed certificate as described below.

1. Access the server with this URL:
   `https://gordon-1:8000/`

   NOTE
   Remember to start your URL with HTTPS, rather than HTTP. Otherwise, the browser will return an error.

2. The server responds with a **There is a problem with this website's security certificate** notification like this:



There is a problem with this website's security certificate.

The security certificate presented by this website was not issued by a trusted certificate authority.
The security certificate presented by this website was issued for a different website's address.

Security certificate problems may indicate an attempt to fool you or intercept any data you send to the server.

We recommend that you close this webpage and do not continue to this website.

Click here to close this webpage.
Continue to this website (not recommended).
More information

3. Click **Continue to this website (not recommended)**.
4. Enter your MarkLogic Server username and password at the prompt.

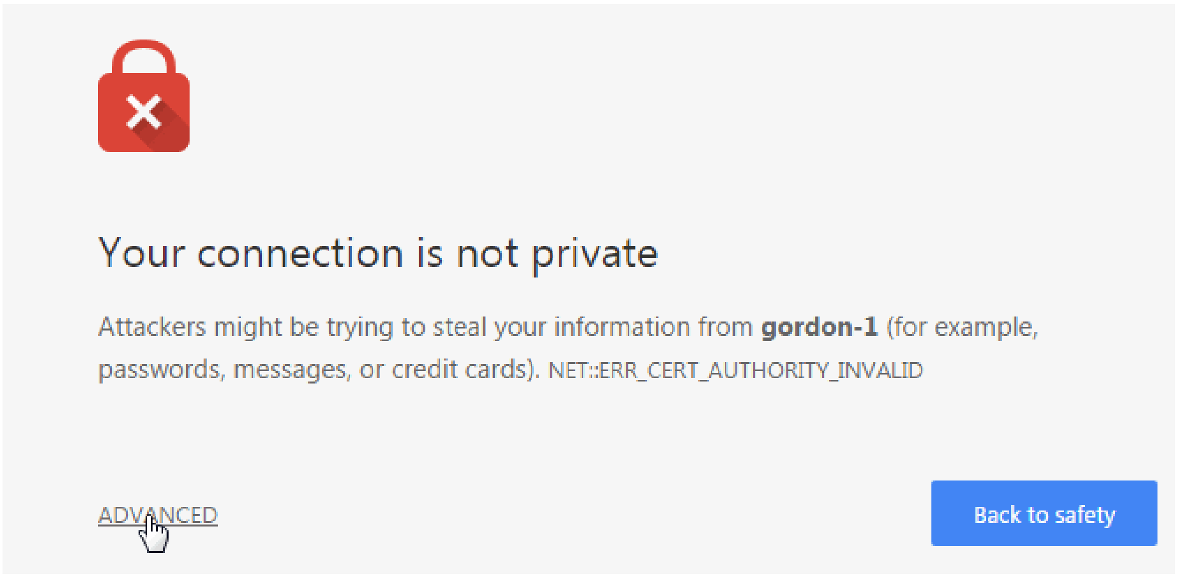
10.4.2. Creating a Security Exception in Google Chrome

If you have not imported the MarkLogic Server certificate authority into your Chrome browser, when you first access an SSL-enabled server, you will receive an error notifying you that you have accessed an untrusted server. You can bypass this security exception by accepting the certificate. For example, if you enable SSL on the HTTP server, App-Services, you can accept the self-signed certificate as described below.

1. Access the server with this URL:
   `https://gordon-1:8000/`

NOTE
Remember to start your URL with HTTPS, rather than HTTP. Otherwise, the browser will return an error.

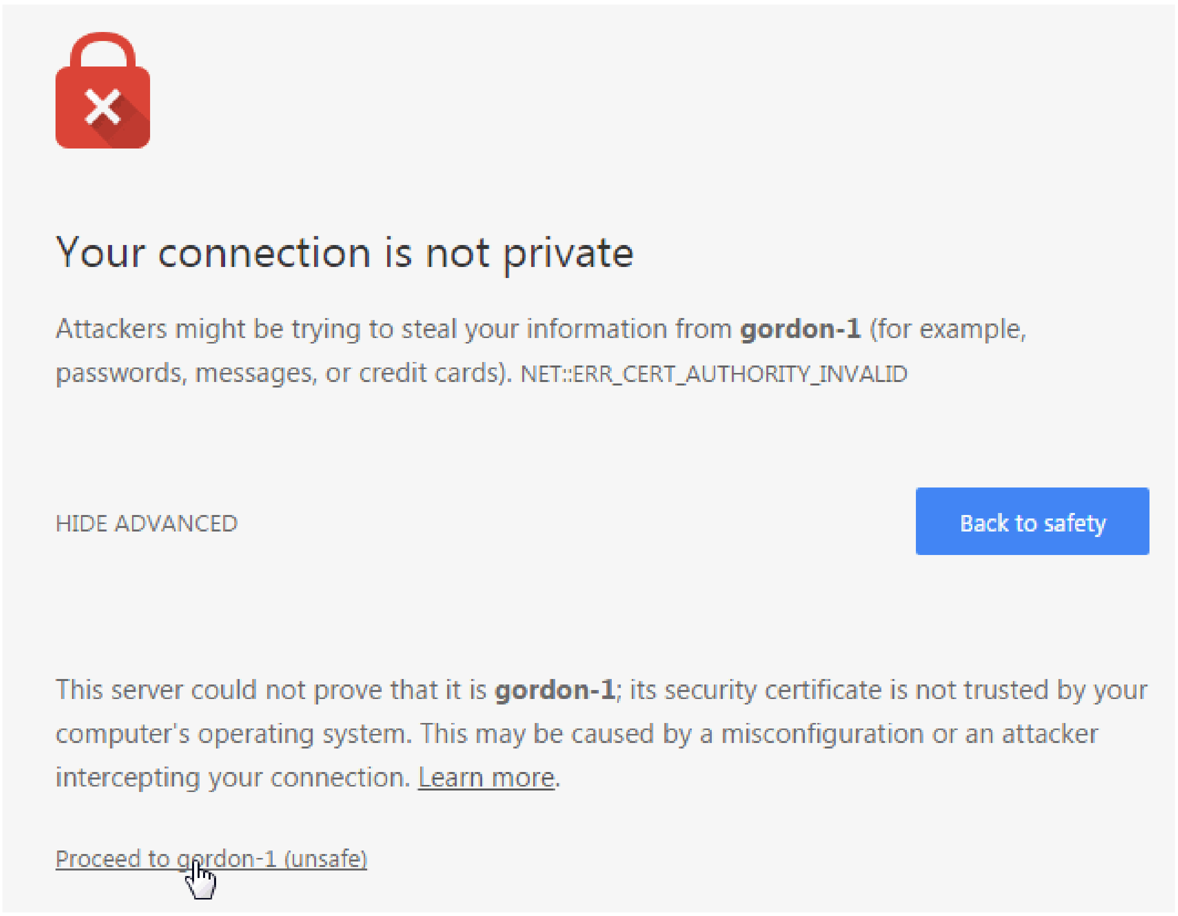
2. The server responds with a **Your connection is not private** notification like this:



Your connection is not private

Attackers might be trying to steal your information from **gordon-1** (for example, passwords, messages, or credit cards). NET::ERR_CERT_AUTHORITY_INVALID

3. Click **ADVANCED**.
4. At the bottom of the expanded window, select **Proceed to *hostname* (unsafe)**.



Your connection is not private

Attackers might be trying to steal your information from **gordon-1** (for example, passwords, messages, or credit cards). NET::ERR_CERT_AUTHORITY_INVALID

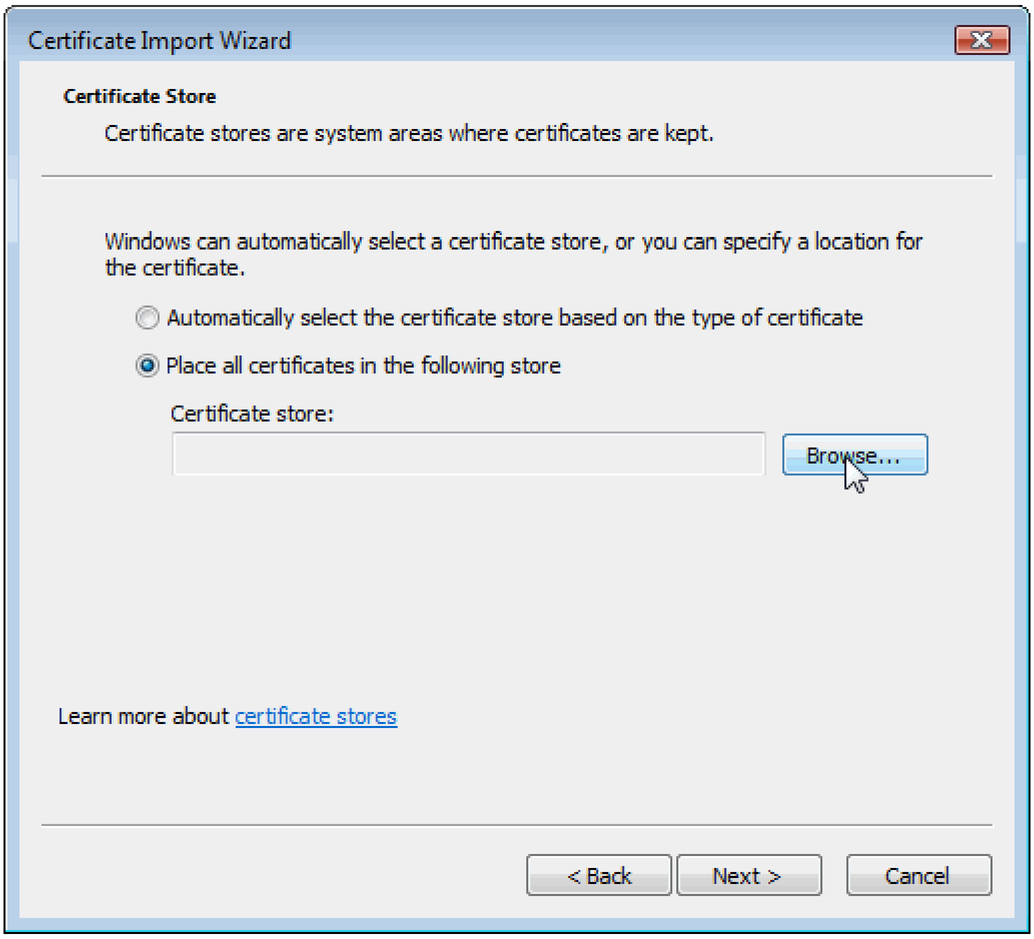
This server could not prove that it is **gordon-1**; its security certificate is not trusted by your computer's operating system. This may be caused by a misconfiguration or an attacker intercepting your connection. Learn more.

5. Enter your MarkLogic Server username and password at the prompt.

10.4.3. Importing a Self-Signed Certificate Authority into Windows

This section describes how to import the Certificate Authority into Windows for use by the Internet Explorer browser and WebDAV clients.

1. Open the Admin Interface.
2. Click **Security** in the left tree menu.
3. Click **Certificate Templates** in the left tree menu.
4. Click the certificate template name.
5. Click the **Status** tab to display the Certificate Template Status page.
6. Click **Import**.
7. Save or open the file.
8. In the Certificate window, click **Install Certificate**.
9. In the Certificate Import Wizard window, select **Place all certificates in the following store** and click **Browse**.



Certificate Import Wizard

Certificate Store

Certificate stores are system areas where certificates are kept.

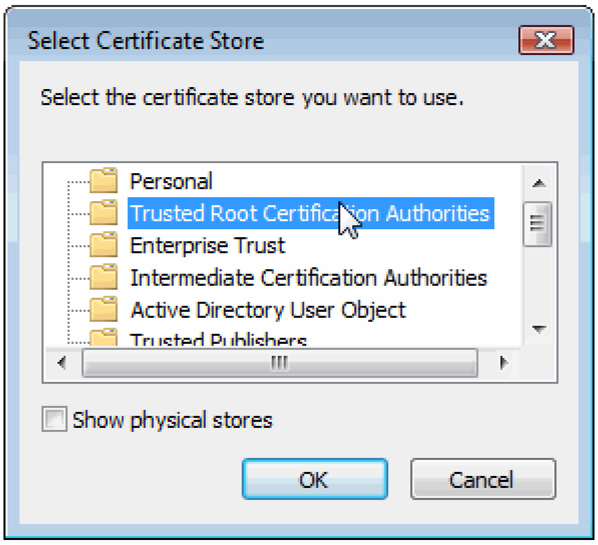
Windows can automatically select a certificate store, or you can specify a location for the certificate.

- ○ Automatically select the certificate store based on the type of certificate
- ● Place all certificates in the following store

Certificate store:

Learn more about certificate stores

10. In the Select Certificate Store window, select **Trusted Root Certification Authorities** and click **OK**.



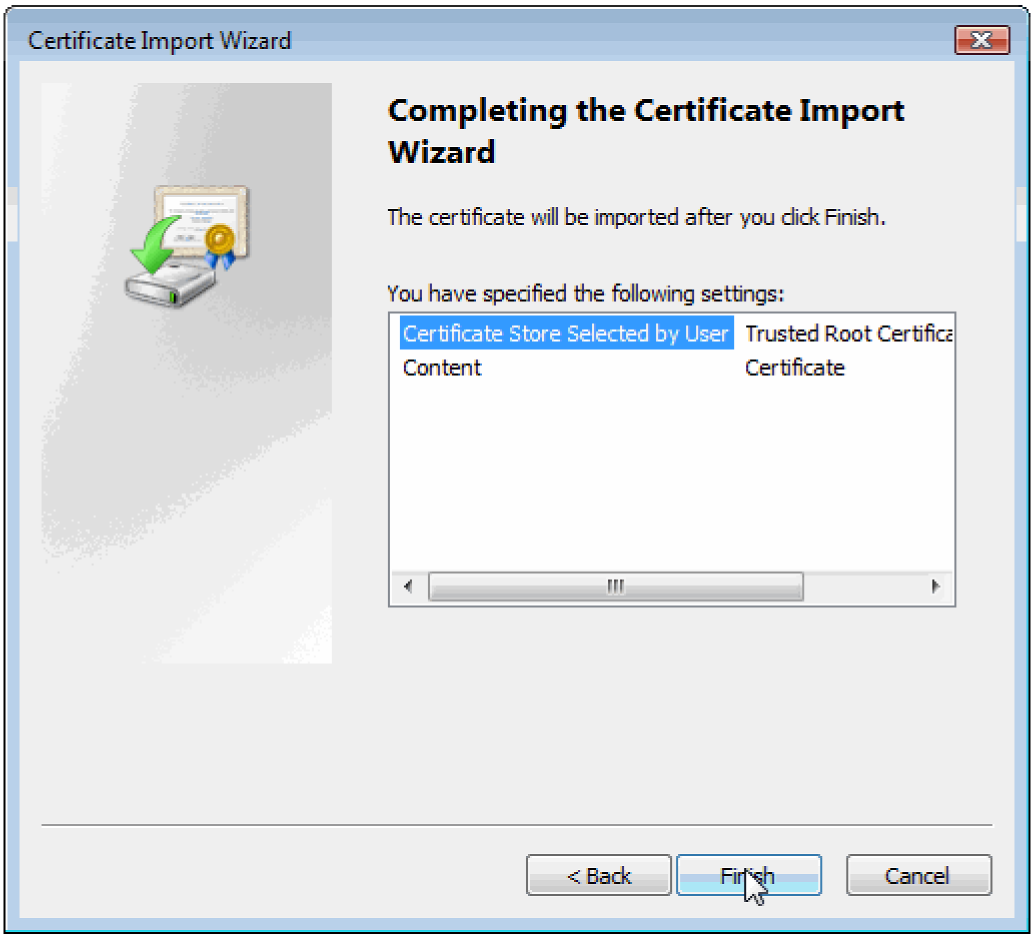
Select Certificate Store

Select the certificate store you want to use.

- Personal
- Trusted Root Certification Authorities
- Enterprise Trust
- Intermediate Certification Authorities
- Active Directory User Object
- Trusted Publishers

☐ Show physical stores

11. In the Certificate Import Wizard window, click **Next**.
12. On the Completing the Certificate Import Wizard page, select **Certificate Store Selected by User** and click **Finish**.



Certificate Import Wizard

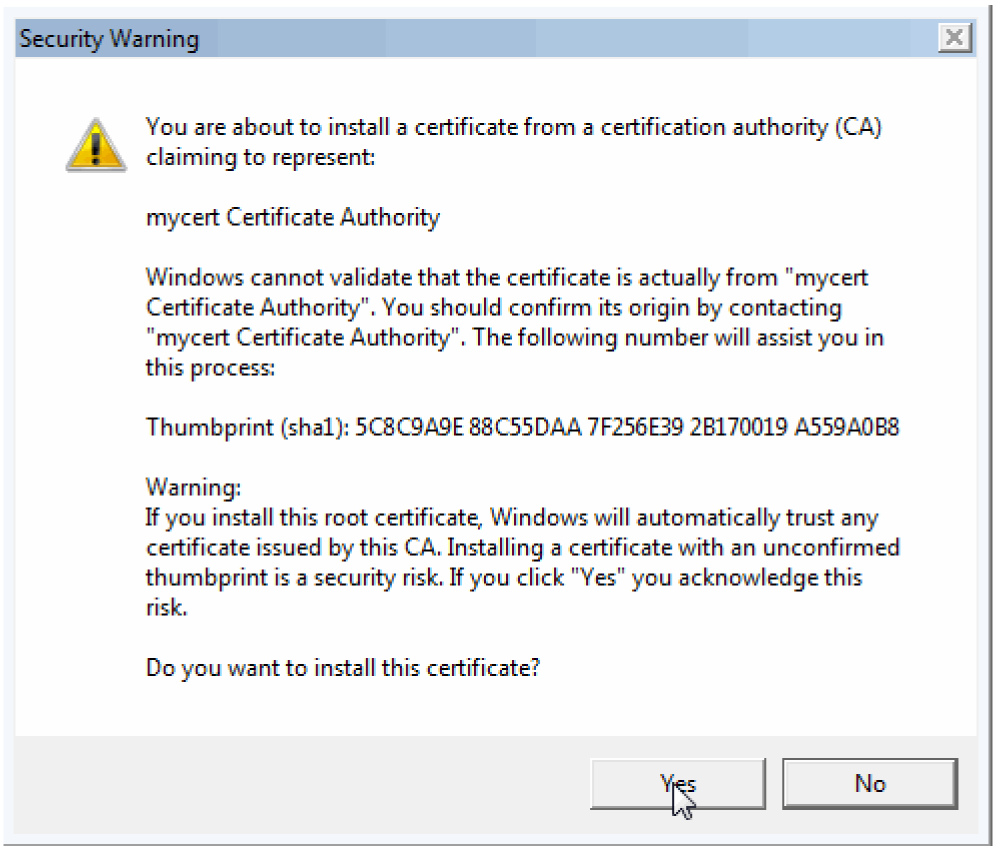
Completing the Certificate Import Wizard

The certificate will be imported after you click Finish.

You have specified the following settings:

Certificate Store Selected by User	Trusted Root Certifica
Content	Certificate

13. In the Security Warning page, click **Yes**.



You are about to install a certificate from a certification authority (CA) claiming to represent:

mycert Certificate Authority

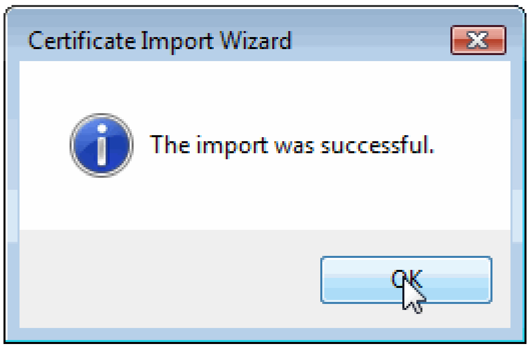
Windows cannot validate that the certificate is actually from "mycert Certificate Authority". You should confirm its origin by contacting "mycert Certificate Authority". The following number will assist you in this process:

Thumbprint (sha1): 5C8C9A9E 88C55DAA 7F256E39 2B170019 A559A0B8

Warning:
If you install this root certificate, Windows will automatically trust any certificate issued by this CA. Installing a certificate with an unconfirmed thumbprint is a security risk. If you click "Yes" you acknowledge this risk.

Do you want to install this certificate?

14. When you see **The import was successful prompt**, click **OK**.



The import was successful.

15. In the Certificate Information window, click **OK** to exit.

You should now be able to access the SSL-enabled server from your Internet Explorer browser or WebDAV client.

10.5. Procedures for Obtaining a Signed Certificate

This section describes how to obtain a signed certificate and import it into your server.

NOTE
No outside authority is used to sign certificates used between servers communicating over the internal XDQP connections in a cluster. Such certificates are self-signed and trusted by each server in the cluster. For details, see Enabling SSL communication over XDQP in *Administrating MarkLogic Server*.

10.5.1. Generating and Downloading Certificate Requests

Once the server is created or modified with SSL enabled, you can generate one or more PEM-encoded certificate requests.

NOTE
You must first assign the certificate template to an App Server, as described in Enabling SSL for an App Server, before you can generate a certificate request.

1. Click **Security** in the left tree menu.
2. Under **Security**, click **Certificate Templates** in the left tree menu.
3. Click the certificate template name. The **Configure** tab opens.
4. Click the **Request** tab. The **Generate Certificate Request** page appears:
5. Select either **All** or **Only those that are needed for missing, expired, temporary, or out of date certificates that are not already pending**.
6. Click **OK**.
7. The certificate template Status page will display. Click on **Download** to download the certificate request to your file system.
8. If the file does not already have a ‘zip’ extension, rename the file by replacing the existing file extension with ‘zip’.
9. Send the zip file containing the certificate requests to a Certificate Authority, such as Verisign.

10.5.2. Signing a Certificate with Your Own Certificate Authority

As an alternative to using a third-party Certificate Authority, you can create your own Certificate Authority, as described in Creating a Certificate Authority. You can then use this Certificate Authority to sign the certificate request using `pki:authority-sign-host-certificate-request()`.

Once signed, you can forward the signed certificate to any MarkLogic Server user, who can then import the signed certificate into their MarkLogic Server host, as described in Importing a Signed Certificate into MarkLogic Server.

For example, to request and sign a certificate from the `mycert` template created in Creating a Certificate Template, do the following:

```
xquery version "1.0-ml";

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

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

let $req :=
  pki:generate-certificate-request(
    pki:get-template-by-name("mcert")/pki:template-id,
    "ServerName", (), ())
let $cert :=
  pki:authority-sign-host-certificate-request(
    xdmp:credential-id("acme-ca"),
    xdmp:x509-request-extract($req),
    fn:current-dateTime(),
    fn:current-dateTime() + xs:dayTimeDuration("P365D"))

return xdmp:x509-certificate-extract($cert)
```

10.5.3. Importing a Signed Certificate into MarkLogic Server

When you receive the PEM file(s) containing signed certificate(s) from the certification authority, import the PEM file(s) into MarkLogic Server. If you are using chained certificates, you will need to import the end and intermediate certificate PEM files into MarkLogic Server. If your MarkLogic Server is to act as a client , you must also import the root certificate.

NOTE
Because the signed certificate is from a trusted certification authority, browsers are already configured to trust the certificate.

1. Click **Security** in the left tree menu.
2. In the left tree menu, under **Security**, click **Certificate Templates** .
3. Click the certificate template name The Configure tab displays.
4. Click the **Import** tab. The Import Certificates page displays:
5. Click on **Choose File** or **Browse** and locate the PEM file(s) containing the signed certificate(s) and select **OK** or **Open**. Zip files can be uploaded directly without the need to unzip them. Alternatively, you can paste an individual certificate(s) into the text area.

NOTE
Users can also import certificates signed by something besides a downloaded CSR from MarkLogic Server. To do this, the user also needs to import the private key of the certificate. The user can only input one certificate at a time if doing this.

10.6. Viewing Trusted Certificate Authorities

You can list all of the certificate authorities that are known to and trusted by the server in the Certificate Authority page. Each CA in the list links to the corresponding Certificate Authority page for that CA.

The Certificate Authority page provides detailed information on the CA, a list of revoked certificates, the option to manually revoke a certificate by ID, and the ability to delete the CA from the server.

1. Click **Security** in the left tree menu.
2. In the left tree menu, under **Security**, click **Certificate Authorities**.
3. The Certificate Authority Summary tab displays the list of trusted certificate authorities.
4. Click on an organization to see details of the certificate authority.

10.7. Importing a Certificate Revocation List into MarkLogic Server

A Certificate Revocation List (CRL) is a list of certificate serial numbers that have been revoked by a certificate authority. The CRL is signed by the certificate authority to verify its accuracy. The CRL contains the revocation date of each certificate, along with the date the CRL was published and the date it will next be published, which is useful in determining whether a newer CRL should be fetched.

You can use `pki:insert-certificate-revocation-list()` to import a CRL into the Security database. certificate authorities typically allow the CRL to be downloaded via HTTP. The document URL in the database is derived from the URL passed into the function, so Inserting a newer CRL retrieved from the same URL will replace the previous one in the database.

For example, the following script imports a PEM- or DER-encoded CRL from Verisign into the Security database:

```
xquery version "1.0-ml";
import module namespace pki = "http://marklogic.com/xdmp/pki"
         at "/MarkLogic/pki.xqy";
let $URI := "http://crl.verisign.com/pca3.crl"
return
    pki:insert-certificate-revocation-list(
         $URI,
         xdmp:document-get($URI)/binary() )
```

NOTE

If the next publication date of the CRL is earlier than the current time, you will receive the following message in the error log: `loadCertificateRevocationLists: Most recent CRL for issuer=<issuer_name> is expired`.

10.8. Deleting a Certificate Template

Deleting a template deletes all signed certificates and pending requests for the template. Before deleting a certificate template, ensure that a certificate with that name is not in use by a server. If a certificate with the same name as the certificate template is in use by a server, the delete operation returns an “Invalid input” error.

To delete an unused certificate template:

1. Click the **Security** icon in the left tree menu.
2. Click the **Certificate Templates** icon on the left tree menu.
3. Click the certificate template name on the left tree menu.
4. On the Certificate Template page, click **Delete**.
5. In the confirmation page, select **OK**.

11. Certificate-Based Authentication

Certificate-based user authentication allows users to log into MarkLogic Server without being required to enter a username and password. Certificate-based user authentication configuration can be achieved using either internal user or external name based user configurations.

11.1. User Certificate Example

There are few common steps/examples listed to add to clarity. In this example setup, the certificate presented by the App Server user (`demoUser1`) will be as follows.

```
Certificate:
        Data:
            Version: 1 (0x0)
            Serial Number: 7 (0x7)
        Signature Algorithm: sha1WithRSAEncryption
            Issuer: C=US, ST=CA, L=San Carlos, O=MarkLogic Corp.,
OU=Engineering, CN=MarkLogic DemoCA
            Validity
                Not Before: Jul 11 02:58:24 2017 GMT
                Not After : Aug 27 02:58:24 2019 GMT
            Subject: C=US, ST=CA, L=San Carlos, O=MarkLogic Corp.,
OU=Engineering, CN=demoUser1
            Subject Public Key Info:
                Public Key Algorithm: rsaEncryption
                    Public-Key: (1024 bit)
                    Modulus:
                        .....................
                    Exponent: 65537 (0x10001)
        Signature Algorithm: sha1WithRSAEncryption
```

11.2. CA Certificate (User Cert Signer) Import from Admin Interface

In order to allow MarkLogic Server to accept the Certificate presented by a user, MarkLogic Server needs a Certificate Authority (CA) to sign the user certificate installed into MarkLogic Server.

Install a CA certificate used to sign the demoUser1 certificate in the Admin Interface, as follows.

1. Click **Security** in the left tree menu.
2. In the left tree menu, under **Security**, click **Certificate Authorities**.
3. Click the **Import** tab and import a certificate, such as the one shown in the example below.

Example CA certificate:

```
Certificate:
        Data:
            Version: 3 (0x2)
            Serial Number: 9774683164744115905 (0x87a6a68cc29066c1)
        Signature Algorithm: sha256WithRSAEncryption
            Issuer: C=US, ST=CA, L=San Carlos, O=MarkLogic Corp.,
OU=Engineering, CN=MarkLogic DemoCA
            Validity
                Not Before: Jul 11 02:53:18 2017 GMT
                Not After : Jul  6 02:53:18 2037 GMT
            Subject: C=US, ST=CA, L=San Carlos, O=MarkLogic Corp.,
OU=Engineering, CN=MarkLogic DemoCA
            Subject Public Key Info:
                Public Key Algorithm: rsaEncryption
                    Public-Key: (4096 bit)
                    Modulus:
                       ......................
                    Exponent: 65537 (0x10001)
            X509v3 extensions:
                X509v3 Subject Key Identifier:
                    D9:45:B9:9A:DC:93:7B:DB:47:07:C6:96:63:57:13:A7:A8
:F1:D0:C8
                X509v3 Authority Key Identifier:
                    keyid:D9:45:B9:9A:DC:93:7B:DB:47:07:C6:96:63:57:13
:A7:A8:F1:D0:C8
                X509v3 Basic Constraints: critical
                    CA:TRUE
                X509v3 Key Usage: critical
                    Digital Signature, Certificate Sign, CRL Sign
        Signature Algorithm: sha256WithRSAEncryption
```

11.3. CA Certificate Import into MarkLogic Server from Query Console

You can also import the Certificate Authority by using `pki:insert-trusted-certificates()` to load the Trusted CA into the Security database in MarkLogic Server, as shown below.

NOTE

If using Query Console, make sure this query is executed against the Security database.

```
xquery version "1.0-ml";

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

pki:insert-trusted-certificates(
  xdmp:document-get("/OurCertificateLocation/DemoLabCA.pem",
  <options xmlns="xdmp:document-get">
    <format>text</format>
  </options>)
)
```

11.4. Certificate Template & Template CA Import into Client (Browser/SSL Client)

To enable SSL on the App Server, do either of the following:

- Create certificate template, as described in Creating a Certificate Template, to utilize Self Signed Certificate.
- Import a signed certificate into MarkLogic Server, as described in Importing a Signed Certificate into MarkLogic Server.

In both of the above cases, you must import the CA used to sign the certificate used by the MarkLogic Server SSL App Server into Client Browser/SSL Client, as described in Procedures for Obtaining a Signed Certificate or Importing a Self-Signed Certificate Authority into Windows.

After creating a certificate template, link the template with the App Server and enable SSL on the App Server.

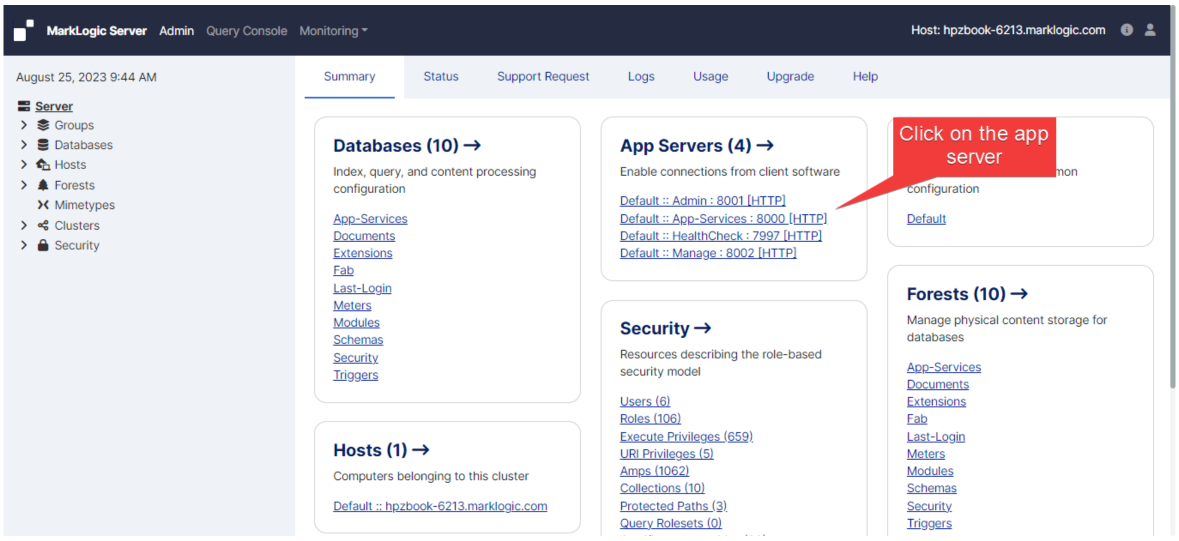
11.5. Creating a MarkLogic Server User to Use Certificate-Based Authentication

When creating an internal MarkLogic Server user to use certificate-based authentication, specify the user name as it appears in the `CN` value of the certificate `Subject` field (`demoUser1` in the example shown in User Certificate Example). When creating an external MarkLogic Server user to use certificate-based authentication, specify the external name as it appears in the whole certificate `Subject` field (`C=US,ST=CA,L=San Carlos,O=MarkLogic Corp.,OU=Engineering,CN=demoUser1` in the example shown in User Certificate Example).

11.5.1. Creating a MarkLogic Server User with an Internal Name

To configure certificate-based user authentication for user, `demoUser1`, as a MarkLogic Server internal user, follow these steps in the Admin Interface:

1. Click **Security** in the left tree menu.
2. Under **Security**, click **Users**.
3. Click the **Create** tab.
4. In the **User Name** field, enter the user name as it appears in the `CN` value of the certificate `Subject` field (`demoUser1` in the example shown in User Certificate Example)
5. Enter and confirm a password.
6. Click **OK**.
7. Next, change the app server configuration. At the top of the left tree menu, click **Server**.
8. Under the **Summary** tab, and the **App Servers** heading, click the name of the app server.



August 25, 2023 9:44 AM

Databases (10) →
Index, query, and content processing configuration
App-Services
Documents
Extensions
Fab
Last-Login
Meters
Modules
Schemas
Security
Triggers

Hosts (1) →
Computers belonging to this cluster
Default :: hpzbook-6213.marklogic.com

App Servers (4) →
Enable connections from client software
Default :: Admin : 8001 [HTTP]
Default :: App-Services : 8000 [HTTP]
Default :: HealthCheck : 7997 [HTTP]
Default :: Manage : 8002 [HTTP]

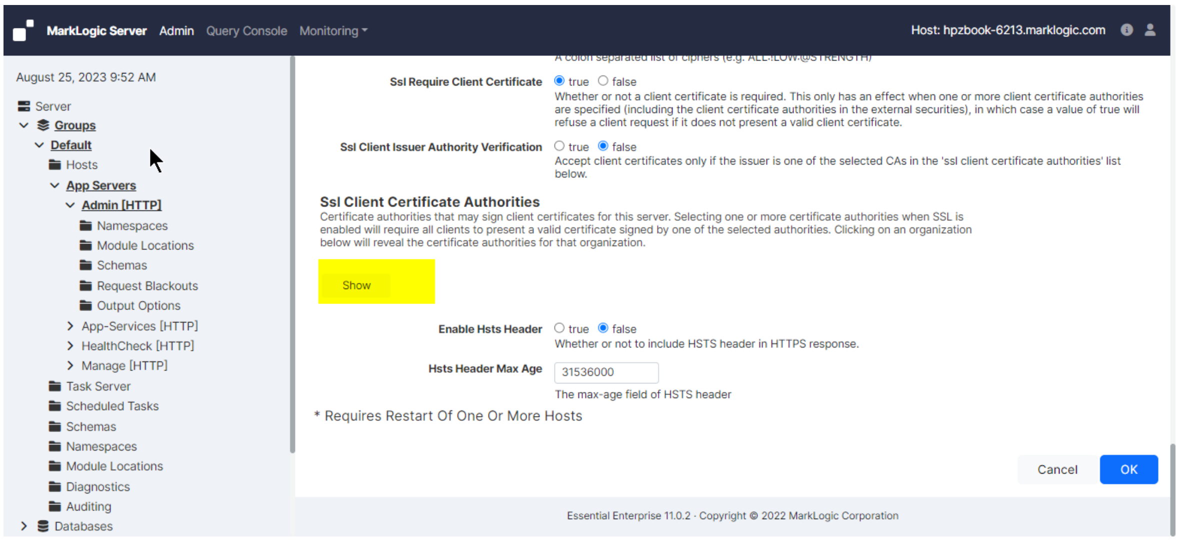
Security →
Resources describing the role-based security model
Users (6)
Roles (106)
Execute Privileges (659)
URI Privileges (5)
Amps (1062)
Collections (10)
Protected Paths (3)
Query Rolesets (0)

mon
configuration
Default

Forests (10) →
Manage physical content storage for databases
App-Services
Documents
Extensions
Fab
Last-Login
Meters
Modules
Schemas
Security
Triggers

9. On the **Configure** tab, set the **Authentication** field to **Certificate**.
10. Set **Internal Security** to **true**.
11. If you do not wish to have the user authenticated as an external user, be sure that **External Securities** are set to **None**,

12. Scroll down to the bottom of the page and in the **Ssl Client Certificate Authorities** section, click **Show**.



A colon separated list of ciphers (e.g. ALL:!LOW:@STRENGTH)

Ssl Require Client Certificate [✓ true] false
Whether or not a client certificate is required. This only has an effect when one or more client certificate authorities are specified (including the client certificate authorities in the external securities), in which case a value of true will refuse a client request if it does not present a valid client certificate.

Ssl Client Issuer Authority Verification true [✓ false]
Accept client certificates only if the issuer is one of the selected CAs in the 'ssl client certificate authorities' list below.

Ssl Client Certificate Authorities
Certificate authorities that may sign client certificates for this server. Selecting one or more certificate authorities when SSL is enabled will require all clients to present a valid certificate signed by one of the selected authorities. Clicking on an organization below will reveal the certificate authorities for that organization.

Enable Hsts Header true [✓ false]
Whether or not to include HSTS header in HTTPS response.

Hsts Header Max Age 31536000
The max-age field of HSTS header

* Requires Restart Of One Or More Hosts

Essential Enterprise 11.0.2 · Copyright © 2022 MarkLogic Corporation

13. Select the certificate authority created in CA Certificate (User Cert Signer) Import from Admin Interface to sign the client/user certificate.
14. Click **OK**.

Once configured, `demoUser1` is now able to access the App Server with a browser that has the user certificate installed, as described in Certificate Template & Template CA Import into Client (Browser/SSL Client).

NOTE
You will also need to assign the necessary roles to `demoUser1` to access the needed MarkLogic Server resources.

11.5.2. Creating a MarkLogic Server User with an External Name

To configure certificate-based user authentication for user, `newUser1`, as a MarkLogic Server user with an external name, follow these steps in the Admin Interface:

1. Click **Security** in the left tree menu.
2. Under **Security**, click **Users**.
3. Click the **Create** tab.
4. In the **User Name** field, enter `newUser1`.
5. Enter a password.
6. In the **Confirm Password** field, confirm the password.
7. In the **External Name** field, enter the entire **Subject** field from the example shown in User Certificate Example.
8. Click **OK**.
9. In the left tree menu. under **Security**, click **External Security**.
10. Click **Create**.
11. Enter a name for the security object in the **External Security Name** field.
12. From the **Authentication** field, select **certificate**.
13. Click **OK**.
14. Click **Server** in the left tree menu.

15. On the **Summary** tab, under **App Servers**, click the appropriate app server.
16. On the **Configure** tab, scroll down to the **External Securities** section.
17. From the drop-down, select the external security object you created in Step 11.
18. In the **SSL Client Certificate Authorities** section, click **Show**.
19. Select the CA certificate you configured in CA Certificate (User Cert Signer) Import from Admin Interface.
20. Click **OK**.

12. Secure Credentials

Secure credentials enable a security administrator to manage credentials, making them available to less privileged users for authentication to other systems without giving them access to the credentials themselves.

Secure credentials consist of a PEM-encoded X.509 certificate and private key and/or a username and password. Secure credentials are stored as secure documents in the Security database on MarkLogic Server, with passwords and private keys encrypted. A user references a credential by name and access is granted if the permissions stored within the credential document permit the access to the user. There is no way for a user to get access to the unencrypted credentials.

Secure credentials allow you to control which users have access to specific resources. A secure credential controls what URIs it may be used for, the type of authentication (e.g. digest), whether the credential can be used to sign other certificates, and the user role(s) needed to access the resource.

The security on a credential can be configured three different ways:

- Credentials that secure a resource by username and password.
- Credentials that secure a resource by a PEM-encoded X.509 certificate and a PEM-encoded private key.
- Credentials that secure a resource by username and password, as well as a PEM-encoded X.509 certificate and a PEM-encoded private key.

In most cases, the private key and X.509 certificate used to configure a secure credential are obtained from a trusted Certificate Authority. However, there may be situations in which you may want to create your own Certificate Authority and generate your own private key and certificate.

12.1. Creating a Secure Credential with Username and Password

This section describes how to use the Admin Interface to create a simple secure credential that grants access to a resource by means of a username and password.

1. In the Admin Interface, click **Security** in the left tree menu.
2. Under **Security**, click **Secure Credentials**.
3. Click **Create**.
4. In the **Credential Name** field, enter the name of the credential.
5. Optionally specify a credential description, username, and password.
6. Leave the **Credential Certificate** and **Credential Private Key** fields empty.
7. Set the **Credential Signing** field to **false**.
8. In the **Target Uri Pattern** field, enter the URIs of the MarkLogic Server App Servers this credential is to protect, starting with `https`.

   > **NOTE**
   > A role with read capability implies execute capability, as well.

9. In the **Target Authorization** field, select the authorization used by the target App Servers.
10. From the **Role** field, select the role required for a user to access the App Servers.
11. From the **Capability** field, select the permissions required for a user to access the App Servers using this credential.
12. Click **OK**.

12.2. Creating a Secure Credential with PEM-Encoded Public and Private Keys

You can skip this procedure if you have obtained a signed Certificate Authority (CA) from a trusted third party. In this case, you can paste the credential and private key into the Secure Credentials window described above in Creating a Secure Credential with Username and Password.

Generating a secure credential that includes PEM-encoded public and private keys is a two-step procedure that is best done in code.

12.2.1. Creating a Certificate Authority

Secure credentials that contain PEM-encoded public and private keys can be used to control access to a CA stored in a MarkLogic Server Security database. To create and insert a CA into the Security database, use `pki:create-authority()`.

For example, the following query creates a CA, named `acme-ca`:

```
xquery version "1.0-ml";

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

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

pki:create-authority(
  "acme-ca", "Acme Certificate Authority",
  element x509:subject {
    element x509:countryName            {"US"},
    element x509:stateOrProvinceName    {"California"},
    element x509:localityName           {"San Carlos"},
    element x509:organizationName       {"Acme Inc."},
    element x509:organizationalUnitName {"Engineering"},
    element x509:commonName             {"Acme CA"},
    element x509:emailAddress           {"ca@acme.com"}
  },
  fn:current-dateTime(),
  fn:current-dateTime() + xs:dayTimeDuration("P365D"),
  (xdmp:permission("admin","read")))
```

12.2.2. Creating Secure Credentials from a Certificate Authority

Once you have created a CA as described in Creating a Certificate Authority, you can use the CA to create a client certificate and private key to build a secure credential.

Use `pki:authority-create-client-certificate()` to create a client certificate with PEM-encoded public/private keys. Next, use `sec:create-credential()` to generate and insert the credential.

For example, to create a secure credential, named `acme-cred`, from the `acme-ca` CA that includes PEM-encoded public and private keys, a username and password, and that enables access to the target, `https://MLserver:8010/.*`, do the following:

```
xquery version "1.0-ml";

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

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

let $tmp :=
  pki:authority-create-client-certificate(
    xdmp:credential-id("acme-ca"),
    element x509:subject {
      element x509:countryName            {"US"},
      element x509:stateOrProvinceName    {"California"},
      element x509:localityName           {"San Carlos"},
      element x509:organizationName       {"Acme Inc."},
      element x509:organizationalUnitName {"Engineering"},
      element x509:commonName             {"Elmer Fudd"},
      element x509:emailAddress           {"elmer.fudd@acme.com"}
    },
    fn:current-dateTime(),
    fn:current-dateTime() + xs:dayTimeDuration("P365D"))

let $cert := $tmp[1]
let $privkey := $tmp[2]

return  sec:create-credential(
    "acme-cred", "A credential with user/password and certificate",
    "admin", "admin", $cert, $privkey,
    fn:false(),
    sec:uri-credential-target("https://MLserver:8010/.*", "digest"),
    xdmp:permission("admin","read"))
```

To create a secure credential, named `simple-cred`, that uses only a username and password, do the following:

```
xquery version "1.0-ml";

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

sec:create-credential(
    "simple-cred", "A simple credential without a certificate",
    "admin", "admin", (), (),
    fn:false(),
    sec:uri-credential-target("https://MLserver:8010/.*", "digest"),
    xdmp:permission("admin","read"))
```

As described in Configuring SSL on App Servers, MarkLogic Server App Servers authenticate clients by means of a host certificate associated with a certificate template. The following example shows how to create a host certificate using the CA described in Creating a Certificate Authority and import it into the `myTemplate` certificate template. For details on how to create a certificate template, see Creating a Certificate Template.

```
xquery version "1.0-ml";

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

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

let $tmp :=
  pki:authority-create-host-certificate(
    xdmp:credential-id("acme-ca"),
    element x509:subject {
      element x509:countryName            {"US"},
      element x509:stateOrProvinceName    {"California"},
      element x509:localityName           {"San Carlos"},
      element x509:organizationName       {"Acme Inc."},
      element x509:organizationalUnitName {"Engineering"},
      element x509:commonName             {"MLserver.marklogic.com"},
      element x509:emailAddress           {"me@marklogic.com"}
    },
    fn:current-dateTime(),
    fn:current-dateTime() + xs:dayTimeDuration("P365D"),
    "www.eng.acme.com", "1.2.3.4")

let $template := pki:template-get-id(
                    pki:get-template-by-name("myTemplate"))
let $cert := $tmp[1]
let $privkey := $tmp[2]

return pki:insert-host-certificate($template, $cert, $privkey)
```

13. External Security

NOTE
We recommend that you configure MarkLogic Server to authenticate through external security. Moving the authentication process to an external system makes it more difficult for threat actors to access MarkLogic Server.

MarkLogic Server allows you to **authenticate** user credentials through these **external agents**:

- Certificate Authorities
- Kerberos Servers
- LDAP Servers
- SAML Identity Providers
- OAuth Identity Providers

External agents are third-party systems that serve as centralized points of authentication, **authorization**, or both. You can configure MarkLogic Server and your app servers to authenticate through any number of these external agents. Meanwhile, a user needs to successfully authenticate through only one of them to gain access. That external agent then provides the user information that MarkLogic Server needs to authorize that user, either **internally** or **externally**.

13.1. Terms Used in This Section

GENERAL TO EXTERNAL SECURITY:

- **Authentication**: The process of verifying user credentials such as username-and-password combinations, certificates, tickets, or tokens with an **external agent** then associating the current session with that authenticated user. However, it does not grant any access or authority to perform any actions on the system. Authentication can be done either internally through **internal users** or externally through an **external agent**.
- **Authorization**: The process of assigning a pre-configured role to the authenticated user. Roles define what an authenticated user is allowed to do on MarkLogic Server. Once a user is externally authenticated, then MarkLogic Server matches the user information that the **external agent** provides to either an **internal user** (**internal authorization**) or role (**external authorization**).
- **External agent**: A third-party user information provider. External agents can authenticate user credentials across multiple servers and applications.
- **External authorization**: The process by which MarkLogic Server searches the Security database for a role whose **external name** matches certain information extracted from the **external agent**'s authentication response. Occurs when the **authorization** field within the **external security object** is set to an option other than `internal`.
- **External name**: A string value that MarkLogic Server uses during **authorization** to match information from the external agent's authentication response to the proper user (**internal authorization**) or role (**external authorization**).
- **External security object**: A named configuration that you create to assign to an app server. This object specifies which **authentication** protocol and **authorization** scheme that app server should use, along with any other parameters necessary for integration with the **external agent**. Multiple app servers can use the same object.
- **External security name**: The name that you give to the **external security object** when you configure it. You use this name when you assign a particular **external security object** to an app server.

- **External user**: A user that exists in an **external agent**’s system rather than in the MarkLogic Server Security Database.
- **Internal authorization**: The process by which MarkLogic Server searches the Security database for a user whose **external name** matches certain information extracted from the **external agent**’s authentication response. Occurs when the **authorization** field within the **external security object** is set to `internal`.
- **Internal user**: A user that exists in the MarkLogic Server Security Database.
- **Temporary user**: An in-memory user that MarkLogic Server temporarily creates for **external authorization**. Its username and roles depend upon how the **external security object** and **external name** are configured.

SPECIFIC TO AUTHENTICATION METHODS:

- **Lightweight Directory Access Protocol (LDAP)**: An authentication protocol for accessing server resources over an internet or intranet network. An LDAP server provides a centralized user database where one password can be used to authenticate a user for access to multiple servers in the network. LDAP is supported on Active Directory on Windows Server 2008 and OpenLDAP 2.4 on Linux and other Unix platforms.

  NOTE
  If running MarkLogic Server on Windows and using LDAP authentication to authenticate users, the username must include the domain name in this form: `userName@domainName`.

- **Kerberos**: A ticket-based authentication protocol for trusted hosts on untrusted networks. Kerberos provides users with encrypted tickets that can be used to request access to particular servers. Because Kerberos uses tickets, both the user and the server can verify each other's identity and user passwords do not have to pass through the network.

  NOTE
  When application-level authentication is enabled with Kerberos authentication, an application can use `xdmp:gss-server-negotiate()` to obtain a username that can be passed to `xdmp:login()` to log into MarkLogic Server.

- **Distinguished Name (DN)**: A sequence of *Relative Distinguished Names* (RDNs), which are attributes with associated values expressed by the form *attribute=value*. Each RDN is separated by a comma in a DN. For example, to identify the user, `joe`, as having access to the server `MARKLOGIC1.COM`, the DN for `joe` would look like this: `UID=joe,CN=Users,DC=MARKLOGIC1,DC=COM`.

  NOTE
  The attributes after `UID` make up what is known as the *Base DN*.

  For details on LDAP DNs, see http://www.rfc-editor.org/rfc/rfc4514.txt.
- **Principal**: A unique identity to which Kerberos can assign tickets. For example, in Kerberos, a user is a principal that consists of a username and a server resource, described as a *realm*. Each user or service that participates in a Kerberos authentication realm must have a principal defined in the Kerberos database.

  A user principal is defined by the format: `username@REALM.NAME`. For example, to identify the user, `joe`, as having access to the server `MARKLOGIC1.COM`, the principal might look like this: `joe@MARKLOGIC1.COM`.
  For details on Kerberos principals, see http://www.kerberos.org/software/tutorial.html#1.3.2.

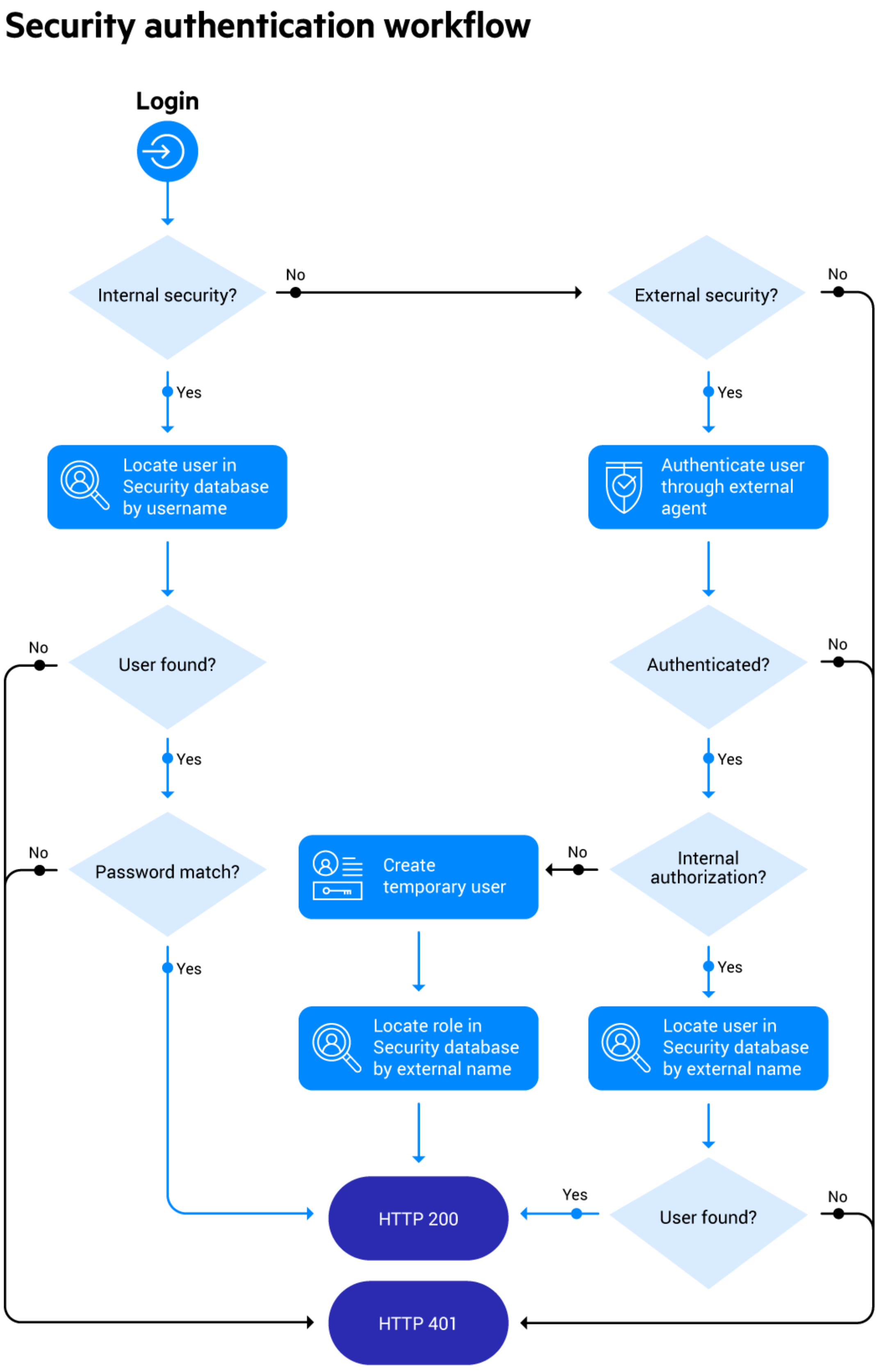
- **Certificate Authentication**: An authentication method that enables HTTPS clients to authenticate themselves to MarkLogic Server via a client certificate, either in addition to, or instead of, a password.
- **SAML (Security Assertion Markup Language)**: An authentication method that defines a Principal (such as a user), an Identity Provider (IDP), and a Service Provider (SP). In this scheme, the Principal requests a service from the Service Provider, which accesses the Identity Provider to authorize the Principal. MarkLogic Server supports SAML, version 2.0.

  NOTE
  MarkLogic Server currently only supports SOAP binding over HTTPS.

- **SAML Entity**: An XML document located in the MarkLogic Server Security database that serves as the SAML Identity Provider.

13.2. How MarkLogic Server Authenticates and Authorizes a User

This flowchart illustrates the logic that MarkLogic Server follows to determine how to authenticate an internal user and how to authenticate then authorize an external user:


Security authentication workflow
Login
Internal security?
No
External security?
No
Yes
Yes
Locate user in Security database by username
Authenticate user through external agent
No
User found?
Authenticated?
No
Yes
Yes
No
Password match?
Create temporary user
No
Internal authorization?
Yes
Yes
Locate role in Security database by external name
Locate user in Security database by external name
HTTP 200
Yes
User found?
No
HTTP 401

Successful Internal User Authentication Flow

When a user logs into an app server, MarkLogic Server checks if the app server is set for internal security. If so, MarkLogic Server searches the Security database for a matching username. If one is found, and the passwords match, then the internal user is authenticated. Authentication is complete.

Successful External User Authentication Flow

If the app server is configured for external security, then MarkLogic Server passes the login information to the configured external agent. The external agent sends a response containing information about the matching user. Authentication is complete, but authorization is also required.

Successful Internal Authorization Flow

If external security is configured for internal authorization, then MarkLogic Server extracts certain information from the authentication response and finds an internal user whose external name matches that information. Authorization is complete.

Successful External Authorization Flow

If the external security object is configured for external authorization, then MarkLogic Server creates a temporary user. MarkLogic Server then extracts certain information from the authentication response and finds any roles whose external names match that information. Finally, MarkLogic Server assigns each matching role to the temporary user. Authorization is complete.

13.3. Configuring MarkLogic Server for External Security

External security happens in three major steps:

1. MarkLogic Server authenticates an app server user by communicating with an external agent through information stored in the external security object.
2. Once the external agent responds with the information that identifies the user, MarkLogic Server extracts from it values to match to external names.
3. MarkLogic Server authorizes the user by matching those external names to either internal users or roles.

To configure MarkLogic Server for external security, follow this outline:

1. Sign up with the external agent of your choice. How to choose one and sign up is beyond the scope of this guide.
2. Create your external security object with the information gleaned from what your external agent provided when you signed up. This information includes attribute names, field names, and other connection details required for the integration and translation of the authentication response. You can create an external security object through one of these methods:
   - Admin Interface
   - `sec:create-external-security()`
   - `POST /manage/v2/external-security`
3. Configure your app servers to use your desired authentication type and external security object.
4. Assign attributes or field values from the external agent as external names:
   - Internal authorization: Assign external names to relevant users through one of these methods:
     - Admin Interface
     - `sec:user-set-external-names()`
     - `sec:create-user()` with the `external-names` parameter
     - `POST /manage/v2/users`
     - `PUT /manage/v2/users/[id-or-name]/properties`
   - External authorization: Assign external names to relevant roles through one of these methods:
     - Admin Interface
     - `sec:role-set-external-names()`

- `sec:create-role()` with the `external-names` parameter
- `POST /manage/v2/roles`
- `PUT /manage/v2/roles/[id-or-name]/properties`

NOTE
All examples in this section were run on a freshly installed instance of MarkLogic Server.

13.4. Defining and Inserting a SAML Entity

SAML authorization is done by means of a SAML entity stored in the MarkLogic Server Security database.

The SAML 2.0 specification provides a standard format for describing a SAML entity. The SAML specification provides for a variety of elements that can be defined in an entity, but only the `AttributeAuthorityDescriptor` element is used by MarkLogic Server. The SAML spec is located at this URL:

`http://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf`

The SAML entity defines an `entityID` in the form of a URL. To make use of a SAML entity, specify its entity ID URL in the “saml entity id” field in the external security configuration.

MarkLogic Server only supports the SAML 2.0 SOAP binding over HTTP. If multiple `AttributeService` elements are specified in the entity, one will be chosen at random. This allows support for multiple hosts in a cluster to be specified when no load balancer is used.

NOTE
The `saml-entity-insert` is only needed for SAML authorization without SAML authentication (for example LDAP authentication and SAML authorization). This is not a common use case. The common use case would be SAML authentication and SAML authorization. There is no Admin Interface mapping for `saml-entity-insert`.

You do not need to use `saml-entity-insert`. To use SAML, you only need `create-external-security` or to use the Admin Interface to configure it.

Use `sec:saml-entity-insert()` to insert the SAML entity into the MarkLogic Server Security database. For example, to insert a SAML entity, identified as `http://example.com/example`, that uses an encoded certificate for authorization, enter:

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

declare namespace md="urn:oasis:names:tc:SAML:2.0:metadata";
declare namespace ds="http://www.w3.org/2000/09/xmldsig#";

sec:saml-entity-insert(
<md:EntityDescriptor entityID="http://example.com/example">
  <md:AttributeAuthorityDescriptor
    protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
    <md:KeyDescriptor>
      <ds:KeyInfo>
        <ds:X509Data>
          <ds:X509Certificate>
MIID+TCCAeGgAwIBAgIJAImAkE0o79czMA0GCSqGSIb3DQEBCwUAMDwxEjAQBgNV
BAoMCUFjbWUgSW5jLjEmMCQGA1UEAwwdQWNtZSBJbmMuIE9wZXJhdGlvbnMgRGly
ZWN0b3IwHhcNMTcwMTA5MjE0MDE0WhcNMjcwMTA3MjE0MDE0WjA8MRIwEAYDVQQK
DAlBY21lIEluYy4xJjAkBgNVBAMMHU9wc0RpciBNYW5hZ2VkIENsdXN0ZXIgQWNj
ZXNzMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwu4iOujPFrkltDel
XgNl1BO/Xbcu6SEWnGCh3yGMwETqx1PnYDlueRuXIrZAHj8FFoKICJIwsARhcixM
ia2vDH0EkZPFGhb0shf0NEt7glDf1uaUava2x2jNXo5YUuiGDUhES50H3A0HS0Nz
WO0TIMaCu1vCTh5IHnKUnQB2MWrNGeb0I3RxOpqhRp6HarTb1u0mQN1iyiQox+pi
67Wh+eZ1313RTQBv8oavJFKHPT6JQK0rOVDXGDez/VajiUJswFNGZ2MgpVxqCDu3
iA+fdTV3TFp8XGYTPYCQgri5OKC9cGmFXzDgIiXqJLR8iAGbQT8YWsCzTzpYtTVN
JnqN/QIDAQABMA0GCSqGSIb3DQEBCwUAA4ICAQDPgcmLCl4kQFp15cfEKuI0QguC
vlCMjaZDDAr86IUDVJkVfm3Ytkw/QswI4ghZkbPuEhRf4SCo37OSR3++sPmMu5MR
gFtsU/UWGm6xXmIrBl/bkK+wmUwrW3DCcZQLZGOTG4o0tXSX+gGlvip5swpBTf5T
BsxJ3Hu479R48fTMIjoJ2gnVvZQ7aqnDqcZkifEskY6E7v431W1GEgccf0EJggnz
eRcTWfReYNy/foKKFuPW5MFYLd6RHOyWxgqJ3Nvroqu6xegVSQYJloJprZhhHx2H
NLZcBNYcgu2RgWNq9Pdjswxn3P1rRjch9YjgzZyjWywQpX+aASpPT2m0ONDYbkWK
V6YZmZbTmDDmwVfR4SK5GB93oxdZ647SfJwVsqN2qyKEDl/P2qwSY1iN851PhXAh
WMEyHfMgPTP22LHyYfQa+ExN5hpD95az+ZBdx+1CTO/9fJmQXvrmD1bNdbpfeKBD
YIv+yyL3UDtKQcMhp8zumt2XYJNAzSMhLkAMe2P7/i+47f5lXiGtrRuDVPyNzddB
VD2cQvB3JvQ7YRmt6BJPFmtuGSlx65d0fN7D3M8I5xtDa3XkmrrivcgOKi7DRSzE
bUu4cwfg7mWFJFDkWNWtIzqeni8658yLuEEgyFBUeW9OVjR2caTUZcSIObD2yvq7
o1oZlzTJxNplg99CCA==
          </ds:X509Certificate>
        </ds:X509Data>
      </ds:KeyInfo>
    </md:KeyDescriptor>
    <md:AttributeService
        Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
        Location="https://ML1:8005/SAML2/SOAP/AttributeQuery"/>
  </md:AttributeAuthorityDescriptor>
</md:EntityDescriptor>
)
```

13.5. Creating a Kerberos Keytab File

If you are configured for Kerberos authentication, then you must create a `services.keytab` file and place it in the MarkLogic Server data directory.

NOTE

The name of the generated keytab file must be `services.keytab`.

13.5.1. Creating a Keytab File on Windows

On Windows platforms, the `services.keytab` file is created using Active Directory Domain Services (AD DS) on a Windows server.

NOTE

If you are using the MD5 bind method and Active Directory Domain Services (AD DS) on a computer that is running Windows Server 2008 or Windows Server 2008 R2, be sure that you have installed the hot fix described in http://support.microsoft.com/kb/975697.

To create a `services.keytab` file, do the following:

1. Using Active Directory Domain Services on the Windows server, create a “user” with the same name as the MarkLogic Server hostname. For example, if the MarkLogic Server is named `mysrvr.marklogic.com`, create a user with the name `mysrvr.marklogic.com`.
2. Create a keytab file with the principal `HTTP/`*hostname* using `ktpass` command of the form:

   ```
   ktpass princ HTTP/<hostname> mapuser <user-account> pass <password>
   out <filename>
   ```

   For example, to create a keytab file for the host named `mysrvr.marklogic.com`, do the following:

   ```
   ktpass princ HTTP/mysrvr.marklogic.com@MLTEST1.LOCAL
   mapuser mysrvr.marklogic.com@MLTEST1.LOCAL pass mysecret
   out services.keytab
   ```
3. Copy the `services.keytab` from the Windows server to the MarkLogic Server data directory on your MarkLogic Server.

13.5.2. Creating a Keytab File on Linux

On Linux platforms, the `services.keytab` file is created as follows:

1. In a shell window, use `kadmin.local` to start the Kerberos administration command-line tool.
2. Use the `addprinc` command to add the principal to Kerberos.
3. Use the `ktadd` command to generate the `services.keytab` file for the principal.
   For example, to create a `services.keytab` file for the host named `mysrvr.marklogic.com`, do the following:

   ```
   $ kadmin.local
   > addprinc -randkey HTTP/mysrvr.marklogic.com
   > ktadd -k services.keytab HTTP/mysrvr.marklogic.com
   ```
4. Copy the `services.keytab` from the Linux Kerberos server to the MarkLogic Server data directory on your MarkLogic Server.

13.6. External Certificate User Authentication

MarkLogic Server 9 includes certificate-based user authentication, which allows users to log into MarkLogic Server without being required to enter a username and password. Previously certificates were only utilized to restrict client access to MarkLogic Server with the Digest/Basic User Authentication Scheme. Certificate-based user authentication configuration can be achieved based user configurations using either an internal user or external name.

13.6.1. Certificate Authentication Based on Internal User vs External Name

The difference between authentication based on an internal user or external name lies in the existence of the Certificate CN field-based user (`demoUser1` in the following example) in the MarkLogic Security database (internal user) versus if the user retrieved from Certificate Subject field (the whole Subject field as DN) is mapped as external name value in any existing user.

User Certificate Examples

Here are a few common examples, shown for clarity.

For the examples, the certificate presented by the App Server User (`demoUser1`) is the following.

```
$ openssl x509 -in UserCert.pem -text -noout
    Certificate:
        Data:
            Version: 1 (0x0)
            Serial Number: 7 (0x7)
        Signature Algorithm: sha1WithRSAEncryption
            Issuer: C=US, ST=NY, L=New York, O=MarkLogic Corporation,
OU=Engineering, CN=MarkLogic DemoCA
            Validity
                Not Before: Jul 11 02:58:24 2017 GMT
                Not After : Aug 27 02:58:24 2019 GMT
            Subject: C=US, ST=NJ, L=Princeton, O=MarkLogic Corporation,
OU=Engineering, CN=demoUser1
            Subject Public Key Info:
                Public Key Algorithm: rsaEncryption
                    Public-Key: (1024 bit)
                    Modulus:
                        ....................
                    Exponent: 65537 (0x10001)
        Signature Algorithm: sha1WithRSAEncryption
```

13.6.2. CA Certificate (User Cert Signer) Import from Admin Interface

To allow MarkLogic Server to accept the certificate presented by a user, MarkLogic Server needs a Certificate Authority (CA) to sign the user certificate installed into MarkLogic Server. You can install a CA Certificate (below) to be used to sign `demoUser1` Cert through the Admin Interface.

Click **Configure** in the left tree menu of the Admin Interface, then click **Security** to expand the options. Click Certificate Authorities, and then click the Import tab.

Paste this text for the trusted certificate into the field:

```
$ openssl x509 -in CACert.pem -text -noout
    Certificate:
        Data:
            Version: 3 (0x2)
            Serial Number: 9774683164744115905 (0x87a6a68cc29066c1)
        Signature Algorithm: sha256WithRSAEncryption
            Issuer: C=US, ST=NY, L=New York, O=MarkLogic Corporation, OU=Engineering,
CN=MarkLogic DemoCA
            Validity
                Not Before: Jul 11 02:53:18 2017 GMT
                Not After : Jul  6 02:53:18 2037 GMT
            Subject: C=US, ST=NY, L=New York, O=MarkLogic Corporation, OU=Engineering,
CN=MarkLogic DemoCA
            Subject Public Key Info:
                Public Key Algorithm: rsaEncryption
                    Public-Key: (4096 bit)
                    Modulus:
                       ......................
                    Exponent: 65537 (0x10001)
            X509v3 extensions:
                X509v3 Subject Key Identifier:
                    D9:45:B9:9A:DC:93:7B:DB:47:07:C6:96:63:57:13:A7:A8:F1:D0:C8
                X509v3 Authority Key Identifier:
                    keyid:D9:45:B9:9A:DC:93:7B:DB:47:07:C6:96:63:57:13:A7:A8:F1:D0:C8
                X509v3 Basic Constraints: critical
                    CA:TRUE
                X509v3 Key Usage: critical
                    Digital Signature, Certificate Sign, CRL Sign
        Signature Algorithm: sha256WithRSAEncryption
```

13.6.3. CA Certificate Import into MarkLogic Server from Query Console

You can also import the Certificate Authority using an XQuery call (`pki:insert-trusted-certificates`) to load the Trusted CA into MarkLogic Server.

This sample Query Console code demonstrates this process.

```
xquery version "1.0-ml";

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

pki:insert-trusted-certificates(
  xdmp:document-get("/OurCertificateLocation/DemoLabCA.pem",
  <options xmlns="xdmp:document-get">
    <format>text</format>
  </options>)
)
```

Be sure that this query is executed against the Security database. (The query is `Import_Trusted_CA.xqy` hosted by GitHub.)

13.6.4. Certificate Template & Template CA Import into Client (Browser/SSL Client)

To enable the SSL app server, do either of the following:

- Create a Certificate Template to utilize a Self-Signed Certificate.
- Import a pre-signed Certificate into MarkLogic Server.

In both of the above cases, you will need to import the Certificate Authority used to sign the certificate used by MarkLogic Server SSL app server into Client Browser/SSL. For example:

- Importing a Self-Signed Certificate Authority into Windows

Once template is created, you can link your Template with your app server to enable the SSL-based app server.

13.6.5. Certificate CN as Internal User vs External Name-Based Internal User

The difference between the two options lies in if the Certificate CN field User (`demoUser1` in our example) exists in the MarkLogic Server Security database as an internal user versus if the user retrieved from the Certificate Subject field is mapped as an external name to any existing user.

Certificate CN Field Value as MarkLogic Server Security Database Internal User

Follow these steps to configure Certificate-Based User Authentication for the user (`demoUser1`) as a MarkLogic Server internal user:

1. Create the user `demoUser1` with the necessary roles in the MarkLogic Server Security database (Internal User).
2. On the AppServer page, set the authentication schema to “Certificate” with Internal Security to set to “true”. Unless you want to have some users authenticated as an external user as well, leave external security object to “none”.
3. The app server will also select the CA that will be used to sign Client/User Certificate as accepted Certificate Authorities (See section CA Certificate earlier for example).

Once configured, accessing the app server with a browser the has the User Certificate (`demoUser1`) installed will be able to log into MarkLogic Server with the internal `demoUser1`.

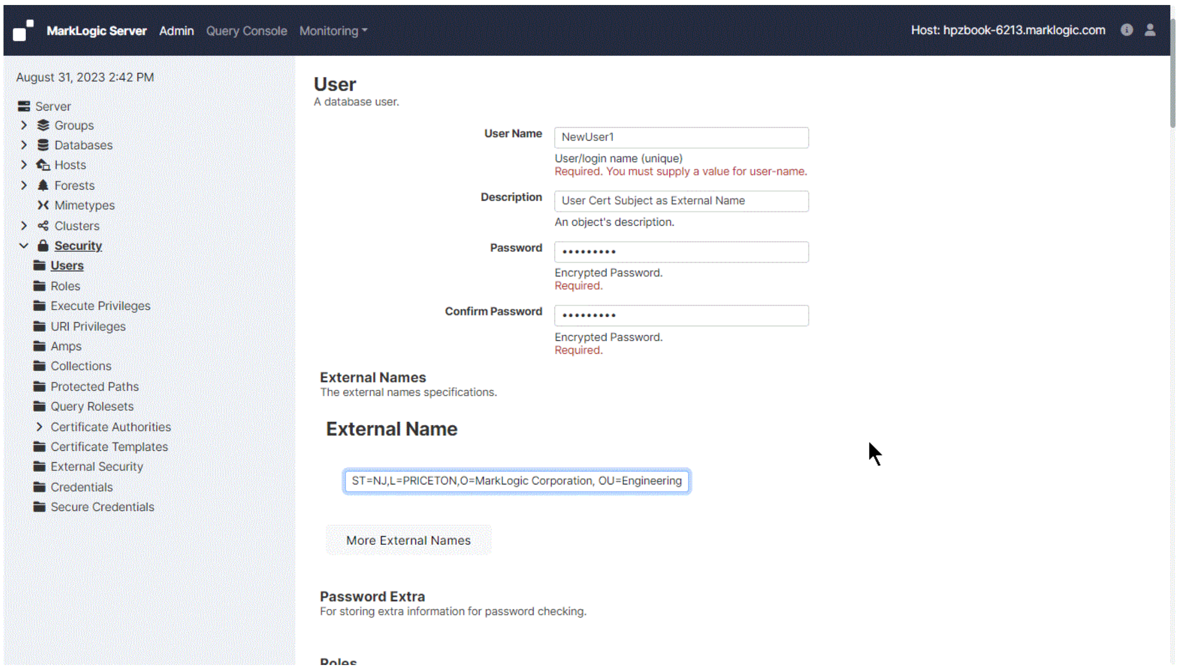
NOTE

You will also need to assign the necessary roles to the internal user to be able to access resources as needed.

User Certificate Subject Field Value as External Name for Internal User

Follow these steps to configure certificate-based user authentication for `demoUser1` as a MarkLogic Server external name for the internal user “newUser1”.

1. Create a user named “`newUser1`” with the necessary roles in the MarkLogic Server Security database (Internal User), and configure the User Certificate Subject field as external name to User.



August 31, 2023 2:42 PM

User
A database user.

User Name: NewUser1
User/login name (unique)
Required. You must supply a value for user-name.

Description: User Cert Subject as External Name
An object's description.

Password: •••••••••
Encrypted Password.
Required.

Confirm Password: •••••••••
Encrypted Password.
Required.

External Names
The external names specifications.

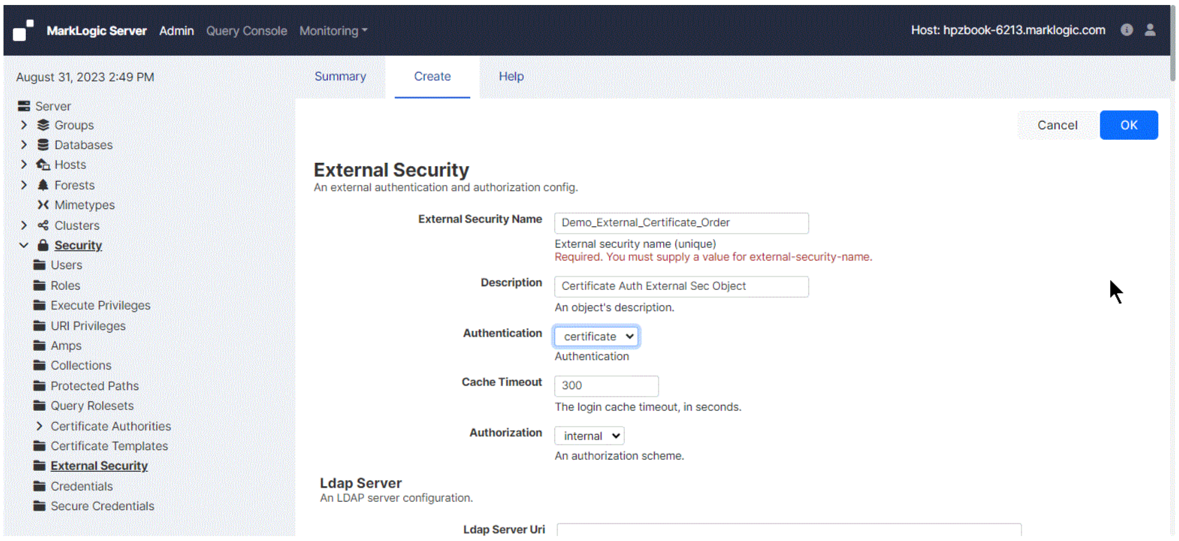
External Name

ST=NJ,L=PRICETON,O=MarkLogic Corporation, OU=Engineering

Password Extra
For storing extra information for password checking.

Roles

2. Create an external security object with certificate-based authentication.



August 31, 2023 2:49 PM

External Security
An external authentication and authorization config.

External Security Name: Demo_External_Certificate_Order
External security name (unique)
Required. You must supply a value for external-security-name.

Description: Certificate Auth External Sec Object
An object's description.

Authentication: certificate
Authentication

Cache Timeout: 300
The login cache timeout, in seconds.

Authorization: internal
An authorization scheme.

Ldap Server
An LDAP server configuration.

Ldap Server Uri

3. For external name (Cert Subject field) based linkage to Internal User, the app server needs to point to our external security object.

13.7. Example External Authorization Configurations

This section provides an example of how Kerberos and LDAP users and groups might be mapped to MarkLogic Server users and roles.

On Active Directory, there is a Kerberos user and an LDAP user assigned to an LDAP group:

- Kerberos Principal: `jsmith@MLTEST1.LOCAL`
- LDAP DN: `CN=John Smith,CN=Users,DC=MLTEST1,DC=LOCAL`
- LDAP memberOf: `CN=TestGroup Admin,CN=Users,DC=MLTEST1,DC=LOCAL`

On MarkLogic Server, the two users and the `ldaprole1` role are assigned external names that map them to the above users and LDAP group.

Kerberos User:

- User name: `krbuser1`
- External names: `jsmith@MLTEST1.LOCAL`

LDAP User:

- User name: `ldapuser1`
- External names: `CN=John Smith,CN=Users,DC=MLTEST1,DC=LOCAL`

Role:

- Role name: `ldaprole1`
- External names: `CN=TestGroup Admin,CN=Users,DC=MLTEST1,DC=LOCAL`

After authentication, the `xdmp:get-current-user()` function returns a different username, depending on the external authorization configuration. The possible combinations of configurations and returned names is shown in this table:

AuthenticationProtocol	AuthorizationScheme	Name Returned
kerberos	internal	`krbuser1`
kerberos	ldap	`jsmith@MLTEST1.LOCAL`(TEMP user with role `ldaprole1`)
ldap	internal	`ldapuser1`
ldap	ldap	`jsmith` (TEMP user with role `ldaprole1`)

13.8. Kerberos Authentication Using xdmp:http-* Functions

Kerberos authentication is supported by the `xdmp:http-get()`, `xdmp:http-post()`, `xdmp:http-put()`, and `xdmp:http-delete()` functions with the `negotiate` authentication option. When `negotiate` is specified, the `username` and `password` are not used. Instead, the server authenticates with the keytab file identified by an environment variable. This effectively does a `kinit` operation with the keytab file and then starts the MarkLogic Server.

To use this feature, you must set the following environment variables:

Environment Variable	Value
`MARKLOGIC_KEYTAB`	Path to the Kerberos client keytab file.
`MARKLOGIC_PRINCIPAL`	Kerberos Principal.

For example, to authenticate `xdmp:http-get()` for Kerberos, your function would look like the following.

XQuery:

```
xdmp:http-get("http://atsoi-z620.marklogic.com:8008/ticket.xqy",
<options xmlns="xdmp:http">
    <authentication method="negotiate">
    </authentication>
</options>)
```

JavaScript:

```
xdmp.httpGet("http://atsoi-z620.marklogic.com:8008/ticket.xqy",
{ "authentication": { "method" : "negotiate" } })
```

The `xdmp:http-get()`, `xdmp:http-post()`, `xdmp:http-put()`, and `xdmp:http-delete()` functions include a `kerberos-ticket-forwarding` option to enable the use of a user credential instead of `MARKLOGIC_PRINCIPAL`.

For example, to forward the ticket (if the user ticket is forwardable), do the following.

XQuery:

```
xdmp:http-get(“http://myhost.com:8005/index.xqy”,
  <options xmlns="xdmp:http">
    <authentication method="negotiate">
    </authentication>
    <kerberos-ticket-forwarding>{”optional”}
    </kerberos-ticket-forwarding>
  </options>)
```

JavaScript:

```
xdmp:httpGet(“http://myhost.com:8005/index.xqy”,
  {
    "authentication": {"method" : "negotiate"},
    "kerberosTicketForwarding": “optional”
  })
```

The `xdmp:http-get()xdmp:http-post()xdmp:http-put()` , and `xdmp:http-delete()` functions also have a `proxy` option to support proxy and proxy tunneling. When an HTTP or HTTPS request is sent to proxy server, the proxy server will forward the request to the destination.

For example, to forward requests to a proxy server, named `http://proxy.marklogic.com:8080`, do the following.

XQuery:

```
xdmp:http-get("http://targethost.marklogic.com/index.html",
  <options xmlns="xdmp:http">
    <proxy>http://proxy.marklogic.com:8080</proxy>
  </options>)
```

JavaScript:

```
xdmp.httpGet("http://targethost.marklogic.com/index.html",
   {proxy:"http://proxy.marklogic.com:8080"})
```

13.9. Kerberos Authentication for Secured HDFS

MarkLogic Server can use Kerberos Secured HDFS as a file system on Linux platforms. MarkLogic Server acts as a client to Kerberos Secured HDFS and should have its own unique identity, so the credentials provided to MarkLogic Server should be different from the Kerberos credentials of other MarkLogic client applications.

MarkLogic Server accesses Kerberos Secured HDFS using the keytab file and principal. To configure Kerberos authentication to Secured HDFS, set the following environment variables in your `/etc/marklogic.conf` file:

Environment Variable	Value
`MARKLOGIC_KEYTAB`	Path to the Kerberos client keytab file.
`MARKLOGIC_PRINCIPAL`	Kerberos Principal to be authenticated.

NOTE

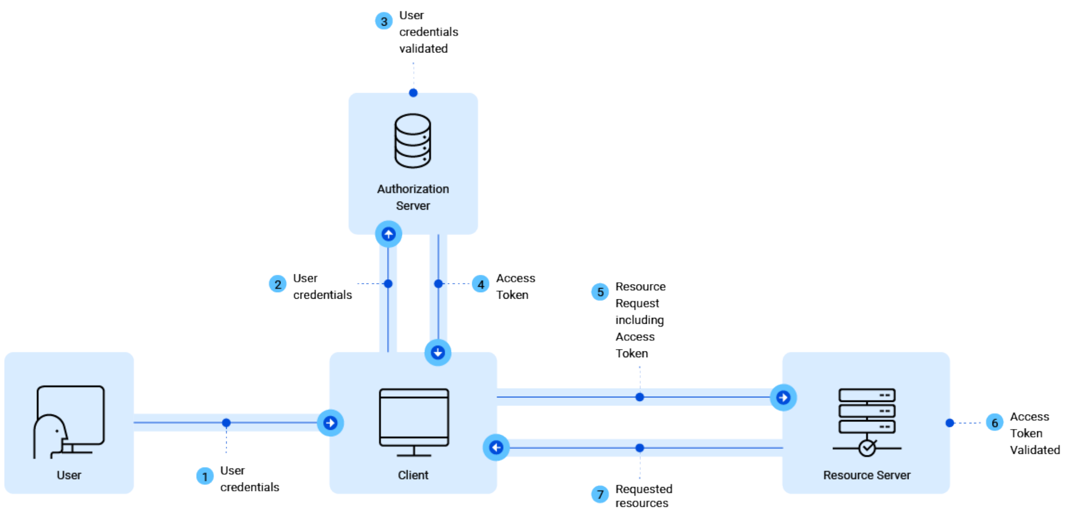
When using rolling upgrades, deploy your credential keytab files after the cluster has been fully upgraded to MarkLogic Server 9. Otherwise, the behavior of accessing secure HDFS will be undefined.

13.10. OAuth-Based Authentication and Authorization

OAuth 2.0 is an open standard for providing both authentication and authorization between two parties: the Authorization Server and the Resource Server.

The flow is shown in this diagram, where the Authorization Server is the external agent (the OAuth vendor) and the Resource Server is a MarkLogic Server app server:

OAuth-based authentication and authorization flow



1. The user sends their credentials to the client.
2. The client sends the user credentials to the Authorization Server: the OAuth vendor acting as the external agent.
3. The Authorization Server validates the user credentials.
4. The Authorization Server sends an Access Token to the client.
5. The client sends a resource request that includes the Access Token to the Resource Server: MarkLogic Server.
6. The Resource Server validates the Access Token.
7. The Resource Server sends the requested resources to the client.

Authentication occurs when the OAuth external agent validates the user, responding with an access token.

Authorization occurs when MarkLogic Server validates the access token included in the request header and assigns to the temporary user any roles with external names that relate to a field that you configured in the external security object.

There are two types of access tokens:

- **Internally managed reference tokens:** The resource server sends these tokens to the Introspection Endpoint of the authorization server for validation.

  NOTE
  MarkLogic Server deprecated support for this token type in v11.2.0.

- **JSON Web Tokens (JWT):** The resource server validates these tokens locally through the JWT signature. The JWT signature is validated through JWT secrets.

NOTE
MarkLogic Server began supporting this token type in v11.2.0.

NOTE
MarkLogic Server stores the JWT secrets in the internal keystore.

If you set up a foreign cluster with your Security database replicated to the foreign cluster, then you must sync your local and foreign internal keystores after creating OAuth **external security objects** on your local cluster. Sync them by using `xdmp:keystore-export()` to export the internal keystore from your local cluster then using `xdmp:keystore-import()` to import the internal keystore into your foreign cluster.

To request resources from MarkLogic Server, you must include the access token in the request header Authorization tag in one of these formats:

- **XML:** `<Authorization>Bearer` {access token goes here}`</Authorization>`
- **JSON:** `"Authorization": "Bearer` {access token goes here}`"`

MarkLogic Server supports configuring OAuth through the external agents listed in this section.

NOTE
You can also attempt to configure through unsupported external agents. All fields needed to configure external security are explained in these sections:

- Reference: The External Security Configuration Page
- Reference: The App Server Configuration Page
- Reference: The User Configuration Page
- Reference: The Role Configuration Page

13.10.1. With PingIdentity

You can set up MarkLogic Server to use the vendor PingIdentity as your OAuth external agent.

To set up the PingFederate server to properly interface with MarkLogic Server, noting, as you go along, the information that you will need later, follow these steps:

1. Register with PingIdentity to obtain a tenancy.
2. Configure a database to store user information.
   a. Set up users.
   b. Set up roles.
      - Note the roles names to configure as external names during role configuration.
3. Make sure that the proper IdP Adapter Mapping is properly configured with your system.
4. If you are using an asymmetric algorithm to sign JWT access tokens, then create or import the key certificate into the PingFederate Server.
5. Make sure that access tokens are properly configured to use JSON Web Tokens.
   a. Create an Access Token Manager that uses JSON Web Tokens as the Token Type.

- Note the Token Type for external security object configuration.

b. Ensure that a proper JWT Secret is configured for the token manager.
To use a symmetric algorithm like HS256 to sign, enter the symmetric key under the symmetric keys section.
To use an asymmetric algorithm like RS256 to sign, you need to have already inserted the key certificate into PingFederate (Step 3).
   - Note the value of JWS Algorithm for external security object configuration.
   - Note the value of Active Symmetry Key ID for external security object configuration.
   - Note the value of JWT Secret for external security object configuration.
   - (Optional) Note the value of JWKS Endpoint Path under Advanced fields JWKS for external security object configuration.

c. Create the JWT access token's payload structure to include claims containing username, roles, and (Optional) privilege information:

```
{
  "payload": {
  "username": <Username info>,
  "roles": <Roles info>,
  "privileges": <Privileges info>
}
```

   - Note the names of the elements from this payload for external security object configuration: username, roles, and (Optional) privileges.

6. Create an OAuth 2.0 client by setting the Access Token Manager for the OAuth client to the Access Token Manager created in Step 5.a.
   - Note the OAuth client name that you chose for external security object configuration.

Your PingFederate Server is now set up to integrate with MarkLogic Server, and you have the information that you need to configure MarkLogic Server external security.

This table shows how the elements that you noted from the PingFederate server map to fields on the MarkLogic Server External Security configuration page and to XQuery and REST API code schema elements. This table also includes the values used in the example setups:

PingIdentity element	**External Security configuration page field**	**Schema element**
OAuth Client Name **EXAMPLE:** `PingExampleClientID`	**OAuth Client ID**	`sec:oauth-client-id`
Token Type **EXAMPLE:** `JSON Web Tokens`	**OAuth Token Type**	`sec:oauth-token-type`
`username`	**OAuth Username Attribute**	`sec:oauth-username-attribute`
`roles`	**OAuth Role Attribute**	`sec:ouath-role-attribute`
(Optional) `privileges`	**OAuth Privilege Attribute**	`sec:oauth-privilege-attribute`
JWS Algorithm **EXAMPLE:** `HS256`	**OAuth JWT Algorithm**	`sec:oauth-jwt-alg`

PingIdentity element	External Security configuration page field	Schema element

JWT Secret
Active Summary Key ID
JWT Secret
EXAMPLE:
Active Summary Key ID:
`PingExampleKeyID`
JWT Secret:
`-----BEGIN PUBLIC KEY-----`
<PEM-converted JWT Secret>
`-----END PUBLIC KEY-----`
	OAuth JWT Secrets
	Secret Key ID
	Secret Value
		`sec:oauth-jwt-secrets`
		`sec:oauth-jwt-key-id`
		`sec:oauth-jwt-secret-value`

(Optional) JWKS Endpoint Path **EXAMPLE:** `https://localhost/pf/JWKS`	**OAuth JWKS URI**	`sec:oauth-jwks-uri`

You will also assign each role name returned in the JWT access token payload under the roles claim to a corresponding MarkLogic Server role as its external name.

EXAMPLE: `external-user-role`

Set up MarkLogic Server integration through one of the methods described in this section.

Through the Admin Interface

To set up OAuth-based authentication and authorization with Ping Identity through the Admin Interface, follow these steps:

1. Create your external security object by setting these fields on the External Security configuration page and clicking **OK**:

Field	Setting
External Security Name	Enter a descriptive name for this external security object that identifies the external agent. **EXAMPLE:** `PingIdentityExampleOAuth`
Description	(Optional) Enter a description for this external security object. **EXAMPLE:** `PingIdentity external security object for OAuth`
Authentication	Choose `oauth` from the dropdown. [v11.2.0 and up] Setting this field to `oauth` makes the **OAuth Server** fields available.
Cache Timeout	Enter a number in seconds after which you want MarkLogic Server to re-authenticate the user with your OAuth external agent. **EXAMPLE:** `300` (default kept) **NOTE** Clear the cache by calling sec:external-security-clear-cache().
Authorization	Choose `oauth` from the dropdown.

OAuth Server fields:

Field	Description
OAuth Flow Type	Choose `Resource server` from the dropdown.
OAuth Vendor	Choose `Ping Identity` from the dropdown.
OAuth Client ID	Enter the name of the OAuth client you created. **EXAMPLE:** `PingExampleClientID`
OAuth Token Type	Choose `JSON Web Tokens` from the dropdown.
OAuth Username Attribute	`username`
OAuth Role Attribute	`roles`
OAuth Privilege Attribute	(Optional) `privileges`
OAuth JWT Algorithm	Choose `HS256` from the dropdown.

OAuth JWT Secrets	1. Enter the key ID into the left field as the **Secret Key ID** and the public key in PEM format into the right field as the **Secret Value**.
	2. To enter more secrets, click **Add Secret** to expose additional field pairs.
	EXAMPLE:
	
	Secret Key ID = `PingExampleKeyID`
	
	Secret Value =
	
	`-----BEGIN PUBLIC KEY-----`
	
	<PEM-converted key>
	
	`-----END PUBLIC KEY-----`

OAuth JWKS URI	(Optional) JSON Web Key Sets Endpoint for obtaining JSON Web Keys. URI must support TLS (https) or be a loopback URI. **EXAMPLE:** `https://localhost/pf/JWKS`

2. Configure your desired app servers to use this external security object by setting these fields on each App Server configuration page and clicking **OK**:

Field	Setting
Authentication	Choose `oauth` from the dropdown.
Internal Security	Click the `false` radio button.
External Securities dropdown	Choose from the dropdown the **External Security Name** that you gave to your external security object in the previous step. Choose only one. **EXAMPLE:** `PingIdentityExampleOAuth`

3. Assign the external name to your desired roles by setting this field on each Role configuration page and clicking **OK**:

Field	Setting
External Name	Enter the external role name returned in the roles attribute of the JWT token payload that corresponds to this role. **EXAMPLE:** `external-user-role`

MarkLogic Server is now set up for OAuth-based authentication and authorization with Ping Identity.

Through XQuery or JavaScript

NOTE
Run all code against the MarkLogic Server Security database.

To set up OAuth-based authentication and authorization with Ping Identity using XQuery through the Query Console, follow these steps:

1. Create the external security object with code like this:
   XQuery

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

let $oauth-vendor := "Ping Identity",
$oauth-flow-type := "Resource server",
$oauth-client-id := "PingExampleClientID",
$oauth-token-type := "JSON Web Tokens",
$oauth-username-attribute := "username",
$oauth-role-attribute := "roles",
$oauth-privilege-attribute := "privileges",
$oauth-jwt-alg := "HS256",
$oauth-jwt-key-ids := "PingExampleKeyID",
$oauth-jwt-secret-values := "-----BEGIN PUBLIC KEY-----<PEM-converted HS256 JWT
Secret Value>-----END PUBLIC KEY-----",
$oauth-jwks-uri := "https://localhost/pf/JWKS"

let $oauth := sec:oauth-server(
$oauth-vendor,
$oauth-flow-type,
$oauth-client-id,
$oauth-token-type,
$oauth-username-attribute,
$oauth-role-attribute,
$oauth-privilege-attribute,
$oauth-jwt-alg,
$oauth-jwt-key-ids,
$oauth-jwt-secret-values,
$oauth-jwks-uri)

return sec:create-external-security(
'PingIdentityExampleOAuth',
'PingIdentity external security object for OAuth',
'oauth',
300,
'oauth',
(),
(),
$oauth)
```

   JavaScript

```
declareUpdate();
const sec = require('/MarkLogic/security');

const oauthVendor = "Ping Identity";
const oauthFlowType = "Resource server";
const oauthClientId = "PingExampleClientID";
const oauthTokenType = "JSON Web Tokens";
const oauthUsernameAttribute = "username";
const oauthRoleAttribute = "roles";
const oauthPrivilegeAttribute = "privileges";
const oauthJWTAlg = "HS256";
const oauthJWTKeyIds = "PingExampleKeyID";
const oauthJWTSecretValues = "-----BEGIN PUBLIC KEY-----<PEM-converted HS256 JWT
Secret Value>-----END PUBLIC KEY-----";
const oauthJWKSUri = "https://localhost/pf/JWKS";

const oauth = sec.oauthServer(
oauthVendor,
oauthFlowType,
oauthClientId,
oauthTokenType,
oauthUsernameAttribute,
oauthRoleAttribute,
oauthPrivilegeAttribute,
oauthJWTAlg,
oauthKeyIds,
oauthSecretValues,
oauthJWKSUri,
);

sec.createExternalSecurity(
'PingIdentityExampleOAuth',
'PingIdentity external security object for OAuth',
'oauth',
300,
'oauth',
null,
null,
oauth);
```

2. Create any HTTP, XDBC, WebDAV, or ODBC app servers that you wish to configure with this external security object.
3. Configure your app servers to use this external security object with code like this:

   XQuery

```
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 $appserver := <app server name>
let $extsec := "PingIdentityExampleOAuth"

return admin:save-configuration(admin:appserver-set-external-security($config,
admin:appserver-get-id($config, $groupid, $appserver), $extsec, fn:false(), "oauth"))
```

   JavaScript

```
declareUpdate();
const admin = require('/MarkLogic/admin.xqy');
const config = admin.getConfiguration();
const groupid = admin.groupGetId(config, "Default");
const appserver = <app server name>;
const extsec = "PingIdentityExampleOAuth";

admin.saveConfiguration(admin.appserverSetExternalSecurity(config, groupid,
admin.appServerGetId(config, appserver), extsec, fn.false(), "oauth"));
```

4. Assign external names to your desired roles with code like this:

> **NOTE**
> The external names are the values returned under the role attribute of the access token payload.

XQuery

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

let $role-name := <MarkLogic Server role name like "manage-user">
let $external-name := "external-user-role"
return sec:role-set-external-names($role-name, $external-name)
```

JavaScript

```
declareUpdate();
const sec = require('/MarkLogic/security.xqy');

const roleName = <MarkLogic Server role name like "manage-user">;
const externalName = "external-user-role";
sec.roleSetExternalNames(roleName, externalName);
```

MarkLogic Server is now set up for OAuth-based authentication and authorization with Ping Identity.

Through the REST API

To set up OAuth-based authentication and authorization with Ping Identity through the REST API, follow these steps:

1. Create the external security object with code like this:

XQuery/XML

```
curl -X POST --anyauth -k -u <username>:<password> -H "Content-Type:application/xml"
\
-d @create_extsec.xml http://<machine URI>:8002/manage/v2/external-security
```

Contents of `create_extsec.xml`

```
<external-security-properties xmlns="http://marklogic.com/manage/external-security/
properties">
    <external-security-name>PingIdentityExampleOAuth</external-security-name>
    <description>PingIdentity external security object for OAuth</description>
    <authentication>oauth</authentication>
    <cache-timeout>300</cache-timeout>
    <authorization>oauth</authorization>
    <oauth-server>
        <oauth-vendor>Ping Identity</oauth-vendor>
        <oauth-flow-type>Resource server</oauth-flow-type>
        <oauth-client-id>PingExampleClientID</oauth-client-id>
        <oauth-token-type>JSON Web Tokens</oauth-token-type>
        <oauth-username-attribute>username</oauth-username-attribute>
        <oauth-role-attribute>roles</oauth-role-attribute>
        <oauth-privilege-attribute>privileges</oauth-privilege-attribute>
        <oauth-jwt-alg>HS256</oauth-jwt-alg>
        <oauth-jwt-secrets>
            <oauth-jwt-secret>
                <oauth-jwt-key-id>PingExampleKeyID</oauth-jwt-key-id>
                <oauth-jwt-secret-value>-----BEGIN PUBLIC KEY-----PEM-converted HS256
JWT Secret Value-----END PUBLIC KEY-----</oauth-jwt-secret-value>
            </oauth-jwt-secret>
        </oauth-jwt-secrets>
        <oauth-jwks-uri>https://localhost/pf/JWKS/<oauth-jwks-uri>
    </oauth-server>
</external-security-properties>
```

JavaScript/JSON

```
curl -X POST --anyauth -k -u <username>:<password> -H "Content-Type:application/json"
\
-d @create_extsec.json http://<machine URI>:8002/manage/v2/external-security
```

Contents of `create_extsec.json`

```
{
  "external-security-name": "PingIdentityExampleOAuth",
  "description": "PingIdentity external security object for OAuth",
  "authentication": "oauth",
  "cache-timeout": "300",
  "authorization": "oauth",
  "oauth-server": {
    "oauth-vendor": "Ping Identity",
    "oauth-flow-type": "Resource server",
    "oauth-client-id": "PingExampleClientID",
    "oauth-token-type": "JSON Web Tokens",
    "oauth-username-attribute": "username",
    "oauth-role-attribute": "roles",
    "oauth-privilege-attribute": "privileges",
    "oauth-jwt-issuer-uri": "",
    "oauth-jwt-alg": "HS256",
    "oauth-jwt-secret": [
        {
            "oauth-jwt-key-id": "PingExampleKeyID",
            "oauth-jwt-secret-value": "-----BEGIN PUBLIC KEY-----<PEM-converted HS256
JWT Secret Value>-----END PUBLIC KEY----"
        }
    ],
    "oauth-jwks-uri": "https://localhost/pf/JWKS"
  }
}
```

2. Create any HTTP, XDBC, WebDAV, or ODBC app servers that you wish to configure with this external security object.
3. Configure your app servers to use this external security object with code like this:
   XML application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/xml" \
-d '<http-server-properties xmlns="http://marklogic.com/manage"> \
<external-security>"PingIdentityExampleOAuth"</external-security> \
<internal-security>false</internal-security> \
<authentication>oauth</authentication> \
</http-server-properties>' \
http://<machine URI>:8002/manage/v2/servers/<app server name>/properties?group-
id=Default
```

JSON application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/json" \
-d '{"external-security": "PingIdentityExampleOAuth", \
"internal-security": false, \
"authentication": "oauth"}' \
 http://<machine URI>:8002/manage/v2/servers/<app server name>/properties?group-
id=Default
```

4. Assign external names to your desired roles with code like this:

   NOTE
   The external names are the values returned under the role attribute of the access token payload.

   XML application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/xml" \
-d '<role-properties xmlns="http://marklogic.com/manage/role/properties"> \
<external-names> \
<external-name>"external-user-role"</external-name> \
</external-names> \
</role-properties>' http://<machine URI>:8002/manage/v2/roles/<MarkLogic Server role
name like manage-user>/properties
```

   JSON application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/json" \
-d '{"external-name": "external-user-role"}' \
http://<machine URI>:8002/manage/v2/roles/<MarkLogic Server role name like manage-
user>/properties
```

MarkLogic Server is now set up for OAuth-based authentication and authorization with Ping Identity.

13.10.2. With Microsoft Entra

You can set up MarkLogic Server to use the vendor Microsoft Entra (formerly Azure Active Directory) as your OAuth external agent.

To set up Microsoft Entra to properly interface with MarkLogic Server, noting, as you go along, the information that you will need later, follow these steps:

1. Register with Microsoft Entra to obtain a tenancy.
2. Create groups and users:
   a. Create users through the Users page.
   b. Create groups through the Groups page.
      - Note the group UUIDs. You will use these as external names during role configuration.
   c. Add the created users to the proper groups.
3. Register your application with Microsoft Entra:
   - Note the Application ID URI for external security object configuration.
   - Note the Tenant ID for external security object configuration.
4. Create a scope for your application.

5. Customize the payload of the JWT Token to include groups by changing your application's Manifest section **optionalClaims** field to this:

```
"optionalClaims": {
        "idToken": [
            {
                "name": "groups",
                "source": null,
                "essential": false,
                "additionalProperties": []
            }
        ],
        "accessToken": [
            {
                "name": "groups",
                "source": null,
                "essential": false,
                "additionalProperties": []
            }
        ],
        "saml2Token": [
            {
                "name": "groups",
                "source": null,
                "essential": false,
                "additionalProperties": []
            }
        ]
    }
```

6. Program your application to request this token.
7. Obtain public keys and their corresponding key IDs from Microsoft Entra:
   a. Go to `https://login.microsoftonline.com/`**`<your-tenant-UUID>`**`/discovery/v2.0/keys`. On the page that appears, each entry in the **keys** array is a public key containing **kid** as the key ID.
   b. Convert each entry in the **keys** array from JWK to PEM format using any public access tool.
      - Note the key ID for external security object configuration.
      - Note the PEM-converted public key for external security object configuration.

Microsoft Entra is now set up to integrate with MarkLogic Server, and you have the information that you need to configure MarkLogic Server external security.

This table shows how the elements that you noted from Microsoft Entra map to fields on the MarkLogic Server External Security configuration page in the Admin Interface and to XQuery and REST API code schema elements. This table also includes the values used in the example setups:

Microsoft Entra element	External Security configuration page field	Schema element
Application ID URI **EXAMPLE:** `https://testorganazation.onmicrosoft.com/63e66e0c-ed73-4db3-abb7-4faffa154445`	**OAuth Client ID**	`sec:oauth-client-id`
Tenant ID **EXAMPLE:** `https://sts.windows.net/3fc33f01-1894-4196-b81f-54417daac155/`	**OAuth JWT Issuer URI**	`sec:oauth-jwt-issuer-uri`
Name claim **EXAMPLE:** `name`	**OAuth Username Attribute**	`sec:oauth-username-attribute`

Microsoft Entra element	External Security configuration page field	Schema element
Groups claim **EXAMPLE:** `groups`	**OAuth Role Attribute**	`sec:ouath-role-attribute`

JWT Secrets
 	
kid
 	
keys array
 	
EXAMPLE (one kid/keys pair)**:**
 	
kid:
 	
`XRvko8P7A3UaWSnU7bM9nT0MjhA`
 	
keys:
 	
`-----BEGIN PUBLIC KEY-----`
 	
<PEM-converted key>
 	
`-----END PUBLIC KEY-----`
	OAuth JWT Secrets
	
	Secret Key ID
	
	Secret Value
		`sec:oauth-jwt-secrets`
		
		`sec:oauth-jwt-key-id`
		
		`sec:oauth-jwt-secret-values`

You will also assign Microsoft Entra group UUIDs to MarkLogic Server roles as external names. Microsoft Entra groups are analogous to MarkLogic Server roles.

EXAMPLE (of one)**:** `7228762e-cb30-428a-ae1a-3a8cf9e2f728`

Set up MarkLogic Server integration through one of the methods described in this section.

Through the Admin Interface

To set up OAuth-based authentication and authorization with Microsoft Entra through the Admin Interface, follow these steps:

1. Create your external security object by setting these fields on External Security configuration page and clicking **OK**:

Field	Setting
External Security Name	Enter a descriptive name for this external security object that identifies the external agent. **EXAMPLE:** `MicrosoftEntraExampleOAuth`
Description	(Optional) Enter a description for this external security object. **EXAMPLE:** `Microsoft Entra external security object for OAuth`
Authentication	Choose `oauth` from the dropdown. [v11.2.0 and up] Setting this field to `oauth` makes the **OAuth Server** fields available.
Cache Timeout	Enter a number in seconds after which you want MarkLogic Server to re-authenticate the user with your OAuth external agent instead of with the credentials stored in the cache. **EXAMPLE:** `300` (default kept) **NOTE** Clear the cache by calling sec:external-security-clear-cache().
Authorization	Choose `oauth` from the dropdown.

OAuth Server fields:

Field	Setting
OAuth Flow Type	Choose `Resource server` from the dropdown.

Field	Setting
OAuth Vendor	Choose `Microsoft Entra` from the dropdown.
OAuth Client ID	Enter the application ID found on your registered Microsoft Entra application's overview page. **EXAMPLE:** `37b06574-bdf0-42a2-9659-ebeaf8faf1c6`
OAuth JWT Issuer URI	Required: Enter your tenant ID found on your registered Microsoft Entra application's overview page. **EXAMPLE:** `https://sts.windows.net/3fc33f01-1894-4196-b81f-54417daac155/`
OAuth Token Type	Choose `JSON Web Tokens` from the dropdown.
OAuth Username Attribute	`name`
OAuth Role Attribute	`groups`
OAuth JWT Algorithm	Choose `RS256` from the dropdown.

OAuth JWT Secrets	1. Enter the key ID into the left field as the **Secret Key ID** and the public key in PEM format into the right field as the **Secret Value**.
	2. To enter more secrets, click **Add Secret** to expose additional field pairs.
	EXAMPLE:
	
	Secret Key ID = `XRvko8P7A3UaWSnU7bM9nT0MjhA`
	
	Secret Value =
	
	`-----BEGIN PUBLIC KEY-----`
	
	<PEM-converted key>
	
	`-----END PUBLIC KEY-----`

2. Configure your desired app servers to use this external security object by setting these fields on each App Server configuration page and clicking **OK**:

Field	Setting
Authentication	Choose `oauth` from the dropdown.
Internal Security	Click the `false` radio button.
External Securities dropdown	Choose from the dropdown the **External Security Name** that you gave to your external security object in the previous step. Choose only one. **EXAMPLE:** `MicrosoftEntraExampleOAuth`

3. Assign the external name to your desired roles by setting this field on each Role configuration page and clicking **OK**:

Field	Setting
External Name	From the Groups section of your Microsoft Entra tenancy page, enter the UUID of the role within the JWT payload field groups that you want to associate with this MarkLogic Server role. **EXAMPLE:** `7228762e-cb30-428a-ae1a-3a8cf9e2f728`

MarkLogic Server is now set up for OAuth-based authentication and authorization with Microsoft Entra.

Through XQuery or JavaScript

NOTE
Run all code against the MarkLogic Server Security database.

To set up OAuth-based authentication and authorization with Microsoft Entra using XQuery through the Query Console, follow these steps:

1. Create the external security object by executing code like this:
   XQuery

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

let $oauth-vendor := "Microsoft Entra",
$oauth-flow-type := "Resource server",
$oauth-client-id := "37b06574-bdf0-42a2-9659-ebeaf8faf1c6",
$oauth-token-type := "JSON Web Tokens",
$oauth-username-attribute := "name",
$oauth-role-attribute := "groups",
$oauth-jwt-issuer-uri := "https://sts.windows.net/3fc33f01-1894-4196-
b81f-54417daac155/",
$oauth-privilege-attribute := "", (:leave this empty for Entra:)
$oauth-jwt-alg := "RS256",
$oauth-jwt-key-ids := "XRvko8P7A3UaWSnU7bM9nT0MjhA",
$oauth-jwt-secret-values := "-----BEGIN PUBLIC KEY-----<Insert PEM-converted RS256
JWT Secret Value>-----END PUBLIC KEY-----",

$oauth-jwks-uri := "" (:leave this empty for Entra:)

let $oauth := sec:oauth-server(
$oauth-vendor,
$oauth-flow-type,
$oauth-client-id,
$oauth-token-type,
$oauth-username-attribute,
$oauth-role-attribute,
(),
$oauth-jwt-issuer-uri,
$oauth-jwt-alg,
$oauth-jwt-key-ids,
$oauth-jwt-secret-values)

return sec:create-external-security(
'MicrosoftEntraExampleOAuth',
'Microsoft Entra external security object for OAuth',
'oauth',
300,
'oauth',
(),
(),
$oauth)
```

   JavaScript

```
declareUpdate();
const sec = require('/MarkLogic/security');

const oauthVendor = "Microsoft Entra";
const oauthFlowType = "Resource server";
const oauthClientId = "37b06574-bdf0-42a2-9659-ebeaf8faf1c6";
const oauthTokenType = "JSON Web Tokens";
const oauthUsernameAttribute = "name";
const oauthRoleAttribute = "groups";
const oauthJWTIssuerUri = "https://sts.windows.net/3fc33f01-1894-4196-
b81f-54417daac155/";
const oauthJWTAlg = "RS256";
const oauthJWTKeyIds = "XRvko8P7A3UaWSnU7bM9nT0MjhA";
const oauthJWTSecretValues = "-----BEGIN PUBLIC KEY-----<PEM-converted RS256 Secret
Value>-----END PUBLIC KEY-----";

const oauth = sec.oauthServer(
oauthVendor,
oauthFlowType,
oauthClientId,
oauthTokenType,
oauthUsernameAttribute,
oauthRoleAttribute,
"",
oauthJWTIssuerUri,
oauthJWTAlg,
oauthJWTKeyIds,
oauthJWTSecretValues
);

sec.createExternalSecurity(
"MicrosoftEntraExampleOAuth",
"Microsoft Entra external security object for OAuth",
"oauth",
300,
"oauth",
null,
null,
oauth);
```

2. Create any HTTP, XDBC, WebDAV, or ODBC app servers that you wish to configure with this external security object.
3. Configure your app servers to use this external security object with code like this:

   XQuery

```
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 $appserver := <app server name>
let $extsec := "MicrosoftEntraExampleOAuth"

return admin:save-configuration(admin:appserver-set-external-security($config,
admin:appserver-get-id($config, $groupid, $appserver), $extsec, fn:false(), "oauth"))
```

   JavaScript

```
declareUpdate();
const admin = require('/MarkLogic/admin.xqy');
const config = admin.getConfiguration();
const groupid = admin.groupGetId(config, "Default");
const appserver = <app server name>;
const extsec = "MicrosoftEntraExampleOAuth";

admin.saveConfiguration(admin.appserverSetExternalSecurity(config, groupid,
admin.appServerGetId(config, appserver), extsec, fn.false(), "oauth"));
```

4. Assign external names to your desired roles with code like this:

   XQuery

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

let $role-name := <MarkLogic Server role name like "manage-user">
let $external-name := "7228762e-cb30-428a-ae1a-3a8cf9e2f728"
return sec:role-set-external-names($role-name, $external-name)
```

   JavaScript

```
declareUpdate();
const sec = require('/MarkLogic/security.xqy');

const roleName = <MarkLogic Server role name like "manage-user">;
const externalName = "7228762e-cb30-428a-ae1a-3a8cf9e2f72";
sec.roleSetExternalNames(roleName, externalName);
```

MarkLogic Server is now set up for OAuth-based authentication and authorization with Microsoft Entra.

Through the REST API

To set up OAuth-based authentication and authorization with Microsoft Entra through the REST API, follow these steps:

1. Create the external security object with code like this:

   XQuery/XML

```
curl -X POST --anyauth -k -u <username>:<password> -H "Content-Type:application/xml"
\
-d @create_extsec.xml http://<machine URI>:8002/manage/v2/external-security
```

   Contents of `create_extsec.xml`

```
<external-security-properties xmlns="http://marklogic.com/manage/external-security/
properties">
    <external-security-name>MicrosoftEntraExampleOAuth</external-security-name>
    <description>Microsoft Entra external security object for OAuth</description>
    <authentication>oauth</authentication>
    <cache-timeout>300</cache-timeout>
    <authorization>oauth</authorization>
    <oauth-server>
        <oauth-vendor>Microsoft Entra</oauth-vendor>
        <oauth-flow-type>Resource server</oauth-flow-type>
        <oauth-client-id>37b06574-bdf0-42a2-9659-ebeaf8faf1c6</oauth-client-id>
        <oauth-token-type>JSON Web Tokens</oauth-token-type>
        <oauth-username-attribute>name</oauth-username-attribute>
        <oauth-role-attribute>groups</oauth-role-attribute>
        <oauth-jwt-issuer-uri>https://sts.windows.net/3fc33f01-1894-4196-
b81f-54417daac155/</oauth-jwt-issuer-uri>
        <oauth-jwt-alg>RS256</oauth-jwt-alg>
        <oauth-jwt-secrets>
            <oauth-jwt-secret>
                <oauth-jwt-key-id>XRvko8P7A3UaWSnU7bM9nT0MjhA</oauth-jwt-key-
id>
                <oauth-jwt-secret-value>-----BEGIN PUBLIC KEY-----<PEM-converted
RS256 JWT Secret Value>-----END PUBLIC KEY-----</oauth-jwt-secret-value>
            </oauth-jwt-secret>
        </oauth-jwt-secrets>
        <oauth-jwks-uri></oauth-jwks-uri>
    </oauth-server>
</external-security-properties>
```

JavaScript/JSON

```
curl -X POST --anyauth -k -u <username>:<password> -H "Content-Type:application/json"
\
-d @create_extsec.json http://<machine URI>:8002/manage/v2/external-security
```

Contents of `create_extsec.json`

```
{
  "external-security-name": "MicrosoftEntraExampleOAuth",
  "description": "Microsoft Entra external security object for OAuth",
  "authentication": "oauth",
  "cache-timeout": "300",
  "authorization": "oauth",
  "oauth-server": {
    "oauth-vendor": "Microsoft Entra",
    "oauth-flow-type": "Resource server",
    "oauth-client-id": "37b06574-bdf0-42a2-9659-ebeaf8faf1c6",
    "oauth-token-type": "JSON Web Tokens",
    "oauth-username-attribute": "name",
    "oauth-role-attribute": "groups",
    "oauth-jwt-issuer-uri": "https://sts.windows.net/3fc33f01-1894-4196-
b81f-54417daac155/",
    "oauth-jwt-alg": "RS256",
    "oauth-jwt-secret": [
        {
            "oauth-jwt-key-id": "XRvko8P7A3UaWSnU7bM9nT0MjhA",
            "oauth-jwt-secret-value": "-----BEGIN PUBLIC KEY-----<PEM-converted RS256
JWT Secret Value>-----END PUBLIC KEY-----"
        }
    ],
    "oauth-jwks-uri": ""
  }
}
```

2. Create any HTTP, XDBC, WebDAV, or ODBC app servers that you wish to configure with this external security object.

3. Configure your app servers to use this external security object with code like this:
   XML application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/xml" \
-d '<http-server-properties xmlns="http://marklogic.com/manage"> \
<external-security>MicrosoftEntraExampleOAuth</external-security> \
<internal-security>false</internal-security> \
<authentication>oauth</authentication> \
</http-server-properties>' \
http://<machine URI>:8002/manage/v2/servers/<app server name>/properties?group-
id=Default
```

   JSON application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/json" \
-d '{"external-security": "MicrosoftEntraExampleOAuth", \
"internal-security": false, \
"authentication": "oauth"}' \
http://<machine URI>:8002/manage/v2/servers/<app server name>/properties?group-
id=Default
```

4. Assign external names to your desired roles with code like this:
   XML application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/xml" \
-d '<role-properties xmlns="http://marklogic.com/manage/role/properties"> \
<external-names><external-name>7228762e-cb30-428a-ae1a-3a8cf9e2f728</external-name> \
</external-names> \
</role-properties>' http://<machine URI>:8002/manage/v2/roles/<MarkLogic Server role
name like manage-user>/properties
```

   JSON application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/json" \
-d '{"external-name": "7228762e-cb30-428a-ae1a-3a8cf9e2f728"}' \
http://<machine URI>:8002/manage/v2/roles/<MarkLogic Server role name like manage-
user>/properties
```

MarkLogic Server is now set up for OAuth-based authentication and authorization with Microsoft Entra.

13.10.3. With Amazon Cognito

You can set up MarkLogic Server to use the vendor Amazon Cognito as your OAuth external agent.

To set up Amazon Cognito to properly interface with MarkLogic Server, noting, as you go along, the information that you will need later, follow these steps:

1. Register with Amazon Cognito to obtain your tenancy, called a user pool.
   - Note the user pool ID for external security object configuration.
2. Create your Amazon Cognito users and groups.
3. Register your application with Amazon Cognito.
   - Note the app client ID for external security object configuration.
4. Program your application to request this token.
5. Obtain public keys and their corresponding key IDs from Amazon Cognito.
   a. Go to `https://cognito-idp.<Region>.amazonaws.com/<userPoolId>/.well-known/jwks.json`. On the page that appears, each entry in the keys array is a public key containing kid as the key ID.
   b. Convert each entry in the keys array from JWK to PEM format using any public access tool.
   - Note the key ID for external security object configuration.
   - Note the PEM-converted public key for external security object configuration.

Amazon Cognito is now set up to integrate with MarkLogic Server, and you have the information that you need to configure MarkLogic Server external security.

This table shows how the elements that you noted from Amazon Cognito map to fields on the MarkLogic Server External Security configuration page in the Admin Interface and to XQuery and REST API code schema elements. This table also includes the values used in the example setups:

Amazon Cognito element	External Security configuration page field	Schema element
App client ID **EXAMPLE:** `19vomjilg46bbvcpp9qcmeacoc`	**OAuth Client ID**	`sec:oauth-client-id`
User pool ID **EXAMPLE:** `https://cognito-idp.us-east-1.amazonaws.com/us-east-1_fMQqTCMd9`	**OAuth JWT Issuer URI**	`sec:oauth-jwt-issuer-uri`
Name claim **EXAMPLE:** `username`	**OAuth Username Attribute**	`sec:oauth-username-attribute`
Groups claim **EXAMPLE:** `cognito:groups`	**OAuth Role Attribute**	`sec:ouath-role-attribute`

JWT Secrets
 	
kid
 	
keys array
 	
EXAMPLE (one kid/keys pair):
 	
kid:
 	
`fBwvWl/oWKPB9fyhXtZ8EqAhAmljMhk4hW2dd/zpFYs=`
 	
keys:
 	
`-----BEGIN PUBLIC KEY-----`
 	
<PEM-converted key>
 	
`-----END PUBLIC KEY-----`
	OAuth JWT Secrets
	
	Secret Key ID
	
	Secret Value
		`sec:oauth-jwt-secrets`
		
		`sec:oauth-jwt-key-id`
		
		`sec:oauth-jwt-secret-values`

You will also assign Amazon Cognito group names to MarkLogic Server roles as external names. Amazon Cognito groups are analogous to MarkLogic Server roles.

EXAMPLE (of one)**:** `GroupFoo`

Set up MarkLogic Server integration through one of the methods described in this section.

Through the Admin Interface

To set up OAuth-based authentication and authorization with Amazon Cognito through the Admin Interface, follow these steps:

1. Create your external security object by setting these fields on the External Security configuration page and clicking **OK**:

Field	Setting
External Security Name	Enter a descriptive name for this external security object that identifies the external agent. **EXAMPLE:** `AmazonCognitoExampleOAuth`
Description	(Optional) Enter a description for this external security object. **EXAMPLE:** `Amazon Cognito external security object for OAuth`
Authentication	Choose `oauth` from the dropdown. [v11.2.0 and up] Setting this field to `oauth` makes the **OAuth Server** fields available.

Field	Setting
Cache Timeout	Enter a number in seconds after which you want MarkLogic Server to re-authenticate the user with your OAuth external agent instead of with the credentials stored in the cache. **EXAMPLE:** `300` (default kept) **NOTE** Clear the cache by calling sec:external-security-clear-cache().
Authorization	Choose `oauth` from the dropdown.

OAuth Server fields:

Field	Setting
OAuth Flow Type	Choose `Resource server` from the dropdown.
OAuth Vendor	Choose `Amazon Cognito` from the dropdown.
OAuth Client ID	Enter the app client id on your User Pool Application Integration application's overview page. **EXAMPLE**: `19vomjilg46bbvcpp9qcmeacoc`
OAuth JWT Issuer URI	Required: Find the user pool ID from your Amazon Cognito user pool page and construct the JWT issuer URI like this: `https://cognito-idp.`<region>`.amazonaws.com/`<userpoolID> **EXAMPLE** (with user pool ID of `us-east-1_fMQqTCMd9`): `https://cognito-idp.us-east-1.amazonaws.com/us-east-1_fMQqTCMd9`
OAuth Token Type	Choose `JSON Web Tokens` from the dropdown.
OAuth Username Attribute	`username`
OAuth Role Attribute	`cognito:groups`
OAuth JWT Algorithm	Choose `RS256` from the dropdown.
OAuth JWT Secrets	1. Enter the key ID into the left field as the **Secret Key ID** and the public key in PEM format into the right field as the **Secret Value**. 2. To enter more secrets, click **Add Secret** to expose additional field pairs. **EXAMPLE:** **Secret Key ID** = `fBwvWl/oWKPB9fyhXtZ8EqAhAmljMhk4hW2dd/zpFYs=` **Secret Value** = `-----BEGIN PUBLIC KEY-----` <PEM-converted key> `-----END PUBLIC KEY-----`

2. Configure your desired app servers to use this external security object by setting these fields on each App Server configuration page and clicking **OK**:

Field	Setting
Authentication	Choose `oauth` from the dropdown.
Internal Security	Click the `false` radio button.
External Securities dropdown	Choose from the dropdown the **External Security Name** that you gave to your external security object in the previous step. Choose only one. **EXAMPLE:** `AmazonCognitoExampleOAuth`

3. Assign the external name to your desired roles by setting this field on each Role configuration page and clicking **OK**:

Field	Setting
External Name	Enter the group name from the user pool page Groups panel that corresponds to this MarkLogic Server role. **EXAMPLE:** `GroupFoo`

MarkLogic Server is now set up for OAuth-based authentication and authorization with Amazon Cognito.

Through XQuery or JavaScript

NOTE
Run all code against the MarkLogic Server Security database.

To set up OAuth-based authentication and authorization with Amazon Cognito using XQuery through the Query Console, follow these steps:

1. Create the external security object by executing code like this:
   XQuery

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

let $oauth-vendor := "Amazon Cognito",
$oauth-flow-type := "Resource server",
$oauth-client-id := "19vomjilg46bbvcpp9qcmeacoc",
$oauth-token-type := "JSON Web Tokens",
$oauth-username-attribute := "username",
$oauth-role-attribute := "cognito:groups",
$oauth-jwt-issuer-uri := "https://cognito-idp.us-east-1.amazonaws.com/us-
east-1_fMQqTCMd9",
$oauth-privilege-attribute := "", (:leave this empty for Cognito:)
$oauth-jwt-alg := "RS256",
$oauth-jwt-key-ids := "fBwvWl/oWKPB9fyhXtZ8EqAhAmljMhk4hW2dd/zpFYs=",
$oauth-jwt-secret-values := "-----BEGIN PUBLIC KEY-----<PEM-converted RS256 JWT
Secret Value>-----END PUBLIC KEY-----",

$oauth-jwks-uri := "" (:leave this empty for Cognito:)

let $oauth := sec:oauth-server(
$oauth-vendor,
$oauth-flow-type,
$oauth-client-id,
$oauth-token-type,
$oauth-username-attribute,
$oauth-role-attribute,
(),
$oauth-jwt-issuer-uri,
$oauth-jwt-alg,
$oauth-jwt-key-ids,
$oauth-jwt-secret-values)

return sec:create-external-security(
'AmazonCognitoExampleOAuth',
'Amazon Cognito external security object for OAuth',
'oauth',
300,
'oauth',
(),
(),
$oauth)
```

JavaScript

```
declareUpdate();
const sec = require('/MarkLogic/security');

const oauthVendor = "Amazon Cognito";
const oauthFlowType = "Resource server";
const oauthClientId = "19vomjilg46bbvcpp9qcmeacoc";
const oauthTokenType = "JSON Web Tokens";
const oauthUsernameAttribute = "username";
const oauthRoleAttribute = "cognito:groups";
const oauthJWTAlg = "RS256";
const oauthJWTIssuerUri = "https://cognito-idp.us-east-1.amazonaws.com/us-
east-1_fMQqTCMd9";
const oauthJWTKeyIds = "fBwvWl/oWKPB9fyhXtZ8EqAhAmljMhk4hW2dd/zpFYs=";
const oauthJWTSecretValues = "-----BEGIN PUBLIC KEY-----<PEM-converted RS256 Secret
Value>-----END PUBLIC KEY-----";

const oauth = sec.oauthServer(
oauthVendor,
oauthFlowType,
oauthClientId,
oauthTokenType,
oauthUsernameAttribute,
oauthRoleAttribute,
"",
oauthJWTIssuerUri,
oauthJWTAlg,
oauthJWTKeyIds,
oauthJWTSecretValues
);

sec.createExternalSecurity(
"AmazonCognitoExampleOAuth",
"Amazon Cognito external security object for OAuth",
"oauth",
300,
"oauth",
null,
null,
oauth);
```

2. Create any HTTP, XDBC, WebDAV, or ODBC app servers that you wish to configure with this external security object.
3. Configure your app servers to use this external security object with code like this:

   XQuery

```
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 $appserver := <app server name>
let $extsec := "AmazonCognitoExampleOAuth"

return admin:save-configuration(admin:appserver-set-external-security($config,
admin:appserver-get-id($config, $groupid, $appserver), $extsec, fn:false(), "oauth"))
```

   JavaScript

```
declareUpdate();
const admin = require('/MarkLogic/admin.xqy');
const config = admin.getConfiguration();
const groupid = admin.groupGetId(config, "Default");
const appserver = <app server name>;
const extsec = "AmazonCognitoExampleOAuth";

admin.saveConfiguration(admin.appserverSetExternalSecurity(config, groupid,
admin.appServerGetId(config, appserver), extsec, fn.false(), "oauth"));
```

4. Assign external names to your desired roles with code like this:

   XQuery

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

   let $role-name := <MarkLogic Server role name like "manage-user">
   let $external-name := "GroupFoo"
   return sec:role-set-external-names($role-name, $external-name)
   ```

   JavaScript

   ```
   declareUpdate();
   const sec = require('/MarkLogic/security.xqy');

   const roleName = <MarkLogic Server role name like "manage-user">;
   const externalName = "GroupFoo";
   sec.roleSetExternalNames(roleName, externalName);
   ```

MarkLogic Server is now set up for OAuth-based authentication and authorization with Amazon Cognito.

Through the REST API

To set up OAuth-based authentication and authorization with Amazon Cognito through the REST API, follow these steps:

1. Create the external security object with code like this:

   XQuery/XML

   ```
   curl -X POST --anyauth -k -u <username>:<password> -H "Content-Type:application/xml"
   \
   -d @create_extsec.xml http://<machine URI>:8002/manage/v2/external-security
   ```

   Contents of `create_extsec.xml`

```
<external-security-properties xmlns="http://marklogic.com/manage/external-security/
properties">
    <external-security-name>AmazonCognitoExampleOAuth</external-security-name>
    <description>Amazon Cognito external security object for OAuth</description>
    <authentication>oauth</authentication>
    <cache-timeout>300</cache-timeout>
    <authorization>oauth</authorization>
    <oauth-server>
        <oauth-vendor>Amazon Cognito</oauth-vendor>
        <oauth-flow-type>Resource server</oauth-flow-type>
        <oauth-client-id>19vomjilg46bbvcpp9qcmeacoc</oauth-client-id>
        <oauth-token-type>JSON Web Tokens</oauth-token-type>
        <oauth-username-attribute>username</oauth-username-attribute>
        <oauth-role-attribute>cognito:groups</oauth-role-attribute>
        <oauth-jwt-issuer-uri>https://cognito-idp.us-east-1.amazonaws.com/us-
east-1_fMQqTCMd9</oauth-jwt-issuer-uri>
        <oauth-jwt-alg>RS256</oauth-jwt-alg>
        <oauth-jwt-secrets>
            <oauth-jwt-secret>
                <oauth-jwt-key-id>fBwvWl/oWKPB9fyhXtZ8EqAhAmljMhk4hW2dd/zpFYs=</oauth-
jwt-key-id>
                <oauth-jwt-secret-value>-----BEGIN PUBLIC KEY-----<PEM-converted
RS256 JWT Secret Value>-----END PUBLIC KEY-----</oauth-jwt-secret-value>
            </oauth-jwt-secret>
        </oauth-jwt-secrets>
        <oauth-jwks-uri></oauth-jwks-uri>
    </oauth-server>
</external-security-properties>
```

JavaScript/JSON

```
curl -X POST --anyauth -k -u <username>:<password> -H "Content-Type:application/json"
\
-d @create_extsec.json http://<machine URI>:8002/manage/v2/external-security
```

Contents of `create_extsec.json`

```
{
  "external-security-name": "AmazonCognitoExampleOAuth",
  "description": "Amazon Cognito external security object for OAuth",
  "authentication": "oauth",
  "cache-timeout": "300",
  "authorization": "oauth",
  "oauth-server": {
    "oauth-vendor": "Microsoft Entra",
    "oauth-flow-type": "Resource server",
    "oauth-client-id": "19vomjilg46bbvcpp9qcmeacoc",
    "oauth-token-type": "JSON Web Tokens",
    "oauth-username-attribute": "username",
    "oauth-role-attribute": "cognito:groups",
    "oauth-jwt-issuer-uri": "https://cognito-idp.us-east-1.amazonaws.com/us-
east-1_fMQqTCMd9",
    "oauth-jwt-alg": "RS256",
    "oauth-jwt-secret": [
        {
            "oauth-jwt-key-id": "fBwvWl/oWKPB9fyhXtZ8EqAhAmljMhk4hW2dd/zpFYs=ID",
            "oauth-jwt-secret-value": "-----BEGIN PUBLIC KEY-----<PEM-converted RS256
JWT Secret Value>-----END PUBLIC KEY-----"
        }
    ],
    "oauth-jwks-uri": ""
  }
}
```

2. Create any HTTP, XDBC, WebDAV, or ODBC app servers that you wish to configure with this external security object.

3. Configure your app servers to use this external security object with code like this:
   XML application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/xml" \
-d '<http-server-properties xmlns="http://marklogic.com/manage"> \
<external-security>AmazonCognitoExampleOAuth</external-security> \
<internal-security>false</internal-security> \
<authentication>oauth</authentication> \
</http-server-properties>' \
http://<machine URI>:8002/manage/v2/servers/<app server name>/properties?group-
id=Default
```

   JSON application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/json" \
-d '{"external-security": "AmazonCognitoExampleOAuth", \
"internal-security": false, \
"authentication": "oauth"}' \
http://<machine URI>:8002/manage/v2/servers/<app server name>/properties?group-
id=Default
```

4. Assign external names to your desired roles with code like this:
   XML application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/xml" \
-d '<role-properties xmlns="http://marklogic.com/manage/role/properties"> \
<external-names> \
<external-name>GroupFoo</external-name> \
</external-names> \
</role-properties>' http://<machine URI>:8002/manage/v2/roles/<MarkLogic Server role
name like manage-user>/properties
```

   JSON application

```
curl -X PUT --anyauth -k -u <username>:<password> -H "Content-type:application/json" \
-d '{"external-name": "GroupFoo"}' \
http://<machine URI>:8002/manage/v2/roles/<MarkLogic Server role name like manage-
user>/properties
```

MarkLogic Server is now set up for OAuth-based authentication and authorization with Amazon Cognito.

13.11. Reference: The External Security Configuration Page

This section describes how to access the **External Security** configuration page from the Admin Interface and provides a field index.

13.11.1. Accessing the External Security Page

To access the Admin Interface **External Security** configuration page, follow these steps:

1. Click **Security** in the left tree menu.
2. Click **External Security** in the left tree menu.
3. Either click name of an existing external security object or create a new one by clicking the **Create** tab. The **External Security** configuration page appears.
4. Complete the fields as necessary.
5. Click **OK**. Your external security object is modified or created.

13.11.2. External Security Configuration Page Field Index

The tables in this section describe each field available on the **External Security** configuration page.

Field	Description
External Security Name	The name used to identify this external security object. This is the name that you provide when you configure your app server for external security. Include the external agent in the name.
Description	(Optional) The description of this external security object.
Authentication	The authentication protocol to use. This field identifies how the credentials are meant to be validated.
Cache Timeout	The login cache timeout, in seconds. When the timeout period is exceeded, MarkLogic Server re-authenticates the user with the external agent. **NOTE** Clear the cache by calling sec:external-security-clear-cache().
Authorization	The authorization scheme.

The LDAP Server Fields

[v11.2.0 and up] The **LDAP Server** fields appear when either **Authentication** or **Authorization** is `ldap`.

Field	Description
LDAP Server URI	The URI for the LDAP server. Required if either **Authentication** or **Authorization** is `ldap`.
LDAP Base	The base DN for user lookup. Required if either **Authentication** or **Authorization** is `ldap`.
LDAP Attribute	The name of the attribute (for example, `sAMAccountName`) used to identify the user on the LDAP server. Required if either **Authentication** or **Authorization** is `ldap`.
LDAP Default User	The LDAP default user. Required if either **Authorization** = `ldap` or **LDAP Bind Method** is `simple`.
LDAP Password	The password for the **LDAP Default User**. Required if either **Authorization** is `ldap` or **LDAP Bind Method** is `simple`.
Confirm LDAP Password	Field to confirm the **LDAP Password**.
LDAP Bind Method	• `MD5` (default): • Uses the DIGEST-MD5 authentication method with the LDAP server. • Set **LDAP Default User** to the name of a valid LDAP user. • `simple` • Uses simple bind authentication with the LDAP server. • Set **LDAP Default User** to a DN (Distinguished Name). • Set **LDAP Password**. • Use LDAPS (LDAP with SSL) because the password is not encrypted. (That is, make sure **LDAP Server URI** begins with `ldaps://` instead of `ldap://`.) • `external` • Uses a certificate to authenticate with the LDAP server. • Set **LDAP Start TLS** to true. • Enter the certificate into **LDAP Certificate**.
LDAP Memberof Attribute	(Optional) The LDAP attribute for group lookup. If not specified, `memberOf` is used for search for the groups of a user.
LDAP Member Attribute	(Optional) The LDAP attribute for group lookup. If not specified, `member` is used for search for the group of a group.
LDAP Start TLS	Whether or not to use start TLS request to the LDAP server. Set to `true` to use start TLS request. If set to `true`, the **LDAP server URI** should start with `ldap://` instead of `ldaps://`.
LDAP Certificate	The PEM-encoded X.509 certificate for MarkLogic Server to connect the LDAP server using mutual authentication. Required if **LDAP Bind Method** is `external`. Optional if **LDAP Bind Method** is either `MD5` or `simple`.
LDAP Private Key	The PEM-encoded private key corresponding to the **LDAP Certificate**. Required if **LDAP Bind Method** is `external`. Optional if **LDAP Bind Method** is either `MD5` or `simple`.
LDAP Nested Lookup	Whether or not to perform nested group lookup.
LDAP Remove Domain	Whether or not to remove `domain` before matching with `ldap-attribute`.

Field	Description
LDAP Negative Cache Timeout	The LDAP negative cache timeout in seconds. MarkLogic Server caches negative lookups to avoid overloading the external LDAP server. **NOTE** Clear the cache by calling sec:external-security-clear-cache().

The SAML Server Fields

[v11.2.0 and up] The **SAML Server** fields appear when either **Authentication** or **Authorization** is `saml`.

Field	Description
SAML Entity ID	Entity ID of the Identity Provider (IDP). Typically in URL form.
SAML Destination	The URL that identifies the Identity Provider to accept the authentication request.
SAML Issuer	Entity ID of the Service Provider (SP), your MarkLogic Server instance. Typically in URL form.
SAML Assertion Host	The URL that identifies the host making the assertion
SAML IDP Certificate Authority	The certificate used to validate the signature in the authentication request.
SAML SP Certificate	(Required when you use `https` for **SAML Destination**) The certificate used to sign the authentication request.
SAML SP Private Key	(Required when you use `https` for **SAML Destination**) The private key used to sign the authentication request.
SAML Attribute Names	One or more SAML attribute names. These names will be requested as part of the attribute query and mapped as appropriate to internal MarkLogic Server roles.
SAML Authn Signature	The signature algorithm used to generate the SAML authentication signature.
SAML Privilege Attribute Name	(Optional when **Authorization** is `saml`) SAML privilege attribute name. If specified, the name will also be requested as part of the attribute query and mapped to MarkLogic Server privileges.

The OAuth Server Fields

[v11.2.0] The **OAuth Server** fields appear when either **Authentication** or **Authorization** is `oauth`.

Field	Description
OAuth Flow Type	The type of flow the OAuth server will support: • `Resource server` • `Authorization code` [Deprecated as of MarkLogic Server 11.2.0] • `Client credentials` [Deprecated as of MarkLogic Server 11.2.0]
OAuth Vendor	The third-party authorization vendor that will be used with the OAuth server.
OAuth Server URI	[Deprecated as of MarkLogic Server 11.2.0] (Optional) Providing a server URL may help users with auto-population of form parameters. URI must support TLS (HTTPS).
OAuth Authorization Server URI	[Deprecated as of MarkLogic Server 11.2.0] (**OAuth Token Type** = `Internally managed reference tokens`) OAuth introspection endpoint.
OAuth Token Server URI	[Deprecated as of MarkLogic Server 11.2.0] (Optional) Token Endpoint used to obtain access tokens. URI must support TLS (HTTPS).
OAuth Introspection Server URI	[Deprecated as of MarkLogic Server 11.2.0] (**OAuth Token Type** = `Internally managed reference tokens`) OAuth introspection endpoint. TLS (HTTPS) required.

Field	Description
OAuth Scope	[Deprecated as of MarkLogic Server 11.2.0] (**OAuth Flow Type** = `Client credentials`) Scopes to be requested in client flows.
OAuth Client Authentication Method	[Deprecated as of MarkLogic Server 11.2.0] (**OAuth Flow Type** = `Client credentials`) Method for authenticating the client when requesting access tokens.
OAuth Client ID	Client ID of the OAuth server on the vendor.
OAuth Client Secret	[Deprecated as of MarkLogic Server 11.2.0] (**OAuth Flow Type** = `Client credentials` and **OAuth Client Authentication Method** = `Client secret`) The client secret you use to authenticate with the OAuth vendor.
OAuth Redirect URI	[Deprecated as of MarkLogic Server 11.2.0] (**OAuth Flow Type** = `Authorization code`) URI where user is redirected after authentication. URI must support TLS (HTTPS) or be a loopback URI.
OAuth JWT Issuer URI	(**OAuth Vendor** = `Microsoft Entra` or `Amazon Cognito`) MarkLogic Server verifies that this URI matches the URI provided in the JWT Token.
OAuth Token Type	The access token format.
OAuth Username Attribute	(Required) The claim name to store the username information extracted from the access token.
OAuth Role Attribute	(Required) The claim name to store the role information extracted from the access token.
OAuth Privilege Attribute	(Optional) The claim name to store the privilege information extracted from the access token.
OAuth JWT Algorithm	(**OAuth Token Type** = `JSON Web Tokens`) Signature algorithm for JWT access tokens.
OAuth JWT Secrets	The list of secrets MarkLogic Server should use to verify JWT access tokens, also known as validating the JWT access token signature.
OAuth JWKS URI	(Optional) JSON Web Key Sets Endpoint for obtaining JSON Web Keys. URI must support TLS (HTTPS) or be a loopback URI.

13.12. Reference: The App Server Configuration Page

This section describes how to access the relevant app server configuration page from the Admin Interface and provides a field index for any fields you need to configure for external security.

13.12.1. Accessing the App Server Configuration Page

To access the relevant Admin Interface **App Server** configuration page, follow these steps:

1. Click **Groups** in the left tree menu.
2. Click the group in which you want to configure or create the app server (for example, **Default**).
3. Click **App Servers**. The **App Server Summary** page appears.
4. Either click the name of an existing app server or create a new one by clicking the Create HTTP, Create WebDAV, Create XDBC, or Create ODBC tab. The relevant **App Server** configuration page appears.
5. Set the fields as necessary.
6. Click **OK**. The app server is configured for external authentication.

13.12.2. App Server Page Field Index

This table describes the fields relevant to configuring app servers for external security. Other fields are described in these sections:

- Create a new HTTP server
- Create a new WebDAV server
- Create a new XDBC server
- Create a new ODBC server

Field	Description
Authentication	Specifies what type of credentials MarkLogic Server should expect from the end user.

Field	Description
Internal Security	Specifies whether MarkLogic Server should use the Security database for authentication and authorization.
External Securities dropdown	Specifies the name of the external security object that MarkLogic Server should use to externally authenticate this app server. **NOTE** [LDAP] You can configure your app server for multiple LDAP external security objects: 1. The LDAP external security objects are used in the order that you list them here. 2. Once the time specified in `Cache Timeout` passes for the LDAP server specified in the first external security object, then the one in the next external security object on the list is tried.
Default User	(**Authentication** = `application-level`) Specifies the default internal user that MarkLogic Server should automatically log anyone accessing this server in as until that user explicitly logs in.

13.13. Reference: The User Configuration Page

To assign an external name to a user through the Admin Interface **User** configuration page, follow these steps:

1. Click **Security** in the left tree menu.
2. Click **Users**. The **User** summary page appears.
3. Either click the name of an existing user or create a new one by clicking the **Create** tab at the top of the **User** summary page. The **User** configuration page appears.
4. In the **External Name** section, enter the external name (for example, an LDAP Distinguished Name (DN) or a Kerberos User Principal) for the user into the **[add]** field.
5. [Optional] Click **More External Names** to access more **[add]** fields to fill.
6. Complete any other fields as necessary.
7. Click **OK**. The external names provided are assigned to this user.

NOTE
Users further describes how to configure a user.

13.14. Reference: The Role Configuration Page

To assign an external name to a role through the Admin Interface **Role** configuration page, follow these steps:

1. Click **Security** in the left tree menu.
2. Click **Roles**. The **Role** summary page appears.
3. Either click the name of an existing role or create a new one by clicking the **Create** tab at the top of the **Role** summary page. The **Role** configuration page appears.
4. In the **External Name** section, enter the external name (for example, SAML Identity Provider ID + attribute name + value) for the role into the **[add]** field.
5. [Optional] Click **More External Name**s to access more **[add]** fields to fill.
6. Complete any other fields as necessary.

7. Click **OK**. The external names provided are assigned to this role.

NOTE
Roles further describes how to configure a user.

14. Encryption at Rest

Encryption at rest protects your data on media - which is “data at rest” as opposed to data moving across a communications channel, otherwise known as “data in motion.” Increasing security risks and compliance requirements sometimes mandate the use of encryption at rest to prevent unauthorized access to data on disk.

NOTE
To use encryption at rest with an external key management system (KMS), an Advanced Security License key that includes this feature is required. For details on purchasing a license key for the Advanced Security features, contact your MarkLogic Server sales representative. See Licensing for more information.

Encryption at rest can be configured to encrypt data, log files, and configuration files separately. Encryption is only applied to newly created files once encryption at rest is enabled and does not apply to existing files without further action by the user. For existing data, a merge or re-index will trigger encryption of data, a configuration change will trigger encryption of configuration files, and log rotation will initiate log encryption.

14.1. Licensing

The use of an external Key Management System (KMS) or keystore with encryption at rest requires an Advanced Security License, in addition to the regular license. See Licensing for more details.

14.2. Terms and Definitions

The following terms and definitions are associated with encryption at rest.

Term	Definition
Encryption at rest	Encryption of data that is stored on digital media.
KMS	Key Management System.
wallet	The PKCS #11 secured wallet provided and managed by MarkLogic Server that functions as the default standalone KMS.
KEK	A Key Encryption Key used to encrypt or ‘wrap’ another encryption key.
keystore	Repository for crytographic keys in the PKCS #11 secured wallet or any external KMS that is KMIP-server conformant.
KMIP	Key Management Interoperability Protocol (KMIP specification) - governed by OASIS standards body. There are multiple versions of KMIP currently available. MarkLogic Server Encryption supports 1.2.
PKCS #11	One of the Public-Key Cryptography Standards, and also the programming interface to create and manipulate cryptographic tokens. See the OASIS PKCS TC for details.
MKEK	Master Key Encryption Key, resides in the keystore, and is used to generate the CKEK, which is enveloped (encrypted) with the MKEK.
CKEK	Cluster Key Encryption Key, resides in the keystore and is used to encrypt the data (CDKEK), configuration(CCKEK), and log (CLKEK) encryption keys.
CDKEK	Cluster Data Key Encryption Key, used to directly encrypt (wrap) the object key encryption keys (OKEY) for stands, forest journals, and large files.
CCKEK	Cluster Configuration Key Encryption Key, used to encrypt (wrap) the object key encryption keys (OKEY) for configuration files.
CLKEK	Cluster Log Key Encryption Key, used to encrypt (wrap) the object key encryption keys (OKEY) for log files.

Term	Definition
OKEY	Object Encryption Key, otherwise known as the data object encryption key, a symmetric key used to directly encrypt objects like stands, forest journals, large files, configuration files, or log files.
BKEK	Backup Key Encryption Key, used to encrypt backups, both full and incremental. The BKEK is a locally generated backup KEK, that is used to encrypt all files in the backup. The BKEK is encrypted with the CDKEY and the BDKEY.
BDKEK	Backup Database Key, (alternative) only applicable to external KMS configurations. It is used to encrypt a backup in addition to the CDKEK.
HSM	Hardware Security Module or other hardware device is a physical computing device that safeguards and manages digital key materials.
Key strength	The size of key in bits. Usually, the more bits, the stronger the key and more difficult to break; for example, 128-bits, 256 bits, or 512-bits, and so on.
Key rotation	The process of aging out and replacing encryption keys over time.

14.3. Understanding Encryption at Rest

Encryption at rest enables you to transparently and selectively encrypt your data residing on disk (locally or in the cloud) in MarkLogic Server clusters. You can set your options at the cluster level to encrypt data on all the hosts in that cluster.

Three types of data can be encrypted:

- User data - data ingested into MarkLogic Server databases, along with derived data such as indexes, user dictionaries, journals, backups, and so on
- Configuration files - all configuration files generated by MarkLogic Server (for example, whenever a change is made to the configuration file)
- Log files - all log files generated by MarkLogic Server, such as error logs, access logs, service dumps, server error logs, logs for each application server, and the task server logs

There are both MarkLogic Application Server logs and MarkLogic Server logs; *both types of logs* will be encrypted as part of log encryption.

NOTE

If you are using the Default Conversion Option described in The Default Conversion Option in the *Content Processing Framework Guide*. Note that the MarkLogic Converters package may generate temporary files, which are not supported by encryption at rest.

These types of data can each be encrypted separately. You can configure encryption for databases individually, or at the cluster level. Encryption at rest is “off” by default. To use encryption at rest, you need to configure and enable encryption for your database(s), configuration files, and/or log files.

NOTE

To access unencrypted forest data, MarkLogic Server normally uses memory-mapped files. When files are encrypted, MarkLogic Server instead decrypts them to anonymous memory. As a result, encrypted MarkLogic Server forests use more anonymous memory and less file-mapped memory than unencrypted forests.

Encryption at rest provides data confidentiality, but not authentication of identity or access control (permissions). See Authenticating Users and Protecting Documents for information about authentication and other forms of security in MarkLogic Server.

WARNING
If you cannot access your PKCS #11 secured wallet (or external KMS if you are using one), or lose your encryption keys, you will not be able to decrypt any of your encrypted data. There is no “mechanism” to recover the encrypted data. We recommend that you backup your encryption keys in a secure location. See Backup and Restore for more details.

14.4. Keystores - PKCS #11 Secured Wallet or External KMS

A keystore is a secure location where the actual encryption keys used to encrypt data are stored. The keystore for encryption at rest is a key management system (KMS). This keystore can be either the MarkLogic Server embedded PKCS #11 secured wallet, an external KMS that conforms to the KMIP-standard interface, or the native AWS KMS (Amazon Web Services Key Management System). The embedded keystore is installed by default when you install MarkLogic Server 9.0-x or later.

The MarkLogic Server embedded wallet uses a standard PKCS #11 protocol, using the PKCS #11 APIs. The wallet or another KMS, must be available during the MarkLogic Server startup process (or be bootstrapped from MarkLogic Server during start-up). You can also use any KMIP-compliant external keystore with MarkLogic Server or the native AWS KMS.

To configure an external KMS you will need the following information for your cluster:

- Host name
- Port number
- Client certificate
- Server certificate

If you are using the native AWS KMS, you will not need the Client certificate or the Server certificate. You will need the other information.

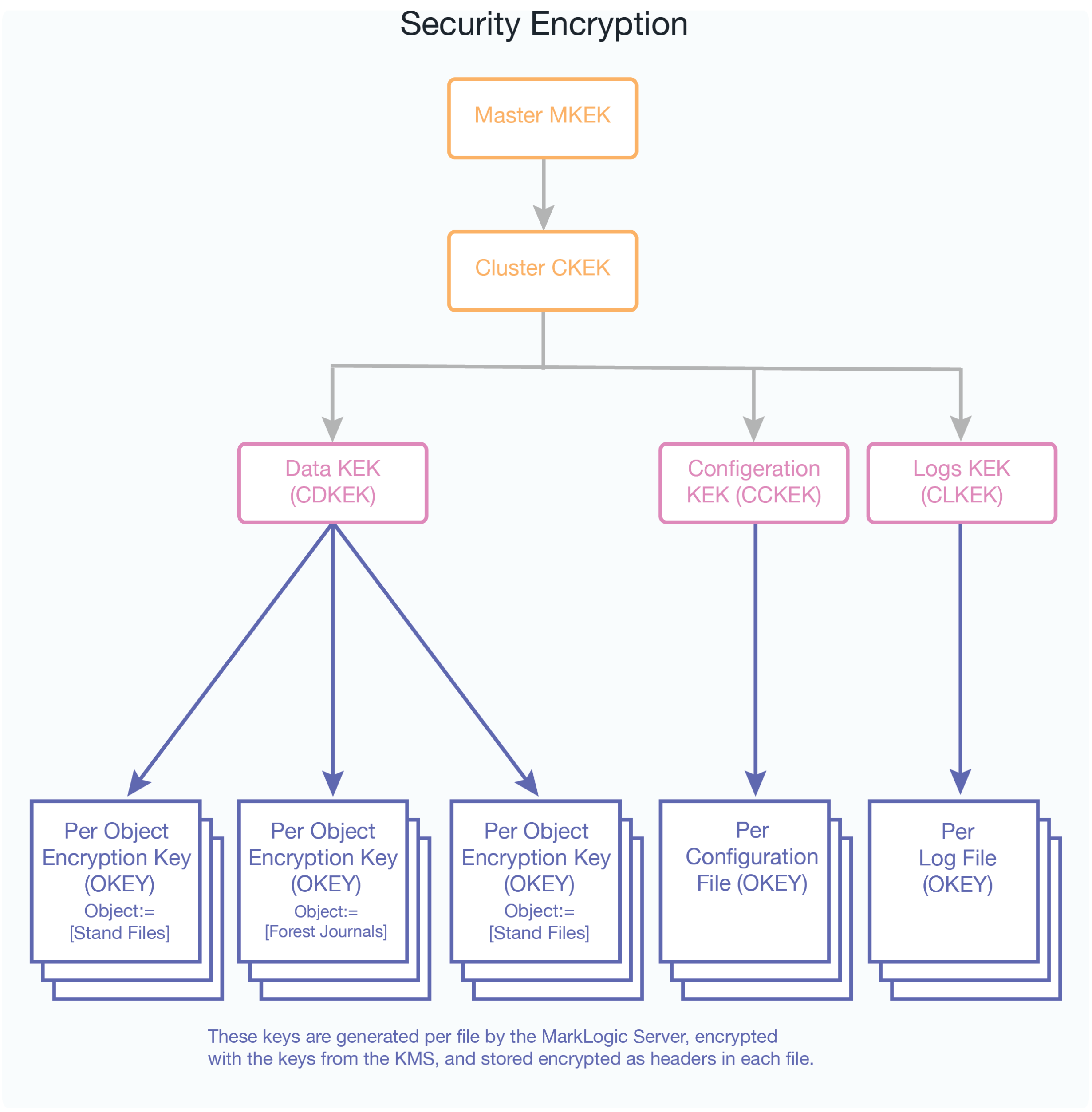
NOTE
If you plan to use an external key management system, configure the external KMS first, then turn on encryption in MarkLogic Server.

For details, see Configuring an External Keystore.

14.5. Encryption Key Hierarchy Overview

The following section provides an overview of the encryption key hierarchy used by MarkLogic Server encryption at rest to secure data. Keys in the encryption hierarchy wrap (or encrypt) those keys below them in the hierarchy. Three possible configurations of the encryption key hierarchy are described. The first is an idealized key hierarchy that provides a generic example. The second is an embedded KMS (the PKCS #11 secured wallet) configuration, and the third shows an external keystore management system (KMS) configuration.

You do not need to completely understand the details of the key hierarchy to use the encryption feature, but this section explains the general concepts involved.


Security Encryption
Master MKEK
Cluster CKEK
Data KEK (CDKEK)
Configuration KEK (CCKEK)
Logs KEK (CLKEK)
Per Object Encryption Key (OKEY) Object:= [Stand Files]
Per Object Encryption Key (OKEY) Object:= [Forest Journals]
Per Object Encryption Key (OKEY) Object:= [Stand Files]
Per Configuration File (OKEY)
Per Log File (OKEY)
These keys are generated per file by the MarkLogic Server, encrypted with the keys from the KMS, and stored encrypted as headers in each file.

The keystore contains the Master Key Encryption Key (MKEK). The keystore generates the Cluster Key Encryption Key (CKEK), which is enveloped (encrypted) with or derived from the Master Key Encryption Key. Both the Master Key Encryption Key and the Cluster Key Encryption Key reside in the keystore (key management system or KMS). These keys never leave the keystore and MarkLogic Server has no knowledge or control over these keys. The keys are referenced from the keystore by their key IDs.

The KMS can be either the internal keystore provided by MarkLogic Server or an external KMIP-compliant KMS; the same mechanism is used by both types of keystores. The configuration happens at the cluster level because there is one keystore configuration per cluster. The encryption feature is fully compliant with the KMIP standard and the Amazon KMS.

The external KMS provides even higher security. The key IDs are provided by the KMS and returned through a TLS tunnel after the MarkLogic Server-generated keys have been sent to the KMS and wrapped (encrypted). The actual encryption keys never leave the KMS.

There are multiple levels to the key hierarchy, each level wrapping (encrypting) the level below it. The KMS generates the Cluster Level Data Encryption Keys for data (CDKEK), configuration files (CCKEK), and log files (CLKEK). The corresponding key (CDKEK, CCKEK, or CLKEY) is used to encrypt (wrap) all the Object Encryption Keys (OKEY) generated by MarkLogic Server for each file, so that an encryption key protects each file, no matter what category (data, configuration files, logs).

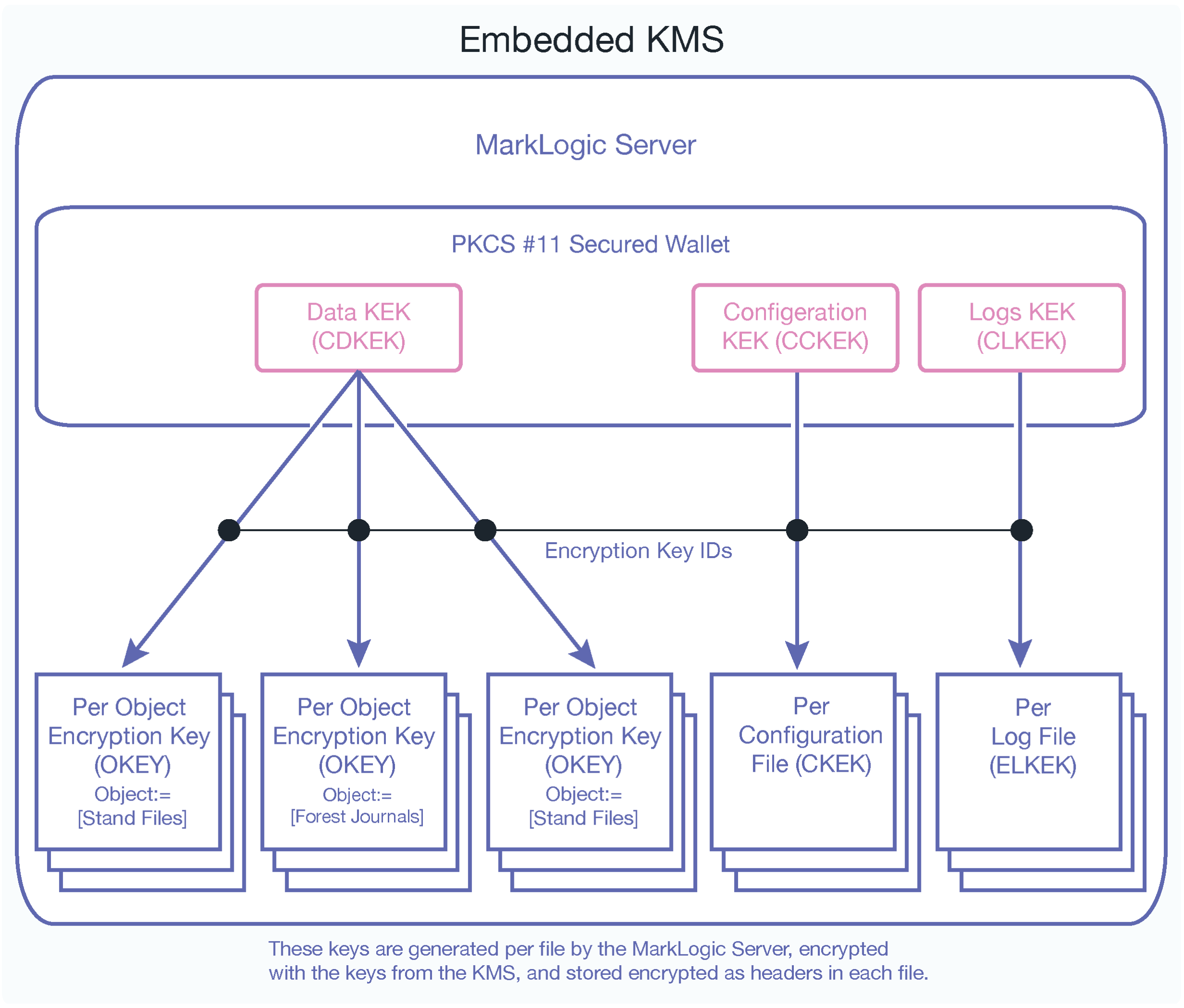
The Object Encryption Keys (OKEY) are randomly generated per file (for stands, journals, config files, log files, and so on.) wrapped (encrypted) with the corresponding keys (CDKEK, CCKEK, or CLKEK). So, an encryption key protects each file within a category (data, configuration files, logs).

For example, the Master Key Encryption Key (MKEK) wraps (encrypts) the Cluster Key Encryption Keys (CKEK), which in turn wraps (encrypts) the Data Key Encryption Key (CDKEK). The Data Key Encryption Key encrypts the Object Encryption Key (OKEY) for a file such as a stand. The keys at the bottom of the diagram are encrypted as headers in each file, wrapped (encrypted) with each of the keys above them in the hierarchy. Each of the three categories of objects (data, configuration files, and logs) has its own key encryption hierarchy.

Database backups are encrypted using a generated backup key (BKEK). This key is then encrypted with the cluster key (CDKEK). See Backup and Restore for more information about backups.

14.5.1. Embedded KMS Key Hierarchy

When you use the embedded PKCS #11 secured wallet provided with MarkLogic Server, the recommended key hierarchy would be similar to this illustration:


Embedded KMS
MarkLogic Server
PKCS #11 Secured Wallet
Data KEK (CDKEK)
Configeration KEK (CCKEK)
Logs KEK (CLKEK)
Encryption Key IDs
Per Object Encryption Key (OKEY) Object:= [Stand Files]
Per Object Encryption Key (OKEY) Object:= [Forest Journals]
Per Object Encryption Key (OKEY) Object:= [Stand Files]
Per Configuration File (CKEK)
Per Log File (ELKEK)
These keys are generated per file by the MarkLogic Server, encrypted with the keys from the KMS, and stored encrypted as headers in each file.

MarkLogic Server generates the Data Key Encryption Key (CDKEK), the Configuration Key Encryption key (CCKEK) and the Logs Key Encryption Key (CLKEK). The Data Key Encryption Key is then used to wrap the OKEYs for the database objects (journals, data files, and so on.). These keys are stored in the wallet (internal KMS). The key IDs are generated in the MarkLogic Server for encryption and decryption by the KMS (the PKCS #11 secured wallet in this case). The configuration happens at the cluster level because there is one keystore per cluster.

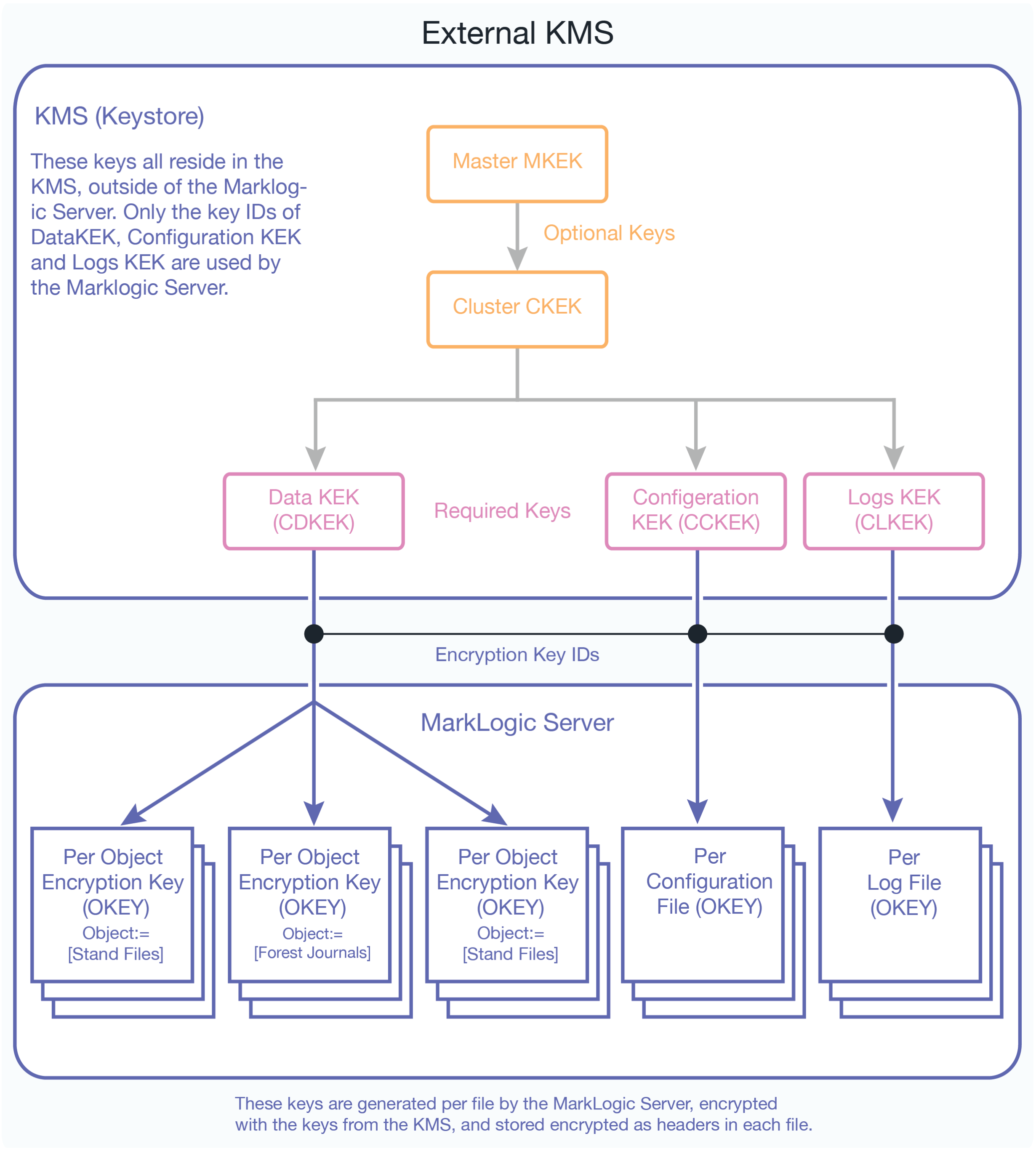
The individual Object Encryption Keys (OKEYs) are then randomly generated and used to directly encrypt individual files (journals, config files, log files, and so on). These keys (the OKEYs) are wrapped (encrypted) with the corresponding KEK for data, config, and logs. A unique key protects (encrypts) each file. The keys at the object levels are wrapped (encrypted by the keys above them) for each category.

For example, the Data Key Encryption Key (CDKEK) wraps (encrypts) the Object Encryption Key (OKEY) for a file such as a journal. The keys at the bottom of the diagram are encrypted (wrapped) by all the keys above them in the hierarchy, and then placed in the header for each file. In the case of the embedded KMS, there is only one CDKEK for the entire cluster - all databases in the cluster will use that key. When using the embedded KMS, it is not possible to use “per database” keys for encryption.

Database backups are encrypted using the locally generated backup key (BKEK) that is used to encrypt all of the files in the backup. The BKEK is then encrypted with the cluster data key (CDKEK) and then encrypted with the cluster key (CKEK). Additionally, you could encrypt this key with the BDKEY and a passphrase. See Backup and Restore for more information about backups.

14.5.2. External KMS Key Hierarchy

The external KMS provides even higher security, along with additional key management features. When you use an external key management system (KMS or keystore), the recommended key hierarchy deployment might look like this illustration:


External KMS
KMS (Keystore)
These keys all reside in the KMS, outside of the Marklogic Server. Only the key IDs of DataKEK, Configuration KEK and Logs KEK are used by the Marklogic Server.
Master MKEK
Optional Keys
Cluster CKEK
Data KEK (CDKEK)
Required Keys
Configeration KEK (CCKEK)
Logs KEK (CLKEK)
Encryption Key IDs
MarkLogic Server
Per Object Encryption Key (OKEY) Object:= [Stand Files]
Per Object Encryption Key (OKEY) Object:= [Forest Journals]
Per Object Encryption Key (OKEY) Object:= [Stand Files]
Per Configuration File (OKEY)
Per Log File (OKEY)
These keys are generated per file by the MarkLogic Server, encrypted with the keys from the KMS, and stored encrypted as headers in each file.

The keystore contains the Master Key Encryption Key (MKEK). The KMS generates or derives the Cluster Key Encryption Key (CKEK), which is enveloped (encrypted) with the Master Key Encryption Key. Both the Master Key Encryption Key and the Cluster Key Encryption Key reside in the KMS keystore. These keys never leave the keystore. MarkLogic Server has no knowledge or control over these keys. The keys are referenced from the keystore by their key IDs. The actual encryption keys never leave the KMS.

There are multiple levels to the key hierarchy in this deployment, each level wrapping (encrypting) the level below it. The KMS generates the cluster level encryption keys for data (CDKEK), configuration files (CCKEK), and log files (CLKEK). The corresponding KEK is used is used to encrypt (wrap) all the Object Encryption Keys (OKEY) generated by MarkLogic Server for each file, so that a unique key protects each file, no matter what category (data, configuration files, logs). A unique key protects each file within a category (data, configuration files, logs).

The corresponding KEK (for data, config, or logs) is used to encrypt (wrap) all the Object Encryption Keys (OKEY) generated by MarkLogic Server for each file, so that an encryption key protects each file, no matter what category (data, configuration files, logs).

For example, the Master Key Encryption Key (MKEK) wraps (encrypts) the Cluster Key Encryption Keys (CKEK), which in turn wraps (encrypts) the Data Key Encryption Key (CDKEK), then wraps (encrypts) the Object Encryption Key (OKEY) for a file such as a stand. The keys at the bottom of the diagram are encrypted (wrapped) by all the keys above them in the hierarchy, and then placed in the header for each file.

Database backups are encrypted using the BKEK, the locally generated backup KEK, the BKEK is encrypted with the CDKEK. Then the CDKEY may be encrypted or derived from the cluster key (CKEK). This last step is outside of the control of MarkLogic Server. You can also use a password or passphrase to encrypt and secure your backup. See Backup and Restore for more information about backups and the use of a passphrase to secure your backup.

NOTE
If you plan to use an external key management system, configure the external KMS first, and then turn on encryption in the MarkLogic Server.

14.6. Example—Encryption at Rest

This section describes a scenario using encryption at rest to encrypt a database. This example is for informational purposes only. It is not meant to demonstrate *the* correct way to set up and use encryption at rest, as your situation is likely to be unique. However, it demonstrates how encryption at rest works and may give you ideas for how to configure your own encryption at rest security model:

To set up encryption at rest for this scenario, you will need Admin privileges. You will need access to both the MarkLogic Server Admin Interface and Query Console.

To run through the example, perform the steps in each section.

14.6.1. Set Up Encryption Example

Install MarkLogic Server 9.0-1 or later. Encryption at rest options are not available in earlier versions of MarkLogic Server. You must explicitly select which data (databases, configuration files, log files, or entire clusters) you want to have encrypted. This example shows how to set up encryption for a single database.

NOTE
The Security database or other databases used by MarkLogic Server will not be encrypted by default. Existing data can be encrypted by forcing a merge or a reindex of the database.

See Configuring Compartment Security for more details.

14.6.2. Encrypt a Database

For this example, we will use the Admin Interface to set up encryption for the Documents database.

1. Select **Databases** from the left tree menu in the Admin Interface.
2. Click on the **Documents** database.
3. On the Database Configuration page, next to data encryption, select on from the drop-down menu. (The other options are `default-cluster` and `off`.)
4. Click **OK**.

If you select `default-cluster`, encryption for that database will default to whatever encryption option has been set for the cluster as a whole. If the cluster is set to encrypt data, this database will be encrypted. If encryption has not been turned on for the cluster, this database will not be encrypted if `default-cluster` is selected. See Cluster Encryption Options for details.

As you access data in your database, it will be encrypted when it is written back to disk. You can view the encryption progress on the Database Status page by looking at the Size and Encrypted Size numbers.

NOTE
To encrypt the existing data in your database, you will need to re-index your database. On the Database Configuration page, click the **reindex** button at the top of the page (below the "**OK**" button), and then click **ok**. You can also force a merge of the database to encrypt the data.

Encryption of large databases will take some time initially. Updates and changes to the database will be fairly transparent to the user after initial encryption. The Size and Encrypted Size numbers will be equal when the encryption process is complete.

14.6.3. Test It Out

Using Query Console, you can run a simple query to verify that the Documents database has encryption turned on.

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

let $config := admin:get-configuration()
return admin:database-get-data-encryption($config, xdmp:database("Documents"))

=>
on
```

You can also check the Size and Encrypted Size numbers on the Database Status page. These numbers will be equal when the encryption process is complete and the entire database is encrypted.

14.6.4. Turn Off Encryption for a Database

1. Select **Databases** from the left tree menu in the Admin Interface.
2. Click on the **Documents** database to turn off encryption.
3. On the Database Configuration page, next to data encryption, select `off` from the drop-down menu.
4. Click **OK**.

To verify that encryption is turned off, run this query in Query Console:

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

let $config := admin:get-configuration()
return admin:database-get-data-encryption($config, xdmp:database("Documents"))

=>
off
```

To decrypt the existing data in your database, you will need to re-index your database. On the Database Configuration page, click the **reindex** button and then click **ok**.

NOTE
You can also decrypt the data by forcing a merge on the database to decrypt its contents. This process may take a while.

14.7. Configuring Encryption at Rest

Install MarkLogic Server version 9.0-x or later. The encryption at rest feature and the PKCS #11 secured wallet are installed by default. You can configure encryption at rest for databases (data encryption), log files (log encryption) and configuration files (config encryption). The encryption feature will need to be configured and enabled for your data to be encrypted.

When you start up MarkLogic Server for the first time after installation, the `keystore.xml` file will be loaded first. It contains the encryption key IDs. After loading the `keystore.xml` configuration, MarkLogic Server validates connectivity to the KMS (local or external) and the validity of the keys stored in `keystore.xml`. Once validated, encryption keys will be loaded and decrypted. Normal startup then continues. If configuration files are encrypted, the file layer will decrypt them as they are being loaded, making the encryption transparent to the cluster.

NOTE
If a node in your cluster is offline for any reason, wait until the host comes back online to make any changes to your encryption at rest settings. Do not change your encryption settings while a host is offline.

14.7.1. Database Encryption Options

You can configure encryption for each database on the Database Configuration page in the Admin Interface. Encryption at rest can be separately enabled per database, or at the cluster level by setting the database encryption to default to the cluster encryption settings. The encryption options for databases are shown in this table:

Encryption	Encryption Option: Default-Cluster	Encryption Option: On	Encryption Option: Off
Database encryption	encryption defaults to cluster setting	encryption enabled for database	encryption off, unless cluster encryption is set to force encryption

With encryption enabled, files are encrypted as they are ingested into the database, or when those files are written back to disk. If you want to encrypt existing data in a database either reindex the database or force a merge on the database. This will take a few minutes depending on the size of database. See Cluster Encryption Options

NOTE
Large binary files are only encrypted during initial ingestion into the database. If you want to encrypt existing large binary files already loaded into MarkLogic Server prior to turning on encryption, you must reindex the database or force a merge.

1. To configure database encryption, go to the Admin Interface and click **Databases** in the left navigation tree.
2. Click on the database you want to encrypt.
3. On the Database Configuration page, next to data encryption, select on from the drop-down menu. (The other options are `default-cluster` and `off`.)
4. Click **OK** when you are done.

14.7.2. Configure Cluster Encryption

You can set cluster encryption options for configuration files and log files, and also set or override the encryption options for databases on the Cluster Configuration page.

Configuration File and Log File Encryption Options

Encryption at rest for configuration files and/or log files is done on the Cluster Configuration page in the Admin Interface. Navigate to this page by choosing **Clusters** from the left tree menu, clicking the cluster name, and then clicking the **Configure** tab.

The encryption options are shown in this table:

File Type	Cluster Encryption Setting: Default On	Cluster Encryption Setting: Default Off	Cluster Encryption Setting: Force
Configuration files	encrypt	do not encrypt	encrypt
Log files	encrypt	do not encrypt	encrypt

NOTE
The `keystore.xml` and `hsm.cfg` files are never be encrypted because they are configuration for the Keystore. The `servers.xml` file is not immediately encrypted until a server (apps server) is updated, a new server is created, or an existing server is deleted. This is because these actions trigger a restart of the MarkLogic Server.

Cluster configuration settings for encryption at rest interact with the encryption settings for databases. You can separately configure encryption for each database on the Database Configuration page in the Admin Interface or set database encryption to default to the cluster encryption settings.

NOTE
The database encryption configuration settings take precedence unless the cluster Force Encryption option is set. If Force Encryption is on, configuration files and log files will be encrypted. Please check all database encryption settings to ensure that they are set correctly.

The following table shows the interaction between the cluster configuration options and the database configuration options. There are three possible database encryption settings and three possible cluster encryption settings. The cell where the row and column intersect shows the outcome of that configuration combination.

Database Encryption Setting	Cluster Encryption Setting: Force Encryption	Cluster Encryption Setting: Default On	Cluster Encryption Setting: Default Off
Default to cluster	encrypt	encrypt	do not encrypt
On	encrypt	encrypt	encrypt
Off	encrypt	do not encrypt	do not encrypt

The Force Encryption option in the Cluster Encryption Settings will force encryption for all of the databases in the cluster. If the Cluster Encryption Setting is Force Encryption (or Default On), or the Database Encryption Setting is On, then the database will be encrypted.

14.7.3. Cluster Encryption Options

You can either configure encryption for the embedded keystore (the PKCS #11 secured wallet) or for an external KMIP-compliant keystore using the Admin Interface. Use the Edit Keystore Configuration page to configure encryption at rest for a cluster. Using this page, you can configure data encryption, configuration file encryption, or encryption of log files.

1. To configure encryption using the embedded keystore in the Admin Interface, click **Clusters** in the left navigation tree.
2. Underneath **Clusters**, click the name of the cluster you want to configure.
3. Click the **Keystore** tab to configure the keystore for encryption at rest.
4. Use the Data Encryption, Config Encryption, Logs Encryption, and KMS Type fields to configure encryption for data, config files, and/or log files. An explanation of the fields is included below.

Setting	Description
data encryption	Specifies whether or not encryption is enabled for user data. These are the options: `force` — Force encryption for all data in the cluster. The database configuration cannot overwrite this setting. `default-on` — By default encryption is on. The database configuration can overwrite this setting. `default-off` — By default encryption is off. The database configuration can overwrite this setting.
config encryption	Specifies whether or not encryption is enabled for configuration files.
logs encryption	Specifies whether or not encryption is enabled for log files.
kms type	Specifies whether the KMS is internal to MarkLogic Server or an external KMS. A keystore is a secure location where the actual encryption keys used to encrypt data are stored. The keystore for encryption at rest is a key management system (KMS). This keystore can be either the MarkLogic Server embedded PKCS #11 secured wallet, or an external third party KMS.

Beneath these options, the Internal KMS and External KMS tabs allow you to specify further options. For the Internal KMS there are these options:

Setting	Description
backup option	The internal KMS is automatically included in backups unless you change the default setting of “include” to “exclude”.
internal data encryption key id	The UUID that identifies the encryption key from the internal KMS that is to be used to encrypt data files.
internal config encryption id	The UUID that identifies the encryption key from the internal KMS that is to be used to encrypt config files.
internal logs encryption id	The UUID that identifies the encryption key from the internal KMS that is to be used to encrypt log files.

5. Click **OK** when you are done.

NOTE

Adding or changing any encryption information will require a restart of all of the hosts in the cluster.

Changing the Internal KMS Password

You can change the password for the internal KMS using the Change Internal KMS Password screen. To change the internal KMS password do the following:

1. Click **Clusters** in the left navigation tree and click the name of the cluster that has the KMS keystore with password that you want to change.
2. Click the **Keystore** tab to open the Edit Keystore Configuration page. Click the **change password** button on the Edit Keystore Configuration page. This opens the Change Internal KMS Password page.
3. Enter the current password in the first field, then enter the new password in the second field. Confirm the new password by entering it again in the third field.
4. Click **OK** when you are done.

14.7.4. Using an Alternative PKCS #11 Device

MarkLogic Server uses SoftHSM as its default hardware security module (HSM). This section describes the process of setting up an alternate hardware security module if you want to use a PKCS #11 HSM (or any other PKCS #11-compliant HSM) by following these steps before starting MarkLogic Server for the first time.

NOTE
This process will only work on a clean data directory with a first time install.

1. The PKCS#11 devices must not be initialized, and no PIN should be set, MarkLogic Server will initialize it and set a PIN.
2. Set environment variable=`MARKLOGIC_P11_DRIVER_PATH` to the PKCS#11 library you want to use.
3. Start MarkLogic Server for the first time.
4. Verify no error messages are logged during startup.

Saving the Embedded KMS to a Different Location

Use the options available in `admin:cluster-set-keystore-wallet-location` to change the location of the backup for the internal wallet.

```
let $dir-name := "/sotfhsm/wallet"
let $config := admin:get-configuration()
return
  admin:cluster-set-keystore-wallet-location($config,$dir-name)
```

The `admin:cluster-set-keystore-wallet-location` function will set the backup location for embedded KMS.

14.7.5. Configure Encryption Using XQuery

Instead of using the Admin Interface, you can configure encryption for your MarkLogic Server instance using XQuery.

In Query Console, you can use `admin:cluster-set-data-encryption` to turn on data encryption for the current database:

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

let $config := admin:get-configuration()
return
  admin:cluster-set-data-encryption($config,"default-on")
```

For example, to set the encryption for log files at cluster level:

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

let $config := admin:get-configuration()
return
  admin:cluster-set-logs-encryption($config, "default-on")

(: returns the new configuration element -- use admin:save-configuration
to save the changes to the configuration or pass the configuration to
other Admin API functions to make other changes.  :)
```

To see whether encryption is turned on for log files, you can run this XQuery in the Query Console:

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

let $config := admin:get-configuration()
return
  admin:cluster-get-logs-encryption($config)
=>
on
 (: returns the encryption setting for log files:)
```

14.7.6. Configure Encryption Using REST

You can use REST Management APIs to work with encryption at rest.

`GET:/manage/v2/databases/{id|name}/properties`

This command gets the current properties of the Documents database, including the encryption status and encryption key ID in JSON format:

```
$ curl -GET --anyauth -u admin:admin \
  -H "Accept:application/json,Content-Type:application/json" \
  http://localhost:8002/manage/v2/databases/Documents/properties
```

Returns

```
{"database-name":"Documents", "forest":["Documents"],
"security-database":"Security", "schema-database":"Schemas",
"triggers-database":"Triggers", "enabled":true,
"data-encryption":"off", "encryption-key-id":"",
```

The same command in XML format:

```
$ curl -GET --anyauth -u admin:admin \
  -H "Accept:application/xml,Content-Type:application/xml" \
  http://localhost:8002/manage/v2/databases/Documents/properties
```

Returns

```
<database-properties xmlns="http://marklogic.com/manage">
  <database-name>Documents</database-name>
  <forests>
    <forest>Documents</forest>
  </forests>
  <security-database>Security</security-database>
  <schema-database>Schemas</schema-database>
  <triggers-database>Triggers</triggers-database>
  <enabled>true</enabled>
  <data-encryption>on</data-encryption>
  <encryption-key-id/>
...
</database-properties>
```

`GET:/manage/v2/security/properties`

This command returns the current encryption status, along with other properties including encryption key ID, for localhost in JSON format:

```
$ curl -GET --anyauth -u admin:admin \
  -H "Accept:application/json,Content-Type:application/json" \
  http://localhost:8002/manage/v2/security/properties
```

Returns:

```
{"keystore":{"data-encryption":"default-off",
"data-encryption-key-id":"091fd9a0-f090-4c7e-91ca-fedfe21dbfef",
"config-encryption":"off", "config-encryption-key-id":"",
"logs-encryption":"off", "logs-encryption-key-id":"",
"host-name":"LOCALHOST", "port":9056}}
```

Here is the same version of the command, this time returning XML:

```
$ curl -GET --anyauth -u admin:admin \
  -H "Accept:application/xml,Content-Type:application/xml" \
  http://localhost:8002/manage/v2/security/properties
```

Returns:

```
<security-properties xsi:schemaLocation="http://marklogic.com/manage/security/properties
manage-security-properties.xsd" xmlns="http://marklogic.com/manage/security/properties"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <keystore>
    <data-encryption>default-off</data-encryption>
    <data-encryption-key-id>8d0b07d8-b655-4408-affd-e49a2ece0af3
      </data-encryption-key-id>
    <config-encryption>off</config-encryption>
    <config-encryption-key-id/>
    <logs-encryption>off</logs-encryption>
    <logs-encryption-key-id/>
    <host-name>LOCALHOST</host-name>
    <port>9056</port>
  </keystore>
</security-properties>
```

`POST:/manage/v2/protected-paths`

This command sets the protected path for `//d` with `read` permissions for `manage-user`:

```
$ curl -POST --anyauth -u admin:admin \
  -d @file.xml -H "Content-Type:application/xml" \
  http://localhost:8002/manage/v2/protected-paths
```

Here is the payload (file.xml):

```
<protected-path-properties xmlns="http://marklogic.com/manage/protected-path/properties">
  <path-expression>//d</path-expression>
  <path-namspaces/>
  <permissions>
    <permission>
      <role-name>manage-user</role-name>
      <capability>read</capability>
    </permission>
  </permissions>
</protected-path-properties>
```

Here is the same operation in JSON:

```
curl -X POST --anyauth -u admin:admin \
-d @file.json -H "Content-Type:application/json" \
http://localhost:8002/manage/v2/protected-paths
```

Here is the payload (file.json):

```
{
"path-expression": "//e",
"path-namespace": [],
"permission": [{
"role-name": ["manage-user"],
"capability": "read"
}]
}
```

`PUT:/manage/v2/databases/{id|name}/properties`

This command will turn on encryption for the Documents database:

```
$ curl -X PUT --anyauth -u admin:admin -d '{"data-encryption":"on"}' \
  -H "Content-Type:application/json" \
  http://localhost:8002/manage/v2/databases/Documents/properties
```

Export Wallet

To export the embedded KMS (the PKCS #11 secured wallet) using REST, you can use this form in XQuery:

```
POST manage/v2/security?
operation=export-wallet&filename=/my/test..wallet&password=test
```

As a curl command (using `MANAGEADMIN="admin"` and `MANAGEPASS="admin"`) it would look like this:

```
curl -v -X POST  --anyauth --user $MANAGEADMIN:$MANAGEPASS \
  --header "Content-Type:application/xml" \
-d@data/security/export-wallet.xml \
  http://$host:8002/manage/v2/security
```

Where `export-wallet.xml` is:

```
<export-wallet-operation xmlns="http://marklogic.com/manage/security">
  <operation>export-wallet</operation>
  <filename>/tmp/mywallet.txt</filename>
  <password>mypassword</nassword>
</export-wallet-operation>
```

Or you can use this form for JavaScript:

```
POST manage/v2/security
{"operation":"export-wallet","filename":"/my/test.wallet","password":"test"}
```

As a curl command (using `MANAGEADMIN="admin"` and `MANAGEPASS="admin"`) it would look like this:

```
curl -v -X POST  --anyauth --user $MANAGEADMIN:$MANAGEPASS \
  --header "Content-Type:application/json" \
-d@data/security/export-wallet.json \
  http://$host:8002/manage/v2/security
```

Where `export-wallet.json` is:

```
{
    "operation":"export-wallet",
    "filename":"/tmp/mywallet.tmp",
    "password":"mypassword"
}
```

NOTE

The export wallet operation saves the wallet to a directory on the server on which MarkLogic Server is running. Similarly, the import wallet operation imports from the filesystem on which MarkLogic Server is running.

Import Wallet

To import the embedded KMS (the PKCS #11 secured wallet) using REST, you can use this form in XQuery:

```
POST manage/v2/security?
operation=import-wallet&filename=/my/test.wallet&password=test
```

As a curl command (using `MANAGEADMIN="admin"` and `MANAGEPASS="admin"`) it would look like this:

```
curl -v -X POST  --anyauth --user $MANAGEADMIN:$MANAGEPASS \
  --header "Content-Type:application/xml" \
-d@data/security/import-wallet.xml \
  http://$host:8002/manage/v2/security
```

Where `import-wallet.xml` is:

```
<import-wallet-operation xmlns="http://marklogic.com/manage/security">
  <operation>import-wallet</operation>
  <filename>/tmp/mywallet.txt</filename>
  <password>mypassword</password>
</import-wallet-operation>
```

Or you can use this form for JavaScript:

```
POST manage/v2/security
{"operation":"import-wallet","filename":"/my/test.wallet","password":"test"}
```

As a curl command (using `MANAGEADMIN="admin"` and `MANAGEPASS="admin"`) it would look like this:

```
curl -v -X POST  --anyauth --user $MANAGEADMIN:$MANAGEPASS \
  --header "Content-Type:application/json" \
-d@data/security/import-wallet.json \
  http://$host:8002/manage/v2/security
```

Where `import-wallet.json` is:

```
{
    "operation":"import-wallet",
    "filename":"/tmp/mywallet.tmp",
    "password":"mypassword"
}
```

NOTE

MarkLogic will only import keys generated by the embedded MarkLogic Server KMS.

14.8. Key Management

Encryption key management for the embedded KMS (the PKCS #11 secured wallet) is handled automatically by MarkLogic Server. Keys are never purged from the wallet, which is encrypted by a MarkLogic Server-generated key activated by a passphrase. The administrator’s password is used as the initial passphrase.

NOTE

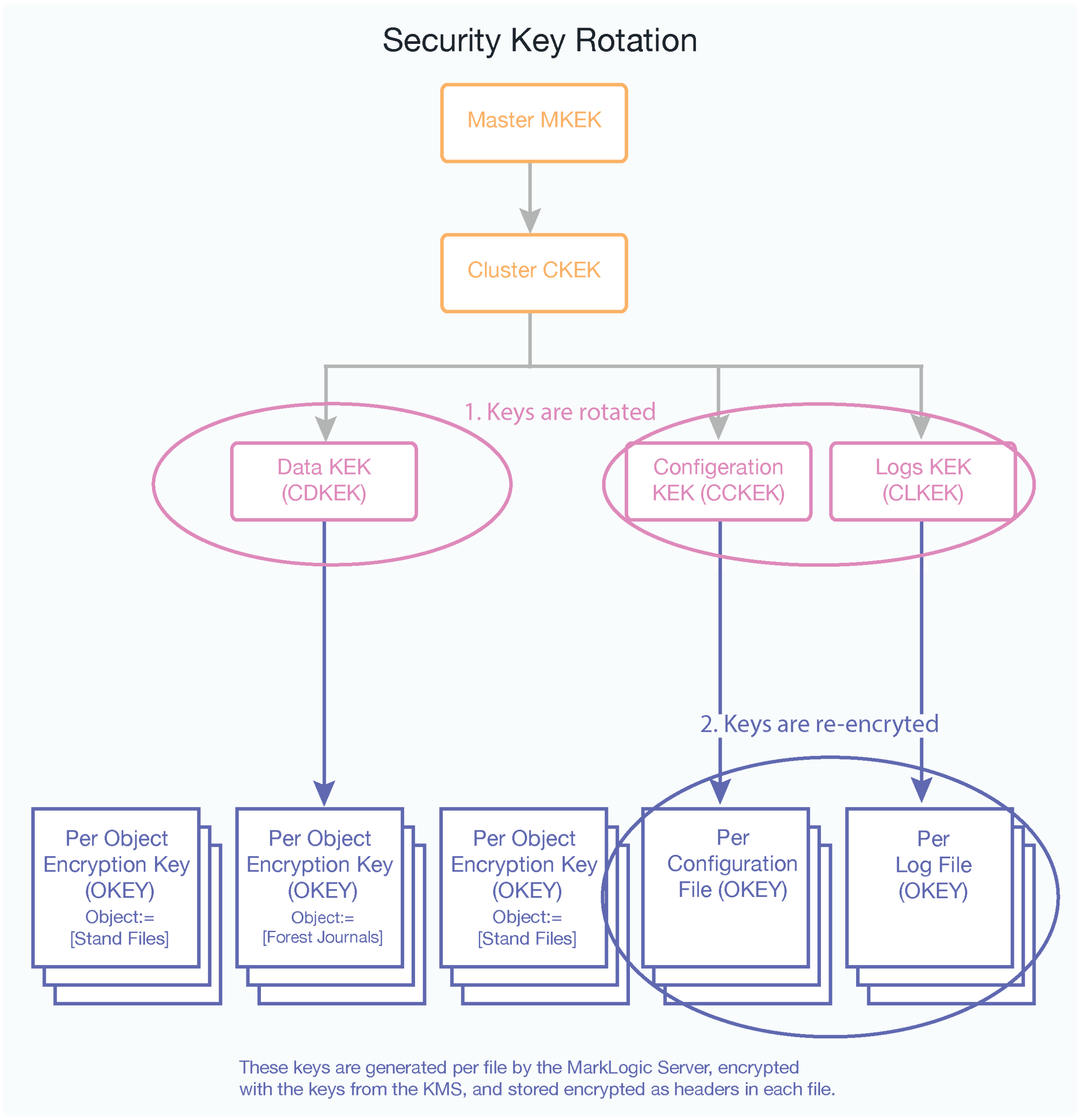
By default, the keystore passphrase is set to the admin password. We strongly recommend that you set a new, different passphrase before turning on encryption. Using a separate passphrase for admin and the keystore helps support the strong security principle called “Separation of Duties”.

This passphrase can be changed using either the XQuery (`xdmp:keystore-set-kms-passphrase`) or JavaScript (`xdmp.keystoreSetKmsPassphrase`) built-ins. As part of key management, you may want to export, import, or rotate encryption keys. MarkLogic Server provides built-in functions for exporting and importing encryption keys, and manually rotating encryption keys. If you require additional key management functionality, you may want to consider an external key management system. See Configuring an External Keystore for more information.

If you believe that an encryption key has been compromised, you should force a merge or start a re-index of your data to change/update the encryption keys. See Key Rotation for more about updating encryption keys.

14.8.1. Key Rotation

For the internal wallet, key encryption keys (KEK) can be manually rotated. Keys can be manually rotated at regular intervals or if an encryption key has been compromised. This type of key rotation can be triggered on individual encryption categories (configuration, data, logs) using MarkLogic Server built-in functions.


Security Key Rotation
Master MKEK
Cluster CKEK
1. Keys are rotated
Data KEK (CDKEK)
Configeration KEK (CCKEK)
Logs KEK (CLKEK)
2. Keys are re-encryted
Per Object Encryption Key (OKEY) Object:= [Stand Files]
Per Object Encryption Key (OKEY) Object:= [Forest Journals]
Per Object Encryption Key (OKEY) Object:= [Stand Files]
Per Configuration File (OKEY)
Per Log File (OKEY)
These keys are generated per file by the MarkLogic Server, encrypted with the keys from the KMS, and stored encrypted as headers in each file.

There are two steps to key rotation. First, rotating the KEK keys (using AES 256 symmetric encryption) used to envelope the object file encryption keys, and second, re-encrypting the object file encryption keys (also using AES 256 symmetric encryption).

After calling the built-in function to rotate encryption keys, all new data will be written to disk using the new key encryption key. Old data will be migrated as it is re-written to disk. If you wish to force re-encryption using the new key, you can either force a merge or re-index the forest.

At the local, host level, you can manually rotate the data keys, configuration keys, and the logs keys (CDKEK, CCKEK, CLKEK) using these APIs:

- `admin:cluster-rotate-config-encryption-key-id`
- `admin:cluster-rotate-data-encryption-key-id`
- `admin:cluster-rotate-logs-encryption-key-id`

NOTE
These key rotation functions are only available for the MarkLogic Server internal KMS (the PKCS #11 secured wallet) and not for any keys that are managed by an external KMS.

At the cluster level, to manually rotate the cluster-level keys use these APIs:

- `admin:group-get-rotate-audit-files`
- `admin:group-get-rotate-log-files`
- `admin:group-set-rotate-audit-files`
- `admin:group-set-rotate-log-files`

NOTE
When you are using an external KMS, MarkLogic Server does not have access to the envelope key, it only has access to the key ID, and asks for the KMS to open the envelope.

Manual Key Rotation

The intermediate fast rotation keys enable immediate envelope key rotation with a minimum of I/O. File level keys can be rotated at any time by forcing a merge. Log rotation and configuration file updates use new keys. Old logs, backups, and configuration files are not re-encrypted.

The internal KMS (the PKCS #11 secured wallet) follows these steps for fast key rotation:

1. User sends rotation key command to MarkLogic Server (for example, `admin:cluster-rotate-data-encryption-key-id()`).
2. MarkLogic Server requests a new data encryption key (CDKEK, CCKEK, CLKEK - the cluster-level encryption keys) from the internal KMS.
3. Only the fast rotation keys are re-encrypted with the new data encryption keys (CDKEK, CCKEK, CLKEK).

An external KMS follows these steps for fast key rotation:

1. The external KMS creates new KEK key (CDKEK, CCKEK, CLKEK - the cluster-level encryption keys).
2. User updates the UUIDs in MarkLogic Server. See Set Up an External KMIP KMS with MarkLogic Server Encryption for UUID details.
3. MarkLogic Server sends a Fast Rotation Key (FRKEK) to the KMS.
4. The external KMS sends new enveloped key back to MarkLogic Server.
5. The enveloped key is saved to disk, per file.

NOTE
Expired keys can be used for decryption but not encryption. Expired keys may be needed for decrypting backups.

14.8.2. Export and Import Encryption Keys

The ability to export and import key encryption keys (KEK) from the PKCS #11 secured wallet (the embedded KMS) is useful when you want to clone a cluster. Exporting a key encryption key (KEK) is restricted to cluster-level keys (CDKEK, CCKEK, CLKEK) and requires a passphrase and a filepath. The data will be exported (encrypted with the passphrase) into a file at the location specified by the filepath.

To export a keystore from the embedded KMS:

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

xdmp:keystore-export("Unique passphrase", "/backups/MarkLogic.wallet.bak")
=>
true
```

To import a keystore into the embedded KMS:

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

xdmp:keystore-import("Unique passphrase", "/backups/MarkLogic.wallet.bak")
=>  true
```

Key encryption keys can only be imported from MarkLogic Server-exported files. Imported keys can only be used for decryption. The import requires the passphrase that was provided at the time of the export.

WARNING

If a duplicate key is supplied during the import, the import will be rejected. Duplicate keys can be caused by importing the keystore twice.

14.8.3. Key Deletion and Key Revocation

For these functions you will need to use an external keystore (KMS).

14.9. Configuring an External Keystore

An external key management system (KMS) or keystore offers additional security for your encryption keys, along with key management capabilities like automatic key rotation, key revocation, and key deletion. If you want the ability to perform these tasks, you will need an external KMS. MarkLogic Server Encryption at Rest supports KMIP 1.2 compliant KMS servers and Amazon's KMS.

NOTE

The use of an external Key Management System (KMS) or keystore with encryption at rest, requires an Advanced Security License, in addition to the regular MarkLogic Server license.

When using an external KMS, usually there is a security administrator role separate from the MarkLogic Server administrator. The security administrator would be the role setting up and configuring the external keystore. The MarkLogic Server administrator can also perform this task, but for greater security it is recommended that the separate security administrator configure the KMS.

NOTE
Having a separate security administrator follows an important security principle called “Separation of Duties” and is recommended by security experts.

This section covers setting up MarkLogic Server encryption for use with an external key management system from the MarkLogic Server Admin Interface on the MarkLogic Server host. You don’t need to have MarkLogic Server encryption turned on for your cluster while you are setting up and configuring the external key management system.

NOTE
If you plan to use an external key management system, we recommend that you configure the external keystore first, and then turn on encryption in the MarkLogic Server.

The installation process for the external keystore will vary depending on the type of external KMIP-compliant KMS you plan to use. A security administrator must configure the external keystore using the administration set up tools that come with the external KMS. This section provides a high-level overview of the process from the MarkLogic Server point of view.

14.9.1. Types of External KMS Deployments

There are a variety of types of external key management systems. An external key management system deployment may be one of the following types:

- A virtual KMS instance running in a VM (virtual machine) environment, or in a private or public cloud
- A physical appliance running a KMS server
- A dedicated FIPS 140-2 Level 3 appliance
- A dedicated hardened FIPS 140-2 Level 4 appliance

These systems are listed by increasing levels of security.

14.9.2. Using MarkLogic Server Encryption with AWS Key Management System

Amazon Web Services (AWS) provides a key management system (KMS) that you can use with MarkLogic Server encryption at rest to encrypt your data. The AWS KMS is supported for customers running their cluster on AWS in the cloud. You must set up your AWS KMS encryption keys and configure the encryption key IDs in your MarkLogic Server before using the AWS KMS.

To set up the AWS key management system, first set up your AWS instance. See Getting Started with MarkLogic Server on AWS and Overview of MarkLogic Server on AWS in the *MarkLogic Server on Amazon Web Services (AWS) Guide* for details.

The AWS KMS keys (data, config, and log encryption keys) needed to encrypt and decrypt data must be configured in MarkLogic Server before using encryption.

You cannot use the master key and roles from the MarkLogic Server KMS to access the AWS KMS, so you will need to have a Key Administrator specify access to the AWS KMS keys on a per-key basis tied to the user's IAM role. The Key Administrator can specify access using the Encryption Keys section of the IAM AWS management console. See the next section (Encryption on EBS Volumes) for details and the AWS documentation regarding key policies for more information.

WARNING
If an encryption key stored in the AWS KMS is disabled for any reason, it cannot be used for encryption or decryption, and MarkLogic Server loses access to any data encrypted with the disabled key. Deleting a key will lead to permanent data loss as deleted keys can never be recovered. Any keys created in the AWS KMS are cluster management keys and should never be deleted. See https://docs.aws.amazon.com/kms/latest/developerguide/enabling-keys.html for more information.

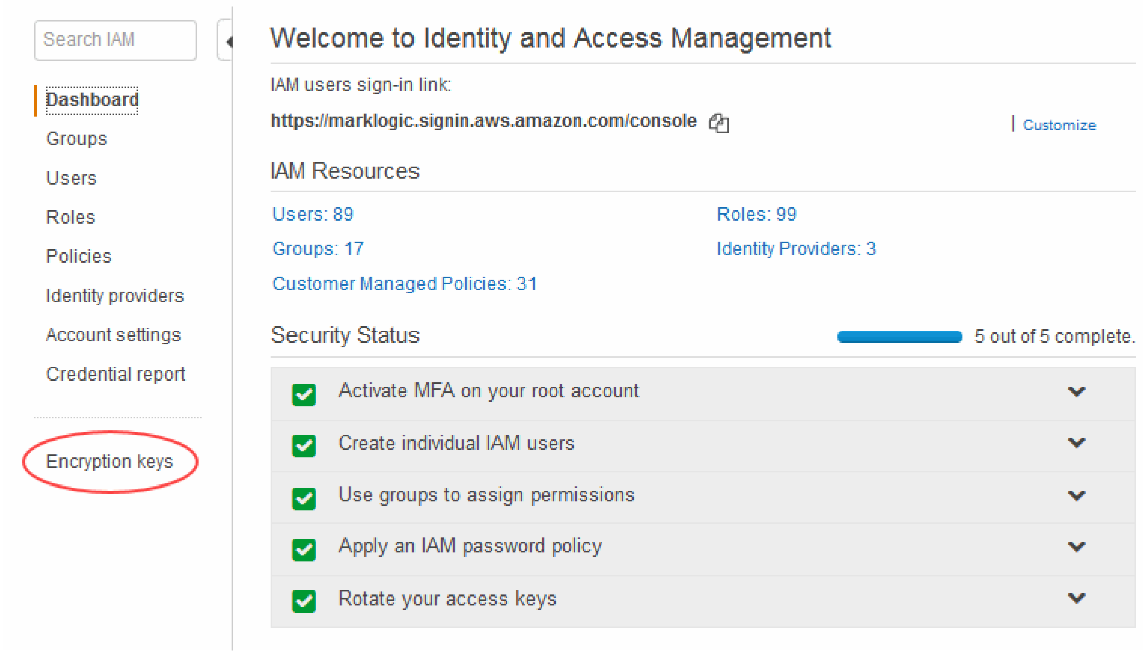
AWS KMS on EC2

If your cluster is running on AWS, the IAM role associated with the EC2 instance running MarkLogic Server is used to access the AWS KMS on behalf of MarkLogic Server. The hostname and port number will be automatically entered in the correct fields in the Keystore tab of the Admin Interface.

The key policy is tied to the user's IAM role. To set up your IAM role and privileges, see Creating an IAM Role in the *MarkLogic Server on Amazon Web Services (AWS) Guide*.

Once you have set up your MarkLogic Server (and IAM roles if necessary), follow these steps:

1. In AWS, navigate to the AWS IAM Management Console.
2. Click **Encryption keys** at the bottom of the left navigation bar.



Welcome to Identity and Access Management

IAM users sign-in link:
https://marklogic.signin.aws.amazon.com/console

IAM Resources

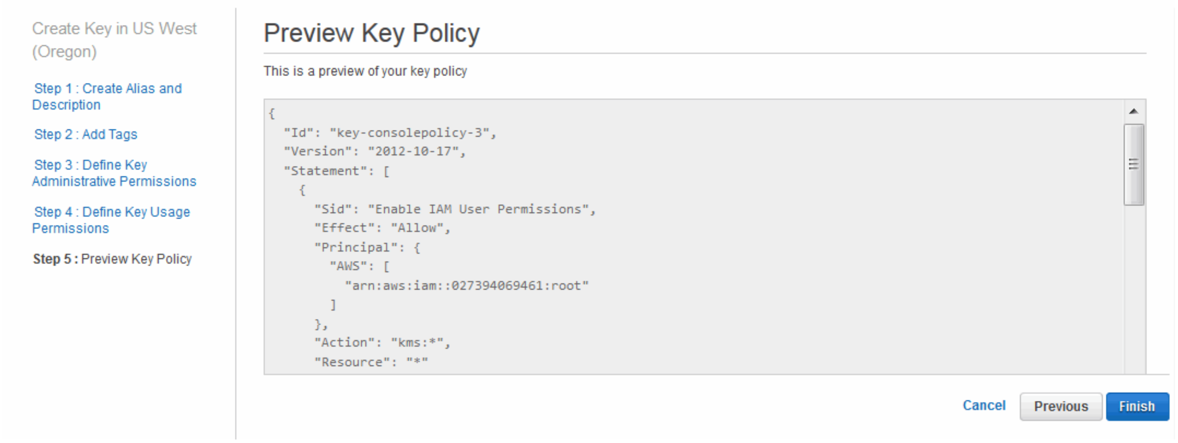
Users: 89
Roles: 99
Groups: 17
Identity Providers: 3
Customer Managed Policies: 31

Security Status 5 out of 5 complete.

✓ Activate MFA on your root account
✓ Create individual IAM users
✓ Use groups to assign permissions
✓ Apply an IAM password policy
✓ Rotate your access keys

3. In the next screen, pick a region (in the same region as your MarkLogic Server instance).
4. Create the key following the steps indicated. In the next step, be sure to give each key you create a descriptive name so that you can tell them apart.

5. In the last step of this process, you can preview the key policy you just created. Be sure to authorize your MarkLogic Server instance to use the key.


Create Key in US West (Oregon)
Step 1 : Create Alias and Description
Step 2 : Add Tags
Step 3 : Define Key Administrative Permissions
Step 4 : Define Key Usage Permissions
Step 5 : Preview Key Policy
Cancel
Previous
Finish

Preview Key Policy

This is a preview of your key policy

```
{
  "Id": "key-consolepolicy-3",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Enable IAM User Permissions",
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "arn:aws:iam::027394069461:root"
        ]
      },
      "Action": "kms:*",
      "Resource": "*"
```

6. Click **Previous** to go back and make any changes, if necessary. Click **Finish** when you are done checking the Key policy you just created.
7. From the AWS IAM Management Console, click **Encryption keys** in the left navigation bar again and open the list of encryption keys. Be sure to select the same region from the drop down that you chose when creating the key to see the correct list.
8. Find the key that you just created. Select and copy the key ID from the list. Repeat the process for the other keys.

NOTE
To separate the encryption keys for data, configuration, and log files, we recommend that you create three separate encryption keys. Give each type of key a descriptive name (for example, `ML_data_key`) for the type of content it will be used to encrypt.

9. Enter the following information to identify the external KMS and the required encryption keys. Add the appropriate encryption key ID to each field.

NOTE
We recommend that you create three separate encryption key IDs (one for data, one for configuration, and one for logs). Give each a descriptive name in order to help distinguish between them.

Setting	Description
`host name`	The host name of the external Key Management Server (KMS).
`port`	The external KMS client socket port number.
`external data encryption key id`	The UUID that identifies the encryption key from the external KMS that is to be used to encrypt data files.
`external config encryption key id`	The UUID that identifies the encryption key from the external KMS that is to be used to encrypt config files.
`external logs encryption key id`	The UUID that identifies the encryption key from the external KMS that is to be used to encrypt log files.

For more about IAM roles and privileges, see Creating an IAM Role in the *MarkLogic Server on Amazon Web Services (AWS) Guide*. To learn more about using MarkLogic Server with Amazon Web Services, see the *MarkLogic Server on Amazon Web Services (AWS) Guide*.

Encryption on EBS Volumes

Elastic Block Storage Volume is a durable, block-level storage device that you can attach to a single EC2 instance. Encryption on EBS offers a simple encryption solution for your EBS volumes without the need to build, maintain, and secure your own key management infrastructure. AWS EBS volumes support encryption with a custom key.

Starting in MarkLogic Server 9.0-8, this capability is supported by MarkLogic Server for AWS. Users can turn on encryption on EBS volumes on their cluster and also optionally specify a custom key for volumes. This can be done using MarkLogic Server CloudFormation templates and Managed Cluster Feature. See The Managed Cluster Feature and Deploying MarkLogic on EC2 Using CloudFormation in the *MarkLogic Server on Amazon Web Services (AWS) Guide*.

If a cluster is created by the MarkLogic Server CloudFormation template, a same encryption key will be used to encrypt all EBS volumes in the cluster. If encryption option is specified, all volumes attached to an instance will apply the same setting. EBS Encryption is only supported by some EC2 instance types, mostly the new generation. The key that is used to encrypt the volume must be in the same region.

NOTE

KMS keys are never transmitted outside of the AWS regions in which they were created.

Enhanced AWS S3 Encryption Support

Starting with MarkLogic Server 9.0-8, Amazon AWS S3 support with encryption is built into MarkLogic Server as an available file system or a storage location for backup/restore. When MarkLogic Server writes or updates objects on AWS S3, it can use the AWS KMS server-side encryption to protect data. You can choose the encryption method by GUI or API.

To use the AWS KMS key to encrypt data that will be stored on AWS S3, specify which key to be used to encrypt. You can do this using the Admin Interface or by using the `admin:group-set-s3-server-side-encryption-kms-key` API. To find the S3 encryption key (if it has already been set) use the `admin:group-set-s3-server-side-encryption-kms-key` API.

To set the AWS KMS in the MarkLogic Server Admin Interface, navigate to Groups Configuration page. Scroll down to the S3 protocol configuration field. Select `https` as the s3 protocol and `aws:kms` as the s3 server-side encryption. Paste the s3 server-side encryption kms key into the field.

Configure the external KMS keys as shown in the previous section.

14.9.3. Using MarkLogic Server Encryption with Microsoft Azure Key Vault

Microsoft Azure Key Vault can encrypt your data in MarkLogic Server. Azure Key Vault is supported for customers running their cluster on Microsoft Azure. You must set up your Azure Key Vault, create the encryption keys in Key Vault, and configure the encryption key IDs in your MarkLogic Server before using the keys to encrypt data in MarkLogic Server.

To set up the Microsoft Azure Key Vault, first set up your Azure instance. See Getting Started with MarkLogic Server on Azure and Overview of MarkLogic Server on Azure for details. Keys are governed by access policies created by the Key Administrator. See the next section (Microsoft Azure Key Vault) for details and the Azure documentation regarding key policies for more information.

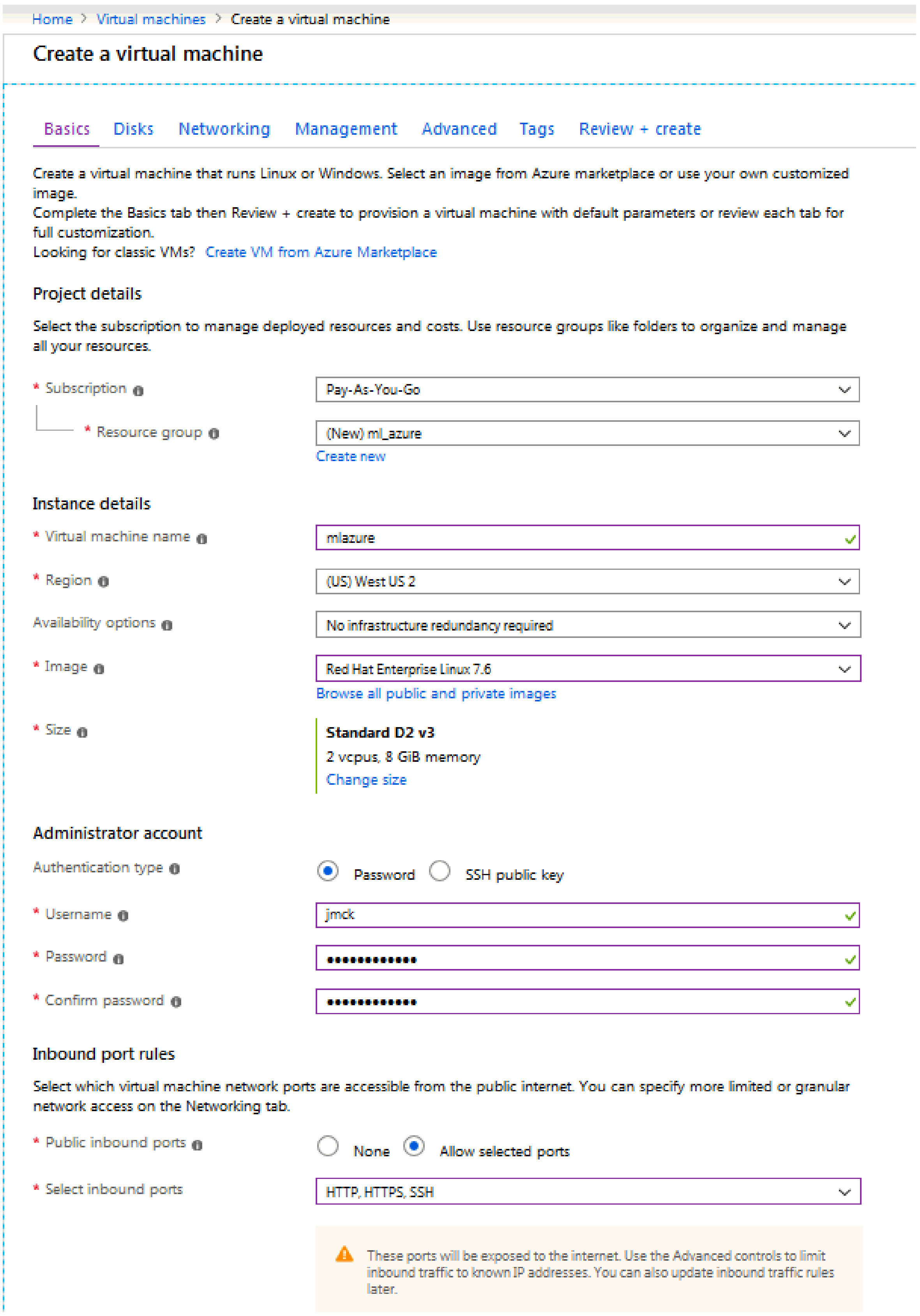
WARNING
If an encryption key stored in the Azure Key Vault is disabled, it cannot be used for encryption or decryption and MarkLogic Server loses access to any data encrypted with the disabled key. Deleting a key will lead to permanent data loss as deleted keys can never be recovered.

Microsoft Azure Key Vault

To set up Microsoft Azure Key Vault, you will create a virtual machine (VM) on Azure. Then create a Key Vault, set up your access policy, and create your encryption keys in the Key Vault.

Create a Virtual Machine in Azure

1. On the Azure Home page, click **Virtual machines**.
2. Click **Add** to create a new virtual machine (VM). This page appears:


Home > Virtual machines > Create a virtual machine

Create a virtual machine

Create a virtual machine that runs Linux or Windows. Select an image from Azure marketplace or use your own customized image.
Complete the Basics tab then Review + create to provision a virtual machine with default parameters or review each tab for full customization.
Looking for classic VMs? Create VM from Azure Marketplace

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription	Pay-As-You-Go
* Resource group	(New) ml_azure Create new

Instance details

* Virtual machine name	mlazure
* Region	(US) West US 2
Availability options	No infrastructure redundancy required
* Image	Red Hat Enterprise Linux 7.6 Browse all public and private images
* Size	Standard D2 v3 2 vcpus, 8 GiB memory Change size

Administrator account

Authentication type	[✓ Password] SSH public key
* Username	jmck
* Password	•••••••••••••
* Confirm password	•••••••••••••

Inbound port rules

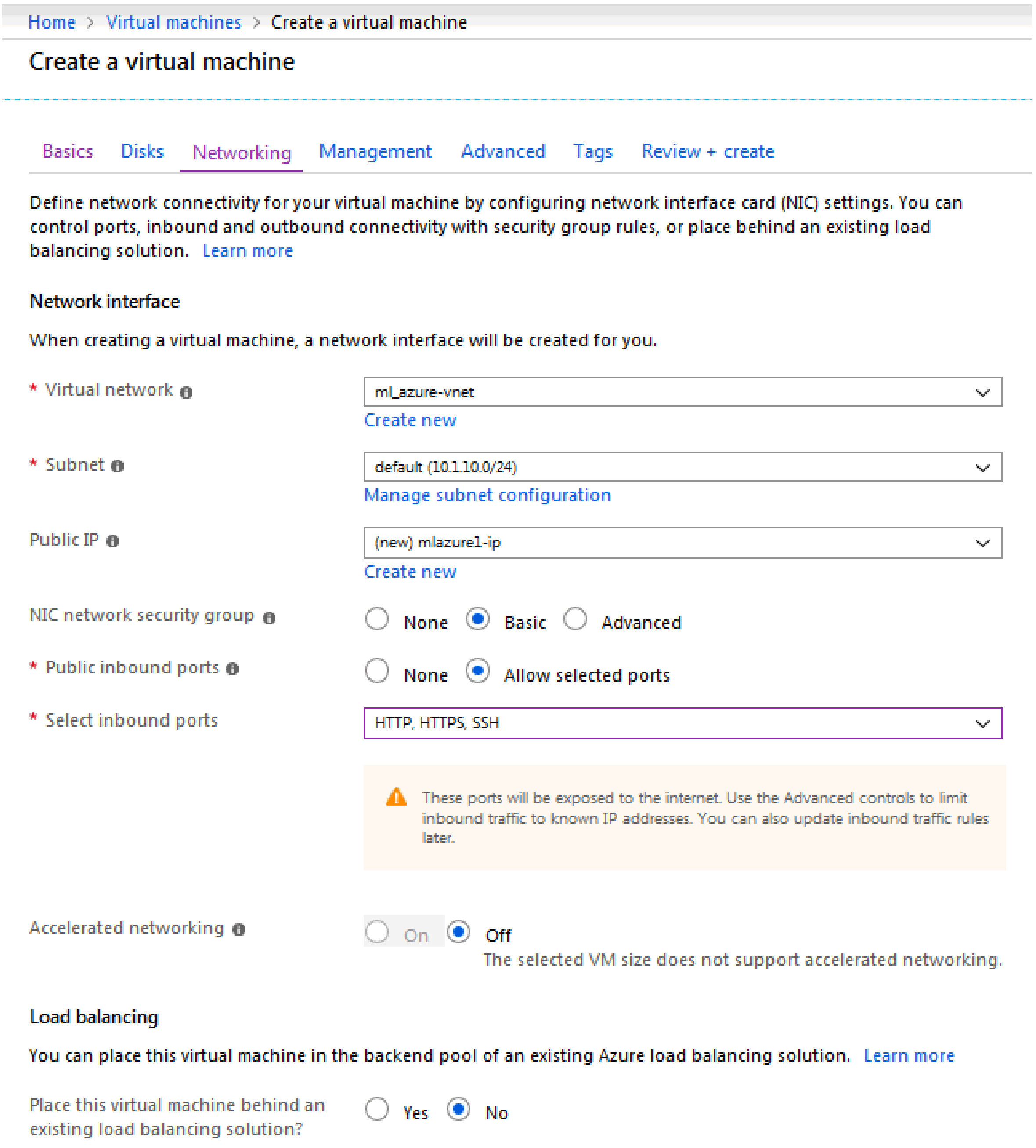
Select which virtual machine network ports are accessible from the public internet. You can specify more limited or granular network access on the Networking tab.

* Public inbound ports	None [✓ Allow selected ports]
* Select inbound ports	HTTP, HTTPS, SSH

These ports will be exposed to the internet. Use the Advanced controls to limit inbound traffic to known IP addresses. You can also update inbound traffic rules later.

3. Enter information into the fields for the basic setup:
   a. In Resource group, select or create a resource.
   b. In Virtual machine name, enter a name for the new virtual machine.
   c. In Region, select a region to host the virtual machine (for example, `West US 2`).

   d. In Image, select an image type (for example, `Red Hat`).
   e. In Authentication type, choose either password (and fill in the username and password fields) or SSH public key.
4. Click the **Networking** tab. This page appears:



Home > Virtual machines > Create a virtual machine

Create a virtual machine

Basics Disks Networking Management Advanced Tags Review + create

Define network connectivity for your virtual machine by configuring network interface card (NIC) settings. You can control ports, inbound and outbound connectivity with security group rules, or place behind an existing load balancing solution. Learn more

Network interface

When creating a virtual machine, a network interface will be created for you.

* Virtual network	ml_azure-vnet Create new
* Subnet	default (10.1.10.0/24) Manage subnet configuration
Public IP	(new) mlazure1-ip Create new
NIC network security group	None [✓ Basic] Advanced
* Public inbound ports	None [✓ Allow selected ports]
* Select inbound ports	HTTP, HTTPS, SSH

These ports will be exposed to the internet. Use the Advanced controls to limit inbound traffic to known IP addresses. You can also update inbound traffic rules later.

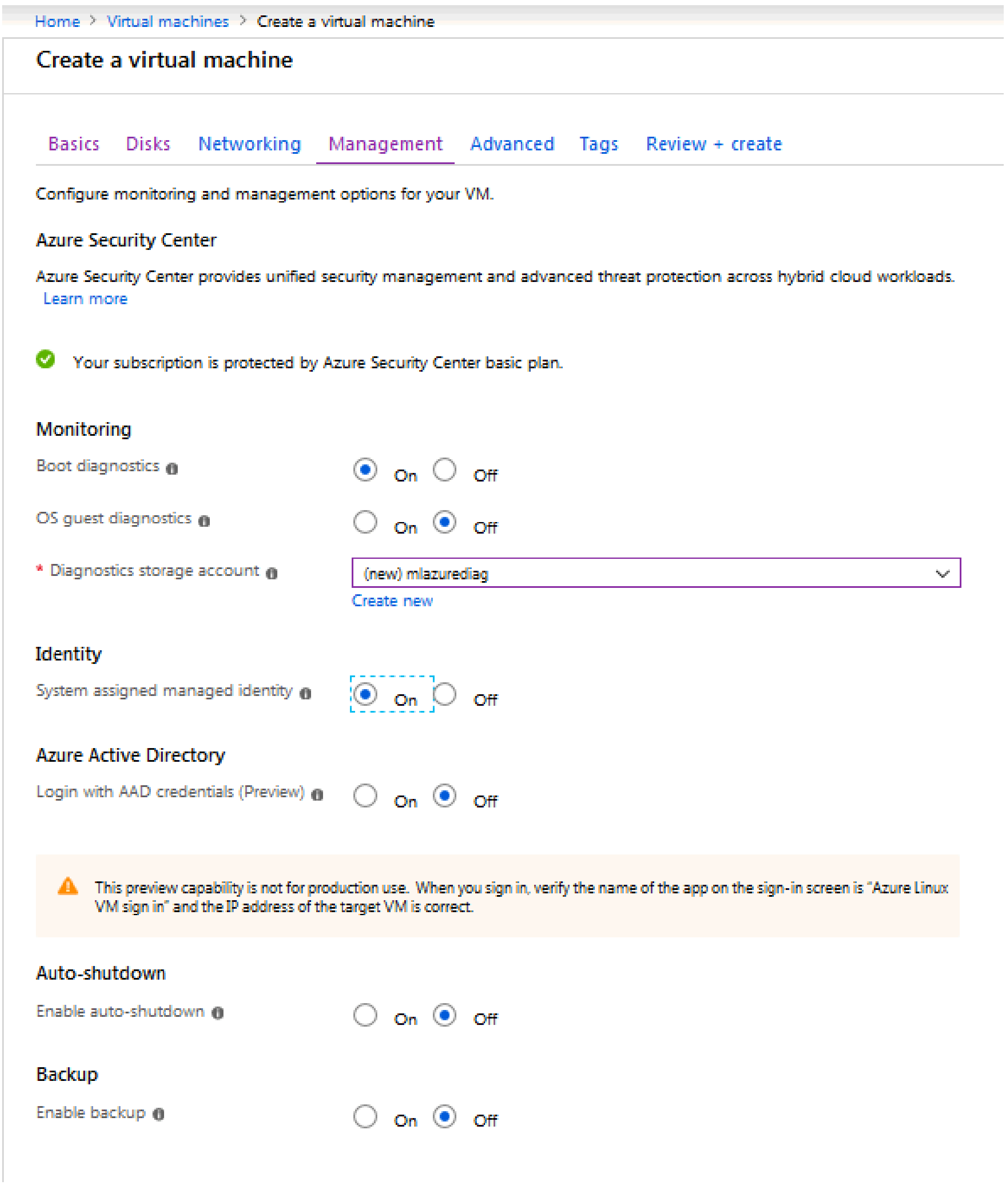
Accelerated networking	On [✓ Off] The selected VM size does not support accelerated networking.

Load balancing

You can place this virtual machine in the backend pool of an existing Azure load balancing solution. Learn more

Place this virtual machine behind an existing load balancing solution?	Yes [✓ No]

   a. In NIC network security group, select Basic.
   b. In Select inbound ports, select (80, 443, 22).
5. Under the Management tab, set System assigned managed identity to On.


Home > Virtual machines > Create a virtual machine

Create a virtual machine

Configure monitoring and management options for your VM.

Azure Security Center

Azure Security Center provides unified security management and advanced threat protection across hybrid cloud workloads. Learn more

Your subscription is protected by Azure Security Center basic plan.

Monitoring

Boot diagnostics: [✓ On] Off

OS guest diagnostics: On [✓ Off]

* Diagnostics storage account: (new) mlazurediag

Create new

Identity

System assigned managed identity: [✓ On] Off

Azure Active Directory

Login with AAD credentials (Preview): On [✓ Off]

This preview capability is not for production use. When you sign in, verify the name of the app on the sign-in screen is "Azure Linux VM sign in" and the IP address of the target VM is correct.

Auto-shutdown

Enable auto-shutdown: On [✓ Off]

Backup

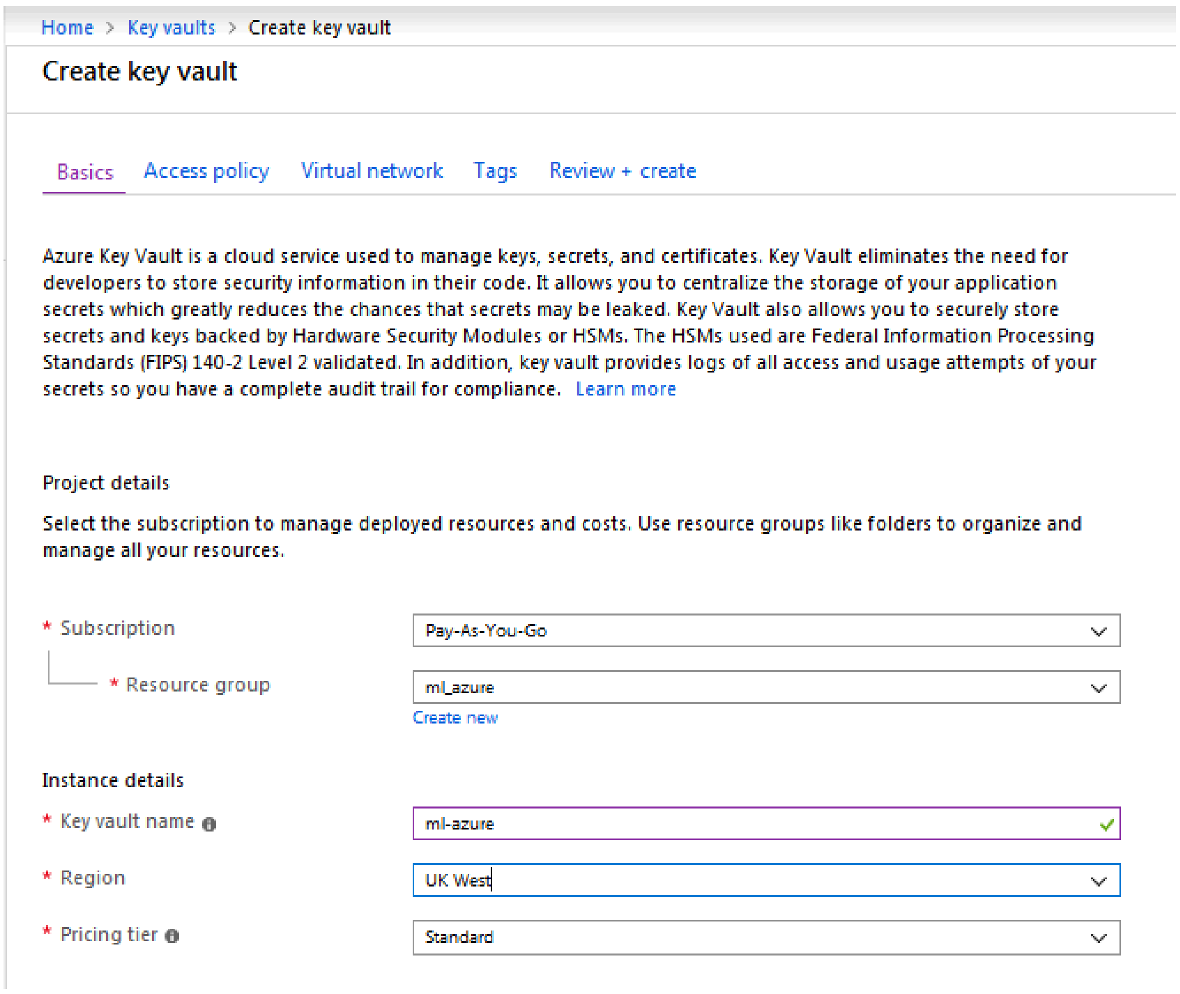
Enable backup: On [✓ Off]

6. Under the Review tab, enter your preferred email address and phone number.
7. Review your information and click **Create**. The create process may take a bit of time.
8. Once the virtual machine has been created, configure the Key Vault.

Configure Azure Key Vault

To create an Azure Key Vault, follow these steps:

1. Navigate to Key Vaults under Home (use Search to find Key Vaults).



Home > Key vaults > Create key vault

Create key vault

Azure Key Vault is a cloud service used to manage keys, secrets, and certificates. Key Vault eliminates the need for developers to store security information in their code. It allows you to centralize the storage of your application secrets which greatly reduces the chances that secrets may be leaked. Key Vault also allows you to securely store secrets and keys backed by Hardware Security Modules or HSMs. The HSMs used are Federal Information Processing Standards (FIPS) 140-2 Level 2 validated. In addition, key vault provides logs of all access and usage attempts of your secrets so you have a complete audit trail for compliance. Learn more

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription	Pay-As-You-Go
* Resource group	ml_azure
	Create new

Instance details

* Key vault name	ml-azure
* Region	UK West
* Pricing tier	Standard

2. Create a new Key Vault with name/resource group/location and a new access policy with keys permissions (decrypt and encrypt) and principle (your newly created VM).
3. Under Settings navigate to Keys, and generate new keys for data/config/logs encryption. Use these keys IDs to configure MarkLogic Server encryption.

Install MarkLogic Server

Install MarkLogic Server on the Azure virtual machine. See Set Up a Simple Deployment in the *MarkLogic Server on Microsoft® Azure® Guide* for details. Once MarkLogic Server is installed on Azure, start MarkLogic Server and navigate to the Admin Interface (port 8001).

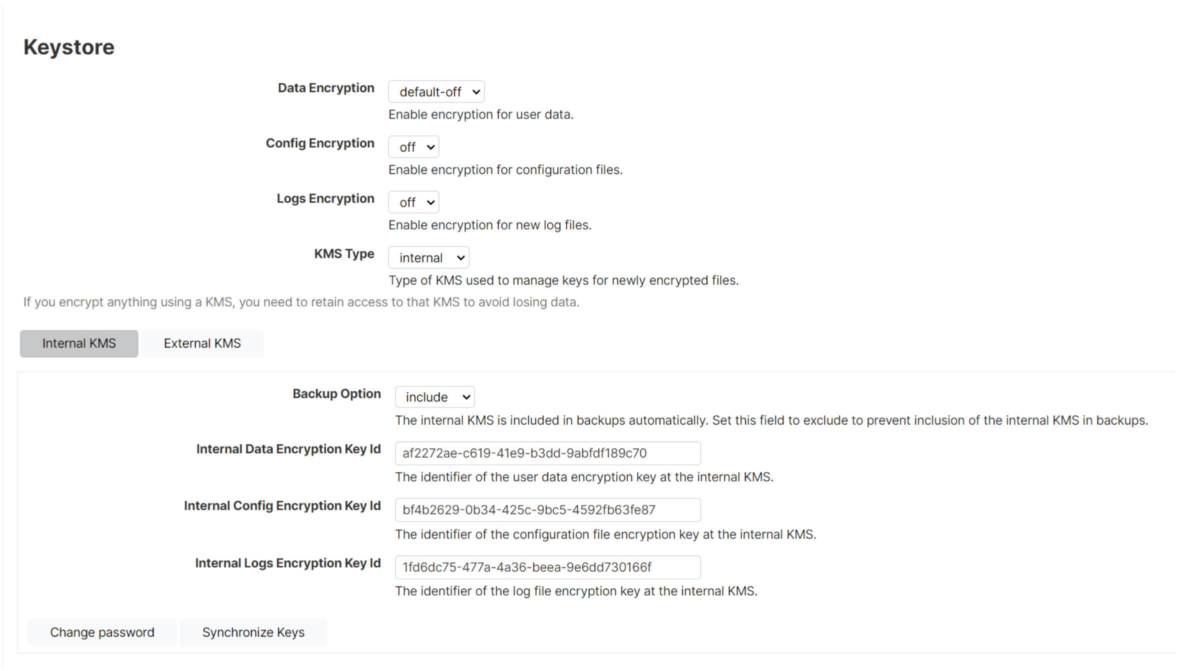
NOTE

You may need to stop the firewall from the command line (`sudo service firewalld stop`).

Add Encryption Configuration Settings to MarkLogic Server

To add encryption configuration settings to MarkLogic Server, follow these steps in the MarkLogic Server Admin Interface:

1. Click **Clusters** in the left navigation bar.
2. Click the **Keystore** tab. This screen appears:


Keystore

Data Encryption: default-off
Enable encryption for user data.

Config Encryption: off
Enable encryption for configuration files.

Logs Encryption: off
Enable encryption for new log files.

KMS Type: internal
Type of KMS used to manage keys for newly encrypted files.

If you encrypt anything using a KMS, you need to retain access to that KMS to avoid losing data.

Internal KMS | External KMS

Backup Option: include
The internal KMS is included in backups automatically. Set this field to exclude to prevent inclusion of the internal KMS in backups.

Internal Data Encryption Key Id: af2272ae-c619-41e9-b3dd-9abfdf189c70
The identifier of the user data encryption key at the internal KMS.

Internal Config Encryption Key Id: bf4b2629-0b34-425c-9bc5-4592fb63fe87
The identifier of the configuration file encryption key at the internal KMS.

Internal Logs Encryption Key Id: 1fd6dc75-477a-4a36-beea-9e6dd730166f
The identifier of the log file encryption key at the internal KMS.

3. In kms type, select `external`.
4. Click the **External KMS** tab.
5. Enter the following information to identify the Azure Key Vault and the required encryption key identifiers, adding the appropriate encryption key ID to each field:
   - Set host name using DNS Name from the Azure Key Vault (without the beginning `https://` and the ending `/`, and ending with `vault.azure.net`).
   - Set port to 443.
   - Copy the encryption key IDs for the Azure Key Vault into the external data encryption key id, external config encryption key id, and external logs encryption key id fields.
6. Click **OK** to configure encryption.

NOTE
We recommend that you create three separate encryption key IDs (one for data, one for configuration, and one for logs). Give each a descriptive name in order to help distinguish between them.

Setting	Description
host name	The host name of the external Key Vault.
port	The external Key Vault client socket port number.
external data encryption key id	The identifier of the encryption key from the external KMS that is to be used to encrypt data files.
external config encryption key id	The identifier of the encryption key from the external KMS that is to be used to encrypt config files.
external logs encryption key id	The identifier of the encryption key from the external KMS that is to be used to encrypt log files.

For more about roles and privileges, see the *MarkLogic Server on Microsoft® Azure® Guide*.

14.9.4. Set Up an External KMIP KMS with MarkLogic Server Encryption

To configure the external key management system using the MarkLogic Server Admin Interface on the MarkLogic Server host, you will need the following information for your external KMS:

- Host name - the hostname of the key management system
- Port number - the port number used to communicate with KMS
- Data encryption key ID (UUID generated by external KMS)
- Configuration encryption key ID (UUID generated by external KMS)
- Logs encryption key ID (UUID generated by external KMS)

The TLS certificates, used to secure the communication with the KMS, must be stored locally on each host in the MarkLogic Server data directory (`/var/opt/MarkLogic`). By default, the files are expected to be located in the MarkLogic Server data directory and must have the following names:

- `kmip-CA.pem` - The root/certificate of the CA that signed the certificate request for MarkLogic Server.
- `kmip-cert.pem` - The certificate that was issued to MarkLogic Server and the one that was signed by the CA.
- `kmip-key.pem` - The private key that was generated for MarkLogic Server and is associated with the Certificate issued to MarkLogic (kmip-cert). (Optional for some KMS servers.)

These certificates are the Certificate Authority (CA) for the root of the certificate chain for the `kmip-cert.pem`. A certificate could be a self-signed root used by an enterprise or an external CA. Copy these files into the MarkLogic Server data directory (`/var/opt/MarkLogic`). The location and name of these files can be changed by calling the admin functions. See Admin APIs for Encryption at Rest for details.

NOTE

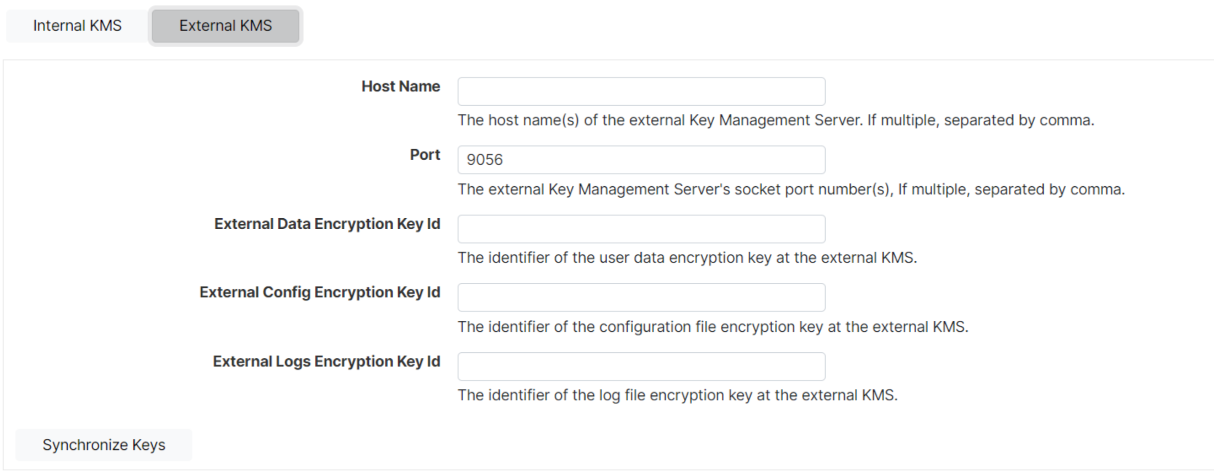
These settings are cluster wide, so each individual host must have a local copy at the location specified.

14.9.5. High Availability and Failover with External KMS

Encryption at rest enables you to specify multiple hosts, multiple ports, and multiple KMIP credentials to connect to KMIP servers. The information to connect to these servers are specified in the fields on the external Key Management Service (KMS) section of the **Edit Keystore Configuration** page. The information must be validated at configuration time. For each host with specified, if there must exist a PEM-encoded Certificate Authority file and a PEM-encoded KMIP certificate file accessible to each node of the MarkLogic Server.

For each host specified, there must exist a PEM-encoded Certificate Authority file and PEM-encoded KMIP certificate file accessible to each node of MarkLogic Server.

The PEM files are looked up with the user-specified path or default location for the first host. For subsequent hosts, the file names are expected to be accessible through the original file name pre-pended by the host’s index in the configuration sequence.



Host Name	The host name(s) of the external Key Management Server. If multiple, separated by comma.
Port	9056 — The external Key Management Server's socket port number(s), If multiple, separated by comma.
External Data Encryption Key Id	The identifier of the user data encryption key at the external KMS.
External Config Encryption Key Id	The identifier of the configuration file encryption key at the external KMS.
External Logs Encryption Key Id	The identifier of the log file encryption key at the external KMS.

For example, if the configured host names are “kms1.marklogic.com” and “kms2.marklogic.com”. The configured port is 9010. The specified CA file is at “path/CA.pem”. The specified certificate file is at “/path/cert.pem”. The configuration must be validated through the following:

1. File `/path/CA.pem`, `/path/1-CA.pem`, `/path/cert.pem`, and `/path/1-cert.pem` all exist.
2. The user-specified encryption keys can be validated through connecting to `kms1.marklogic.com` at port `9010`.
3. The user-specified encryption keys can be validated through connecting to `kms2.marklogic.com` at port `9010`.

If the first specified KMIP host stops responding, the program will try to connect to each of the other hosts on the user-specified list in turn until it successfully connects.

If for some reason the program is unable to connect with a valid KMIP server after multiple attempts, it will report an exception.

14.10. Set Up the External KMS

In most cases, an external KMS is configured by security administrator, a separate role from the MarkLogic Server admin role. However, in some cases the security administrator may also be the MarkLogic admin role.

If you don’t already have the external KMS configured and running, set up the external KMS using the appliance’s interface before turning on MarkLogic Server encryption. The steps in the process for setting up the external KMS will depend on the type of KMIP-compliant external KMS you are using.

Make sure that:

- The external key management system is set up, running, and provisioned first to use KMIP 1.2, before you configure MarkLogic Server encryption.
- To secure communications between the KMS and MarkLogic Server obtain the required certificates; KMIP TLS certificate, CA of the KMS, private key for the client (optional for some KMS servers).

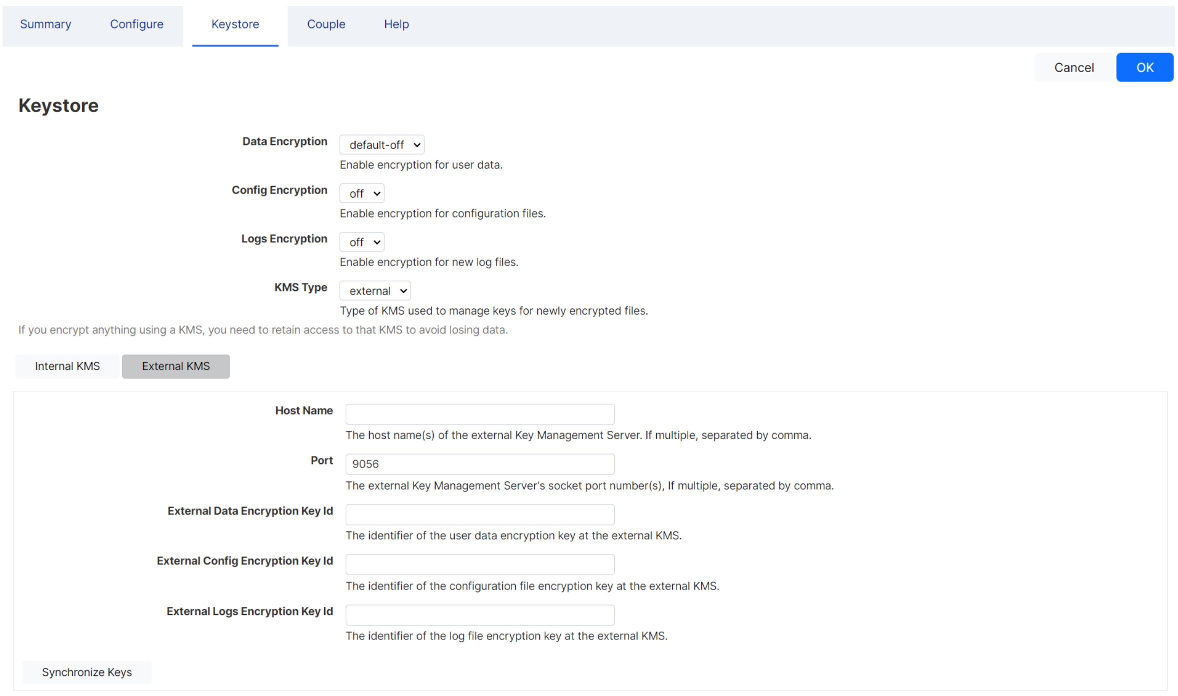
The security administrator can enable encryption for user data, configuration files, and/or logs, either per cluster or per database. You must use the administration tools that come with the external KMS to set up the external keystore.

NOTE
The external key management system (KMS) must be available during the MarkLogic Server startup process. Access to the external KMS must be granted to all nodes in the cluster.

14.10.1. Set Up MarkLogic Server Encryption

Before you set up encryption at rest, be sure that your cluster has upgraded to MarkLogic Server 9. If the cluster has not been upgraded, the encryption feature will not be available.

1. Set up your external KMS, if not already set up. See Set Up an External KMIP KMS with MarkLogic Server Encryption for details.
2. Get the generated encryption key IDs from the external KMS (for data, config, and logs as needed). If you are using data encryption, configuration file encryption, and log encryption, and you want different encryption keys for each, you will need three encryption key IDs (UUIDs).
3. Click **Clusters** in the left navigation tree, then click the name of the cluster to configure.
4. Click the **Keystore** tab, then click the external radio button next to Key Management System (KMS). Additional fields for setting up the external KMS are displayed.
5. Provide the host name and port number for your external KMS in the appropriate fields.



Keystore

Data Encryption	default-off — Enable encryption for user data.
Config Encryption	off — Enable encryption for configuration files.
Logs Encryption	off — Enable encryption for new log files.
KMS Type	external — Type of KMS used to manage keys for newly encrypted files.

If you encrypt anything using a KMS, you need to retain access to that KMS to avoid losing data.

Host Name	The host name(s) of the external Key Management Server. If multiple, separated by comma.
Port	9056 — The external Key Management Server's socket port number(s), If multiple, separated by comma.
External Data Encryption Key Id	The identifier of the user data encryption key at the external KMS.
External Config Encryption Key Id	The identifier of the configuration file encryption key at the external KMS.
External Logs Encryption Key Id	The identifier of the log file encryption key at the external KMS.

NOTE
Replace the existing host name and port and any existing encryption key IDs, with the information for the external KMS.

6. Add the encryption key IDs (generated by the external KMS) for the types of encryption you are configuring (data, configuration, and/or logs), to the appropriate fields on the Edit Keystore Configuration page in the Admin Interface.
7. Click **OK**.
8. Turn on the types of encryption you wish from Admin Interface (data encryption, configuration file encryption, and/or log file encryption).

NOTE
Adding the encryption information will require a restart of all of the hosts in your cluster.

When using an external KMS, key encryption keys (KEK) might be rotated according to the policy set in the KMS. Each time that the keys are rotated in an external KMS, you will have to update the new KEK IDs (UUIDs - like key encryption keys - KEKs) to MarkLogic Server. Data will then start to be encrypted with new KEK ID, as described in Key Rotation. The object keys (OKEYs) with be enveloped by the external KMS and the new keys as MarkLogic Server uses the IDs to request that the OKEY be enveloped with the corresponding KEK ID.

Encryption at rest may be configured using REST, XQuery, or JavaScript APIs. See APIs for Encryption at Rest for details.

14.10.2. Transitioning from PKCS #11 Secured Wallet to an External KMS

Transitioning from the internal PKCS #11 secured wallet to an external KMS will re-encrypt of all configuration files and forest labels. Re-encryption will happen the next time a file is written to disk. If you want to force re-encryption of all data, start a re-index of the database.

Customer-provided cluster KEK IDs will be validated against the KMS for encryption/decryption. If any KEK ID validation fails or MarkLogic Server cannot connect to the KMS, there will be no changes to the configuration files.

Even after you have migrated to an external KMS, the PKCS #11 secured wallet will retain and manage any encryption keys that were generated before the migration to the external keystore.

To migrate from the PKCS #11 secured wallet to an external keystore (KMS) do the following:

1. **Important**: Before you start the transition to an external KMS, back up the wallet that contains all of the internal keys.
2. Confirm that the external KMS is running and available. See Set Up an External KMIP KMS with MarkLogic Server Encryption .
3. Enable the desired encryption options from the MarkLogic Server Admin Interface. MarkLogic Server encryption will now use the encryption keys supplied by the external KMS.

14.10.3. Transitioning from an External KMS to PKCS #11 Secured Wallet

WARNING
Moving from an external KMS to the internal KMS will downgrade your overall security, as the external KMS is more secure than the internal PKCS #11 secured wallet.

If for some reason you want to stop using your external KMS and revert to using the internal PKCS #11 secured wallet, use the steps in this section to transition to the internal PKCS #11 wallet.

To migrate encryption to internal the PKCS #11 wallet, do the following:

1. **Important**: Before you start the transition to an external KMS, back up the wallet that contains all of the internal keys.
2. Turn off encryption on all categories and force decryption of all encrypted forests by issuing a merge command.

3. Ensure that all data is un-encrypted, forest status reports encryption size.
4. Set the configuration back to the internal PKCS #11 KMS and rotate the key encryption keys. See Key Rotation for more information.
5. Re-index or force a merge of the database to re-encrypt your data.

> **NOTE**
> Encrypted read-only forests will need to be set to `updates-allow all and merge` or they will be inaccessible.

14.11. Administration and Maintenance

This section covers additional tasks you may want to perform once you have configured encryption.

14.11.1. Backup and Restore

Individual backup files are encrypted with the cluster data encryption key (CDKEK). Backups are forest driven, so data from an encrypted forest will also be encrypted in backups. Configuration files included in a backup will be encrypted if the cluster is enabled for configuration file encryption. This encryption works with full backups, incremental backups, and journal archiving.

> **NOTE**
> If any forest in the backup has encryption enabled, then the entire backup will be encrypted.

The encryption keys residing in the PKCS #11 secured wallet (the embedded KMS) will be exported as part of a full backup by default. This is true whether encryption is configured to use the internal KMS or an external KMS. Full backups will include this exported copy of the keystore, encrypted using the embedded KMS passphrase, unless you specify otherwise. See Excluding the Embedded KMS from a Backup.

> **WARNING**
> If you cannot access your PKCS #11 secured wallet (or external KMS if you are using one), or lose your encryption keys, you will not be able to decrypt any of your encrypted data (including backups). There is no workaround to recover the encrypted data. We recommend that you backup your encryption keys in a secure location.

The built-in function `admin:cluster-set-keystore-passphrase()` can be used to change the KMS passphrase. When you first set up encryption, we strongly recommend that you change the KMS passphrase to something other than the admin passphrase. This is to ensure that you utilize the Separation of Duties security principle as much as possible.

NOTE
By default, the keystore passphrase is automatically set to the admin password. We strongly recommend that you set a new, different passphrase before turning on encryption.

During an internal keystore backup/restore, data is added to the embedded PKCS #11 secured wallet; no keys are deleted. The encrypted file containing the keys is named `kms.exp`. The exported keystore is not imported during a restore from a backup. If you need to restore the keys, use the `xdmp:keystore-import()` function. The keystore passphrase will be required to decrypt the exported keystore file when restoring backups on another MarkLogic Server instance.

NOTE
To change the keystore passphrase, the current password or passphrase is required.

To restore an encrypted backup to the same cluster:

Restore the backup as usual. See Backing Up and Restoring a Database in *Administrating MarkLogic Server* for details.

To restore an encrypted backup to a different cluster:

1. Use the `xdmp:keystore-import()` function to import the keystore. This function requires the keystore passphrase of the cluster where the backup was created, to decrypt the keystore:
   `xdmp:keystore-import("keystore passphrase", "/backup-directory/kms.exp")`
   The import process will reject duplicate keys and log a warning that includes the ID of the rejected keys. Imported keys can only be used for decryption.
2. Restore the backup as usual. See Backing Up and Restoring a Database in *Administrating MarkLogic Server* for details.

   NOTE
   As long as the current database being restored is encrypted, the restored database will also be encrypted.

Using this process, you can move your encrypted backups from one system to another and restore them, as long as you have the passphrase and import the keystore into the new system before restoring the backup. See Backup and Restore Overview in *Administrating MarkLogic Server* for more information about backup and restore procedures.

WARNING
If you lose the cluster configuration information, you must first manually restore the keystore before an encrypted backup can be restored.

To export your keystore, use the `xdmp:keystore-export()` function: `xdmp:keystore-export("strong passphrase", "/backups/MarkLogic.wallet.bak")`

This function exports all of the encryption keys stored in the MarkLogic Server-embedded KMS (the PKCS #11 secured wallet) and stores them at the location provided to the function.

Excluding the Embedded KMS from a Backup

WARNING
If you set the backup option to `exclude` and turn off the automatic inclusion of the keystore, you are responsible for saving keystore (the embedded KMS) to a secure location. If you cannot access your PKCS #11 secured wallet (or external KMS if you are using one), or lose your encryption keys, *you will not be able to decrypt any of your encrypted data* (including backups).

By default, the MarkLogic Server-embedded KMS (the PKCS #11 secured wallet) is automatically included in a backup. You can exclude the embedded wallet using the options in `admin:cluster-set-keystore-backup-option`. The `include` or `exclude` options enable you to choose whether to have the embedded KMS included as part of backups.

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

let $option := "exclude"
let $config := admin:get-configuration()
return
  admin:cluster-set-keystore-backup-option($config,$option)
```

Setting the option to `exclude` prevents the embedded KMS from being included in the backup.

Backups Using a Secondary Key

MarkLogic Server encryption at rest includes the ability to use a secondary backup key encryption key (BDKEK) for encrypting backups when encryption is configured with an external KMS. Using this BDKEK you can restore your backup to a new system, one that might not have access to the CDKEK and/or CCKEK.

For example, with this XQuery statement you can back up your Documents database using the BDKEK:

```
xdmp:database-backup(xdmp:database-forests(xdmp:database("Documents")),"/backups/Data",
fn:true(),
"/backups/JournalArchiving", 15,"bf44aab-3f7a-41d2-a6a5-fc41a0e5e0cf")
```

Or you could use server-side JavaScript:

```
xdmp.databaseBackup(xdmp.databaseForests(xdmp.database("Documents")), "/backups/Data",
fn:true(),
"/backups/JournalArchiving", 15,"bf44aab-3f7a-41d2-a6a5-fc41a0e5e0cf");
```

In these examples “`bf44aab-3f7a-41d2-a6a5-fc41a0e5e0cf`” is the secondary backup key (BDKEK).

The built-ins `xdmp:database-backup` and `xdmp:database-incremental-backup` have an optional argument to take advantage of the BDKEK from the external KMS. The REST API can also take advantage of a secondary backup key as part of the backup operations.

Backups Using a Passphrase

MarkLogic Server also provides the ability to encrypt backups with a backup passphrase. The `xdmp:dababase-backup` and `xdmp:database-incremental-backup` APIs take an optional argument for the passphrase (`$backup-passphrase`).

Similarly, the built-in `xdmp:database-restore` for restoring a database accepts an optional parameter for the backup passphrase (`$backup-passphrase`). Using a passphrase, a user can restore into any system without requiring import of the original keys or connection to an external KMS.

14.11.2. Tool to View Encrypted Log Files Outside of MarkLogic Server

MarkLogic Server encryption at rest includes the `mlecat` command line tool, which can be used to view encrypted log files outside of the server.

NOTE
Windows users use `mlecat.bat` instead of `mlecat`.

The `mlecat` tool can be used successfully in either of these conditions:

- If the `mlecat` tool is given access to the MarkLogic Server data directory and the `.pem` files.
- If the log files are encrypted with a user-specified logs passphrase and the same logs passphrase is passed to `mlecat` with `-p` option.

NOTE
The `mlecat` tool should be run by a user with sufficient OS privileges to access the PKCS#11 wallet (located by default at `/var/opt/MarkLogic`). It is suggested that the user be a member of group running MarkLogic Server (by default `daemon`).

If you want to decrypt log files without having access to your KMS, you must set a `logs-encryption-passphrase`. To set this passphrase, use the `admin:cluster-set-keystore-logs-encryption-passphrase()` function. 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()
let $passphrase := "dazzling zebras"
let $config := admin:cluster-set-keystore-logs-encryption-passphrase
   ($config,$passphrase)
return admin:save-configuration($config)
```

NOTE
Log file encryption must be enabled for this passphrase to be used.

For every OS you must add `MARKLOGIC_INSTALL_DIR` and `MARKLOGIC_INSTALL_DIR/bin` to your `PATH`. For example,

```
PATH=$MARKLOGIC_INSTALL_DIR:$MARKLOGIC_INSTALL_DIR/bin:$PATH
```

For more about setting environment variables on various platforms, see the information about installation and data directories as part of Installing MarkLogic in *Installing MarkLogic Server*.

To see the command line options for the `mlecat` tool, invoke `mlecat` with no arguments:

```
mlecat
==>
mlecat [option] filepath(s)
option:
  -i iDIR, iDir is the MarkLogic Server Install directory, alternatively the environmental
variable
MARKLOGIC_INSTALL_DIR can be used to set this value.
  -d dDIR, dDIR is the MarkLogic Server Data directory, alternatively the environmental
variable
MARKLOGIC_DATA_DIR can be used to set this value
  -p PASS, PASS is your logs-encryption-passphrase (if you are using one);
  [-f] filepath(s), one or more file paths (-f can be specified before each file for
explicit file list)
```

For example:

```
mlecat -p admin /var/opt/MarkLogic/Logs/ErrorLog.txt
```

Defaults for the MarkLogic Server data and install directories are shown in this table:

Platform	Installation Directory	Default Data Directory (for configuration and log files)
Windows	`c:\Program Files\MarkLogic`	`c:\Program Files\MarkLogic\Data`
Red Hat Linux	`/opt/MarkLogic`	`/var/opt/MarkLogic`
Mac OS X	`~/Library/MarkLogic`	`~/Library/Application Support/MarkLogic/Data`

For more about setting environment variables on various platforms, see the information about installation and data directories as part of Installing MarkLogic in *Installing MarkLogic Server*.

14.11.3. Disaster Recovery/Shared Disk Failover

Unless you have suffered a complete loss of your host, disaster recovery should work just fine with encryption at rest. See High Availability and Disaster Recovery in the *Concepts Guide* for information about setting up shared disk failover and steps for disaster recovery.

If you have experienced a complete loss of your host, follow these steps:

1. Reinstall and configure a new MarkLogic Server host.
2. Import the keystore and keys from a backup (using `xdmp:keystore-import`). See Export and Import Encryption Keys for details.
3. Perform a restore from backup as usual. See Backing Up and Restoring a Database in *Administrating MarkLogic Server* for more information.

14.12. APIs for Encryption at Rest

The encryption at rest feature includes APIs for working with encryption, using either the default keystore (the internal PKCS #11 secured wallet) or a KMIP-compliant external KMS.

14.12.1. Built-ins for Encryption at Rest

These functions will work with both the internal PKCS #11 secured wallet, or an external KMIP-compliant keystore. Using these functions, you can encrypt data and check the status of encryption in your clusters using either JavaScript or XQuery.

These are the Server-Side JavaScript built-ins:

- `xdmp.keystoreExport`
- `xdmp.keystoreImport`
- `xdmp.filesystemFileEncryptionStatus`
- `xdmp.databaseEncryptionAtRest`
- `xdmp.databaseEncryptionKeyId`
- `xdmp.keystoreValidateExported`

These are the Server-Side XQuery built-ins:

- `xdmp:keystore-export`
- `xdmp:keystore-import`
- `xdmp:filesystem-file-encryption-status`
- `xdmp:database-encryption-at-rest`
- `xdmp:database-encryption-key-id`
- `xdmp:keystore-validate-exported`

Using a Credential ID with http-options

The `xdmp:http-options` function now accepts a credential-id when used with XQuery. The schema looks like this:

```
<xs:complexType name="options">
    <xs:sequence>
      <xs:element ref="timeout" minOccurs="0"/>
      <xs:element ref="data" minOccurs="0"/>
      <xs:element ref="headers" minOccurs="0"/>
      <xs:element ref="credential-id" minOccurs="0"/>
      <xs:element ref="authentication" minOccurs="0"/>
      <xs:element ref="client-cert" minOccurs="0"/>
      <xs:element ref="client-key" minOccurs="0"/>
      <xs:element ref="pass-phrase" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

14.12.2. Admin APIs for Encryption at Rest

These functions are used to set the mode and descriptions for the host, set the keystore host name and the keystore host port. You can also set the keystore data key ID, config key ID, or logs key ID, along with setting the keystore serve certificate and enabling encryption.

These server-side XQuery functions work with either the PKCS #11 secured wallet or a third-party KMIP-compliant keystore:

- `admin:cluster-get-config-encryption`
- `admin:cluster-get-data-encryption`
- `admin:cluster-get-logs-encryption`
- `admin:cluster-set-config-encryption`
- `admin:cluster-set-data-encryption`
- `admin:cluster-set-logs-encryption`
- `admin:database-get-data-encryption`
- `admin:database-set-data-encryption`
- `admin:cluster-set-keystore-passphrase`

- `admin:cluster-set-keystore-logs-encryption-passphrase`
- `admin:cluster-get-keystore-backup-option`
- `admin:cluster-get-keystore-wallet-location`

The `admin:cluster-rotate-`*`xxxx`*`-encryption-key-id` APIs are only for use with the embedded KMS provided by MarkLogic Server (the PKCS #11 secured wallet). Using these functions with an external KMS will cause an error.

- `admin:cluster-rotate-config-encryption-key-id`
- `admin:cluster-rotate-data-encryption-key-id`
- `admin:cluster-rotate-logs-encryption-key-id`
- `admin:group-get-rotate-audit-files`
- `admin:group-get-rotate-log-files`
- `admin:group-set-rotate-audit-files`
- `admin:group-set-rotate-log-files`

These next two APIs are used in transitioning from an internal keystore (the PKCS #11 secured wallet) to an external KMIP-compliant keystore. If these functions are set to external, MarkLogic Server will first look for the external keystore to verify the keys.

- `admin:cluster-set-keystore-kms-type`
- `admin:cluster-get-keystore-kms-type`

These functions are designed to work with an external KMIP-compliant keystore:

- `admin:cluster-get-config-encryption-key-id`
- `admin:cluster-set-config-encryption-key-id`
- `admin:cluster-get-data-encryption-key-id`
- `admin:cluster-set-data-encryption-key-id`
- `admin:cluster-get-keystore-host-name`
- `admin:cluster-set-keystore-host-name`
- `admin:cluster-get-keystore-port`
- `admin:cluster-set-keystore-port`
- `admin:cluster-get-logs-encryption-key-id`
- `admin:cluster-set-logs-encryption-key-id`
- `admin:cluster-get-keystore-kmip-CA-path`
- `admin:cluster-set-keystore-kmip-CA-path`
- `admin:cluster-get-keystore-kmip-certificate-path`
- `admin:cluster-set-keystore-kmip-certificate-path`
- `admin:cluster-get-keystore-kmip-key-path`
- `admin:cluster-set-keystore-kmip-key-path`
- `admin:database-get-encryption-key-id`
- `admin:database-set-encryption-key-id`

NOTE

The functions designed to work with an external KMS will return an error if you try to use them with the PKCS #11 secured wallet (the default built-in KMS).

14.12.3. REST Management APIs for Encryption

You can manage encryption using the REST Management APIs. Some of the tasks you can do with these APIs include:

- Encryption configuration
- Keystore configuration
- Database configuration
- Database status, including database encryption (encrypted size, total size)
- Cluster status
- Forest status
- Security
- Backups, status (encrypted or not)
- Restore (with property for using private key)

The REST Management APIs that are used to query and manage the cluster security properties include encryption information for database, cluster, and forest.

Below is an XML payload example for the security endpoint:

```
<security-properties xmlns="http://marklogic.com/manage/security/properties"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://marklogic.com/manage/security/properties
manage-security-properties.xsd">
<keystore>
<data-encryption>default-off</data-encryption>
<config-encryption>off</config-encryption>
<logs-encryption>off</logs-encryption>
<kms-type>internal</kms-type>
<host-name>localhost</host-name>
<port>9056</port>
<data-encryption-key-id>92ed7360-458a-427e-abad-c6595b192cb7</data-encryption-key-id>
<config-encryption-key-id>8b9a9bdb-7b0e-41eb-9aa6-ed6e8cb23ad5</config-encryption-key-id>
<logs-encryption-key-id>01c50d02-b43f-46bc-bbe5-6d4111d1180b</logs-encryption-key-id>
</keystore>
</security-properties>
```

And here is a JSON payload example for the security endpoint:

```
{
  "keystore": {
    "data-encryption": "default-off",
    "config-encryption": "off",
    "logs-encryption": "off",
    "kms-type": "internal",
    "host-name": "localhost",
    "port": 9056,
    "data-encryption-key-id":
      "92ed7360-458a-427e-abad-c6595b192cb7",
    "config-encryption-key-id":
      "8b9a9bdb-7b0e-41eb-9aa6-ed6e8cb23ad5",
    "logs-encryption-key-id":
      "01c50d02-b43f-46bc-bbe5-6d4111d1180b"
  }
}
```

These operations are available for encryption key rotation:

```
curl -v -X POST --anyauth --user admin:admin \
  --header "Content-Type:application/json" -d \
  '{"operation":"rotate-config-encryption-key"}' \
  http://localhost:8002/manage/v2/security
```

```
curl -v -X POST --anyauth --user admin:admin \
  --header "Content-Type:application/json" -d \
  '{"operation":"rotate-data-encryption-key"}' \
  http://localhost:8002/manage/v2/security
```

```
curl -v -X POST --anyauth --user admin:admin \
  --header "Content-Type:application/json" -d \
  '{"operation":"rotate-logs-encryption-key"}' \
  http://localhost:8002/manage/v2/security
```

14.13. Interactions with Other MarkLogic Server Features

In most cases the encryption at rest feature will be transparent to the user, that is data on disk will be encrypted, decrypted during use (by users with the appropriate security permissions), and re-encrypted when the data is written back to disk.

14.13.1. Rolling Upgrades

Encryption at rest is a feature introduced in MarkLogic Server 9. Clusters running older versions need to be completely upgraded to MarkLogic Server 9 before using this feature. See Rolling Upgrades in *Administrating MarkLogic Server* for more about rolling upgrades.

NOTE

During upgrades, the default passphrase for the upgraded system is not set. You will need to reset the default passphrase after an upgrade.

14.13.2. Telemetry

The telemetry feature is not available for use until the cluster is upgraded to MarkLogic Server 9.0-1 or later. See Telemetry in *Monitor MarkLogic Server* for more about telemetry.

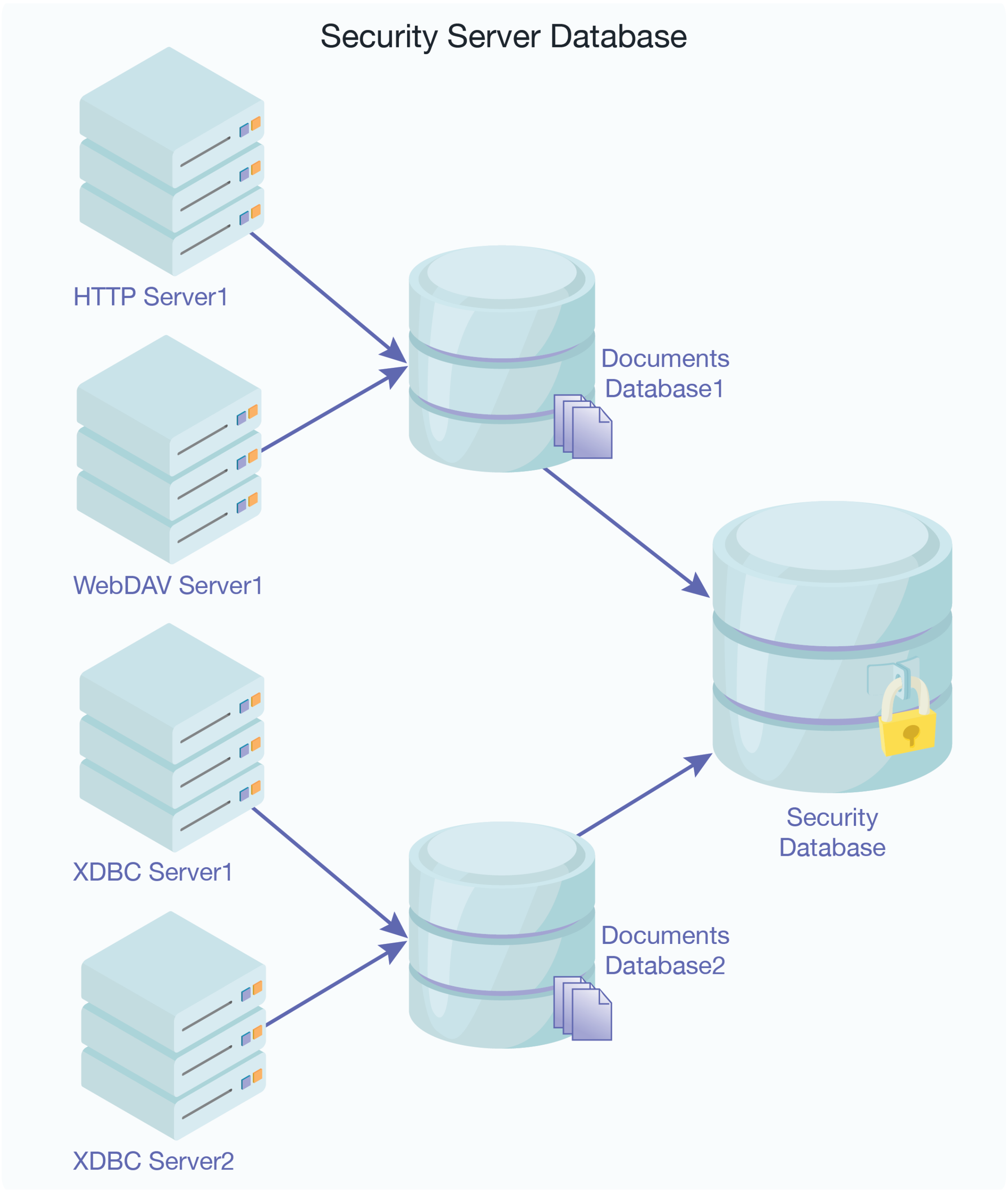
15. Administering Security

This section describes the basic steps 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 Security Administration in *Administrating MarkLogic Server*.

15.1. 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, ODBC, 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 clustered systems as it does for single-node systems; there is always a single security database associated with each HTTP, WebDAV, ODBC, or XDBC server.

The configuration that associates the security database with the database and servers is at the database level. HTTP, WebDAV, ODBC, 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.

In addition to storing users, roles, and privileges that you create, the security database also stores predefined privileges and predefined roles. These objects control access to privileged activities in MarkLogic Server. Examples of privileged activities include loading data and accessing URIs. The security database is initialized during the installation process. For a list of all of the predefined privileges and roles, see Appendix B and Appendix C in *Administrating MarkLogic Server*.

15.2. 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 that specifies the security database.

15.3. Managing and Using Objects in the Security Database

There are two mechanisms available to add, change, delete, and use objects in the security database: the Admin Interface and the XQuery functions. provided by the `security.xqy` library module. This section describes what you can do with each of these mechanisms.

15.3.1. 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 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 *Administrating MarkLogic Server*.

15.3.2. 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` library module includes functions 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 your code.

For the signatures and descriptions of the functions in `security.xqy`, see the *MarkLogic XQuery and XSLT Function Reference*.

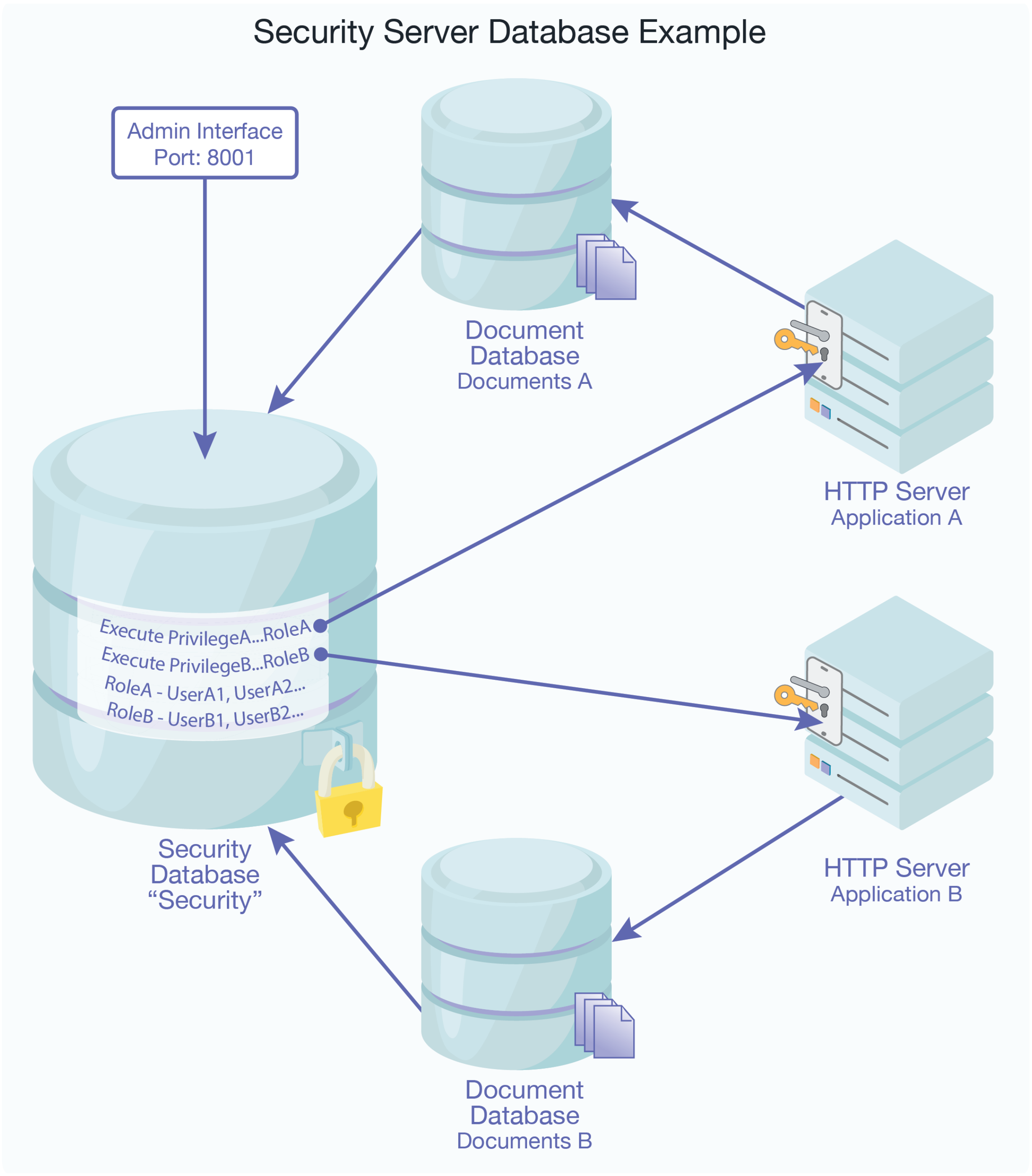
15.4. Backing Up the Security Database

The security database is the central entry point to all of your MarkLogic Server applications. If the security database becomes unavailable, no users can access any applications. Therefore, it is 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 Backing Up and Restoring a Database in *Administrating MarkLogic Server*.

15.5. Example: Using the Security Database in Different Servers

The security database typically is used for the entire system, including all of the HTTP, WebDAV, ODBC, 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 must have the appropriate *login privileges* to log into the server, and therefore they 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 the `Security` database in the diagram simply indicates that the Admin Interface is storing all security objects—users, roles, and privileges—in `Security` database.

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 are created later in this procedure.

   NOTE
   The new execute privileges created using the Admin Interface are stored in the `Security` database. The new roles and users created below are also stored in the `Security` database.

3. Create two new roles. These roles are 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` is 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` is granted `ExecutePrivilegeA` by virtue of its role (`RoleA`) and has login access to `ApplicationA`. Because `ApplicationA` is connected to `DocumentsA`, `UserA1` is 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` have login access to `ApplicationA` and all users assigned `RoleB` 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.

16. Auditing

Auditing is the monitoring and recording of selected operational actions from both application users and administrative users. You can audit various kinds of actions related to document access and updates, configuration changes, administrative actions, code execution, and changes to access control. You can audit both successful and failed activities.

For procedures on setting up auditing as well as a list of audit events, see Auditing Events in *Administrating MarkLogic Server*.

16.1. Why Is Auditing Used?

You typically use auditing to perform the following activities:

- Enable accountability for actions. These might include actions taken on documents, changes to configuration settings, administrative actions, changes to the security database, or system-wide events.
- Deter users or potential intruders from inappropriate actions.
- Investigate suspicious activity.
- Notify an auditor of the actions of an unauthorized user.
- Detect problems with an authorization or access control implementation. For example, you can design audit policies that you expect to never generate an audit record because the data is protected in other ways. However, if these policies generate audit records, then you know the other security controls are not properly implemented.
- Address auditing requirements for regulatory compliance.

16.2. MarkLogic Server Auditing

MarkLogic Server includes an auditing capability. You can enable auditing to capture security-relevant events to monitor suspicious database activity or to satisfy applicable auditing requirements. You can configure the generation of audit events by including or excluding MarkLogic Server roles, users, or documents based on URI. Some actions that can be audited are the following:

- startup and shutdown of MarkLogic Server
- adding or removing roles from a user
- usage of amps
- starting and stopping the auditing system

For the complete list of auditable events and their descriptions, see Auditing Events in *Administrating MarkLogic Server*.

16.3. Configuring Auditing

Auditing is configured at the MarkLogic Server cluster management group level. A MarkLogic Server group is a set of similarly configured hosts in a cluster, and includes configurations for the HTTP, WebDAV, ODBC, and XDBC App Servers in the group. The group auditing configuration includes enabling and disabling auditing for each cluster management group.

Audit records are stored on the local file system of the host on which the event is detected and on which the Server subsystem is running.

Rotation of the audit logs to different files is configurable by various intervals, and the number of audit files to keep is also configurable.

For more details and examples of audit event logs, see Auditing Events in *Administrating MarkLogic Server*.

16.4. Best Practices

Auditing can be an effective method of enforcing strong internal controls enabling your application to meet any applicable regulatory compliance requirements. Appropriate auditing can help you to monitor business operations and detect activities that may deviate from company policy. If it is important to your security policy to monitor this type of activity, then you should consider enabling and configuring auditing on your system.

Be selective with auditing and ensure that it meets your business needs. As a general rule, design your auditing strategy to collect the amount and type of information that you need to meet your requirements, while ensuring a focus on events that cause the greatest security concerns.

If you enable auditing, develop a monitoring mechanism to use the audit event logs. Such a system might periodically archive and purge the audit event logs.

17. Designing Security Policies

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

17.1. Research Your Security Requirements

As a first step in planning 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? Are your applications used only by trusted, internal people or are they open to a wider audience?
- How sensitive is the data you are protecting?

This list is not necessarily comprehensive, but it will get you started thinking about your security policy.

17.2. Plan Roles and Privileges

Depending on your security requirements and the structure of your enterprise or organization, plan the roles and privileges that make the most sense.

1. Determine the level of granularity with which you need to protect objects in the database.
2. 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 `xdmp:security-assert` functions, where needed.
9. Load your documents with the appropriate permissions. If needed, change the permissions of existing documents using the `xdmp:document-add-permissions`, `xdmp:document-set-permissions`, and `xdmp:document-remove-permissions` functions.
10. Assign access privileges to HTTP, WebDAV, ODBC, and XDBC servers as needed.

18. Sample Security Scenarios

This section 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.

18.1. Protecting the Execution of XQuery Modules

One simple way to restrict access to your MarkLogic Server application is to limit the users that have permission to run the application. If you load your Xquery code into a modules database, you can use an execute permission on the XQuery document itself to control who can run it. Then, a user must possess `execute` permissions to run the module. To set up a module to do this, perform the following steps:

1. Using the Admin Interface, specify a modules database in the configuration for the App Server (HTTP or WebDAV) that controls the execution of your XQuery module.
2. Load the XQuery module into the modules database using a URI with an `.xqy` extension like `my_module.xqy`.
3. Set `execute` permissions on the XQuery 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 the URI `http://modules/my_module.xqy`, run a query similar to the following:

   ```
   xdmp:document-set-permissions("http://modules/my_module.xqy",
         xdmp:permission("run_application", "execute") )
   ```
4. Create the `run_application` role.
5. Assign the `run_application` role to the users who can run this application.

Now only users with the `run_application` role can execute this document.

NOTE
Because your application could also contain amped functions, this technique can help restrict access to applications that use amps.

18.2. Choosing the Access Control for an Application

The role-based security model in MarkLogic Server combined with the supported authentication schemes provides numerous options for implementing application access control. This section describes common application access control alternatives.

For details on the different authentication schemes, see Types of Authentication.

18.2.1. Open Access, No Log In

This approach may be 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. This scenario provides all of your users with the `admin` role.

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 you installed MarkLogic Server.

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.

18.2.2. Providing Uniform Access to All Authenticated Users

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.

1. In the Admin Interface, go to the **Users** tab under **Security**.
2. Give all users in the security database the `admin` role.
3. Go to the **Configuration** tab 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 has 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.

18.2.3. Limiting Access to a Subset of Users

This application access control method can be modified or extended to meet the requirements in many application scenarios. 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 named `exe-priv-app1` to represent the privilege to access the App Server.
2. Create a role named `role-app1` that has `exe-priv-app1` execute privilege.
3. Add `role-app1` to the roles of all users in the security database who should have access to this App Server.
4. In the Configuration page for this App 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-app1` for the privilege field. Once this is done, only the users who have the `exe-priv-app1` by virtue of their role(s) are able to access this App 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.

This method of authentication also needs to be accompanied by the appropriate security configuration for both users and documents associated with this App Server. For example, functions such as `xdmp:document-insert` and `xdmp:document-load` throw exceptions unless the user possesses the appropriate execute privileges. Also, users must have the appropriate default permissions (or specify the appropriate permissions with the API) when creating new documents in a database. Documents created by a user who does not have the `admin` role must be created with at least one update permission or else the transaction throws an `XDMP-MUSTHAVEUPDATE` exception. The update permission is required because otherwise once the documents are created no user (except users with the `admin` role) would be able to access them, including the user who created them.

18.2.4. Using 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 App Server, perform the following steps using the Admin Interface:

1. Go to the Configuration tab for the HTTP App 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 is created with the following roles: `rest-reader, rest-extension-user, app-user, harmonized-reader` and is given a password which is randomly generated.
4. Create a custom login page that meets your needs. We 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 `xdmp: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 likely contains a snippet of 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.

NOTE
If you are using a modules database to store your code, the `login.xqy` file still needs to have an `execute` permission that allows the `nobody` (or whichever is the default) user to access the module. For example, you can put an `execute` permission paired with the `app-user` role on the `login.xqy` module document, and make sure the `nobody` user has the `app-user` role (which it does by default).

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 for users and documents. See Introduction to Security for background on the security model.

18.2.5. 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 this access control.

1. Using the Admin Interface, configure the App Server to use a custom login page:
   a. Go to the **Configuration** tab for the HTTP or WebDAV App 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 `nobody` user is automatically created when MarkLogic Server is installed. It is created with the following roles: `rest-reader`, `rest-extension-user`, `app-user`, `harmonized-reader` and is given a password, which is randomly generated.
2. 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 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 App Server.
   b. Add the following code to `login-routine.xqy`:

```
xquery version "1.0-ml"

module namespace widget ="http://widget.com";
declare function widget:try-ip-login(
) as xs:boolean {
  let $ip := xdmp:get-request-client-address()
  return
    if(fn:compare($ip,"127.0.0.1") eq 0) then (:local host:)
      xdmp:login("localuser",())
    else if(fn:starts-with($ip,"<approved-subnet>")) then
      xdmp:login("site-user",())
    else
      fn:false()
};
```

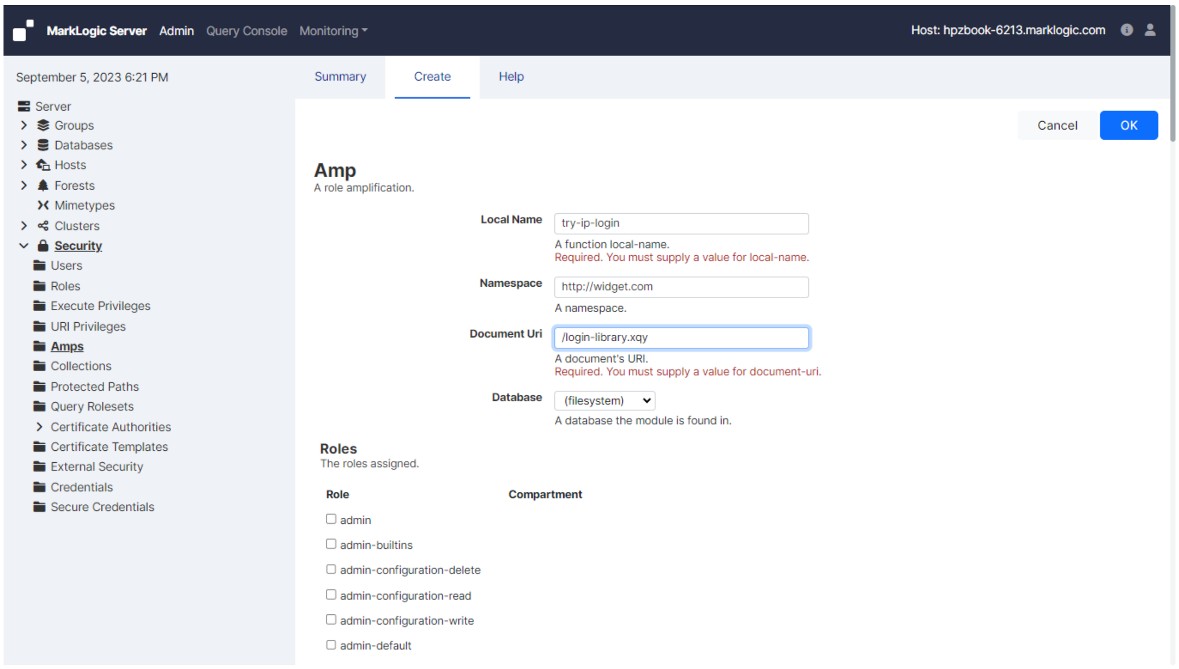
If the user is accessing the application from an approved IP address, `try-ip-login` logs in the user with username `local-user` or `site-user` as appropriate and returns `true`. Otherwise, `try-ip-login` returns `false`.

NOTE
In the code snippet above, the empty sequence () is supplied in place of the actual passwords for `local-user` and `site-user`. The pre-defined `xdmp-login` 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. Add the following code snippet to the beginning of the default page displayed by the application, for example, `default.xqy`.

```
xquery version "1.0-ml";

import module namespace widget = "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")
```

4. Finally, to ensure that the code snippet above is called with the requisite xdmp-login privilege, configure an amp for `try-ip-login`:
   a. Using the Admin Interface, create a role called `login-role`.
   b. Assign the pre-defined `xdmp-login` 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:



Amp
A role amplification.

Local Name: try-ip-login
A function local-name.
Required. You must supply a value for local-name.

Namespace: http://widget.com
A namespace.

Document Uri: /login-library.xqy
A document's URI.
Required. You must supply a value for document-uri.

Database: (filesystem)
A database the module is found in.

Roles
The roles assigned.

Role	Compartment
☐ admin	
☐ admin-builtins	
☐ admin-configuration-delete	
☐ admin-configuration-read	
☐ admin-configuration-write	
☐ admin-default	

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 the `try-ip-login` function is called, the `nobody` user is temporarily amped to the `login-role`. The `nobody` user 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 can access the application with the permissions and privileges of `local-user` or `site-user` as appropriate.

5. 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(fn: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.

18.3. Implementing Security for a Read-Only User

In this scenario, assume that you want to implement a security model that enables your users to run any XQuery code stored in the modules database for a specific App Server with read-only permissions on all documents in the database.

Reviewing the MarkLogic Server security model, recall that users do not have permissions, documents have permissions. And permissions are made up of a role paired with a capability. Additionally, execute privileges protect code execution and URI privileges protect the creation of documents in a specific URI namespace. This example shows one way to implement the read-only user and is divided into two parts.

18.3.1. Steps for Example Setup

To set up this example scenario, perform the following steps, using the Admin Interface:

1. Create a role named `ReadsStuff`.
2. Create a user named `ReadOnly` and grant this user the `ReadsStuff` role.
3. Create a role named `WritesStuff` and grant this role the `ReadsStuff` role.
4. Grant the `WritesStuff` role the `any-uri` privilege, as well as any execute privileges needed for your application code.
5. Create a user named `LoadsStuff` and grant this user the `WritesStuff` role. When you load documents, load them as the `LoadsStuff` user and give each document an update and insert permission for the `WritesStuff` role and a read permission for the `ReadsStuff` role.

Here is sample code to create a set of permissions with `xdmp:permission()`:

```
(xdmp:permission("ReadsStuff", "read"),
xdmp:permission("WritesStuff", "insert"),
xdmp:permission("WritesStuff", "update"))
```

You can also create a set of permissions with the `permissions` option of either `xdmp:document-insert()` or `xdmp:document-load()`.

Also, instead of specifying the permissions when you load documents, you can assign default permissions to the `LoadsStuff` user or the `WritesStuff` role.

18.3.2. Troubleshooting Tips

If you are running a URL rewriter (or an error handler), you need to give the `ReadsStuff` role to the `nobody` user or whichever user is the default user for your App Server. When the URL rewriter executes, the request has not yet been authenticated, so it runs as the default user. The default user is `nobody` unless you have specified a different default for your App Server. The best practice is to create another role, for example `my-app-user` and add an execute permission for the URL rewriter and your error handler (if any) for the `my-app-user` role. This is better because you do not want the `nobody` user to have access to your database.

19. Securing Your Production Deployment

A security system is only as good as its weakest link. This section describes some general principles to think about with an eye toward hardening your entire environment for security.

19.1. Add Password Protections

When your data and business requirements warrant it, design and implement password protections. These protections can range from providing guidelines to your users to implementing programmatic checking to enforce password complexity and management.

Complexity verification verifies that each password is complex enough to provide reasonable protection against intruders who try to break into the system by guessing passwords. This encourages users to create strong passwords.

Password management includes things such as password aging and expiration, automatically locking users out of the application after failed login attempts, and controlling the reuse of old passwords.

To enforce password complexity programmatically, use the password plugins. For more information about the plugin framework and to view a sample password plugin, see System Plugin Framework and Password Plugin Sample in *Developing Server-Side Applications*.

19.2. Adhere to the Principle of Least Privilege

Grant necessary privileges only. Do not provide users or roles more privileges than are necessary. If possible, grant privileges to roles, not individual users. The principle of least privilege is that users are given only those privileges that are actually required to efficiently perform their jobs.

Restrict the following as much as possible:

- The number of users granted the `admin` or `security` roles.
- The number of roles or users who are allowed to make changes to security objects, such as roles, users, and document permissions.
- The number of roles that have capabilities to add, change or remove security-related privileges.

19.3. Infrastructure Hardening

Most computer platforms offer network security features to limit outside access to the system. The purpose of infrastructure hardening is to eliminate as many security risks as possible. It can involve both hardware and software, as well as physical restrictions.

19.3.1. OS-Level Restrictions

The United States National Security Agency develops and distributes security configuration guidance for a wide variety of software, including the most common operating system platforms.

19.3.2. Network Security

Encrypt network traffic between the browser and MarkLogic Server by enabling SSL. You can also enable SSL for intra-cluster communication. For high security needs, make sure MarkLogic Server runs in FIPS mode (which is the default mode). This option restricts your SSL ciphers to those that have met the FIPS 140-2 Level 1 validation requirements. For information on how to configure SSL and FIPS mode, see Clusters in *Administrating MarkLogic Server*.

19.3.3. Port Management

Protect access to the MarkLogic Server Admin Interface and development tool ports:8000, 8001, 8002 behind a corporate firewall. While your MarkLogic Server application may run on a publicly available port, such as port 80, it is good practice to secure the MarkLogic Server Admin Interface and other development application ports behind a firewall.

19.3.4. Physical Access

Ensure that machines running MarkLogic Server are in a physically secure location. Physical access to a server is a high security risk. Physical access to a server by an unauthorized user could result in unauthorized access or modification, as well as installation of hardware or software designed to circumvent security. To maintain a secure environment, you should restrict physical access to your MarkLogic Server host computers.

19.4. Implement Auditing

MarkLogic Server includes an auditing capability. Designing and implementing an auditing policy can be an important part of your overall security planning. For more details, see Auditing in this guide. For procedures related to enabling auditing, see Auditing Events in *Administrating MarkLogic Server*.

19.5. Develop and Enforce Application Security

An important step in creating a MarkLogic Server application is to ensure that it is properly secure. Network security mostly ignores the contents of HTTP traffic, therefore you can’t use network layer protection (firewall, SSL, IDS, hardening) to stop or detect application layer attacks. The Open Web Application Security Project is an open group focused on understanding and improving the security of web applications and web services. You can visit their site at https://owasp.org/. The OWASP Top Ten Project is one starting point for understanding how you can build good security into your application.

19.6. Use MarkLogic Server Security Features

Let collections and document permissions restrict the data access for the user. Do not write your own access restriction code. Write code so that it uses the MarkLogic Server security model and operates on the correct data based on the user’s permissions and the current documents in use.

19.7. Read about Security Issues

Many excellent resources exist on the Internet. These sources contain valuable security-related information for everyone in the enterprise software development and deployment chain from software developers and system administrators to managers. For example, the Defense Information Systems Agency (DISA) site contains technical guidance to “lock down” information systems and software that might otherwise be vulnerable to a malicious computer attack.

Another example is the CERT Program, a part of the Software Engineering Institute, a federally funded research and development center operated by Carnegie Mellon University. This organization is devoted to ensuring that appropriate technology and systems management practices are used to resist attacks on networked systems and to limit damage and ensure continuity of critical services in spite of successful attacks, accidents, or failures.

20. Technical support

Progress Software provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at `http://help.marklogic.com` to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the *Support Handbook* for instructions on registering support contacts and on working with the MarkLogic Server Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at `http://developer.marklogic.com`. For technical questions, we encourage you to ask your question on Stack Overflow.

21. Copyright