

---

# **MarkLogic Server**

---

## **Information Studio Developer's Guide**

Release 4.2  
October, 2010

Last Revised: 4.2-3, February, 2011

## Copyright

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

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

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

Contains LinguistX, from Inxight Software, Inc. Copyright © 1996-2006. All rights reserved. [www.inxight.com](http://www.inxight.com).

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

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

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

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

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

Contains ICU with the following copyright and permission notice:

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

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

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

## Information Studio Developer's Guide

Copyright .....	2
1.0 Introduction to Information Studio .....	5
1.1 Information Studio Components .....	5
1.2 Application Services App Server and Databases .....	6
1.3 Information Studio APIs .....	7
1.4 Configuring Information Studio for Large-Scale Loading Processes .....	7
1.5 Starting Application Services .....	8
2.0 Controlling Access to Information Studio .....	9
2.1 Predefined Roles for Information Studio .....	9
2.1.1 infostudio-user .....	9
2.1.2 infostudio-admin-internal .....	9
2.2 The infostudio-admin User .....	9
3.0 Creating and Configuring Databases .....	10
3.1 Database section of the Application Services Page .....	10
3.2 Creating a New Database .....	11
3.3 Configuring a Database .....	12
3.3.1 Navigating to the Database Settings Page .....	12
3.3.2 Configuring Text Indexes .....	14
3.3.3 Configuring Database Range Indexes .....	15
3.3.3.1 Creating an Attribute Range Index .....	16
3.3.3.2 Creating an Element Range Index .....	17
3.3.4 Configuring Database Fields .....	19
3.4 Deleting a Database .....	20
4.0 Creating and Configuring Flows .....	21
4.1 Information Studio section of the Application Services Page .....	21
4.2 Creating a New Flow .....	22
4.3 Collect .....	23
4.3.1 Selecting a Collector .....	23
4.3.2 Using the Filesystem Directory Collector .....	24
4.3.3 Using the Browser Drop-Box Collector .....	25
4.3.4 Configuring Ingestion Options .....	27
4.4 Transform .....	30
4.4.1 Selecting a Transformer .....	30
4.4.2 Custom XQuery .....	31
4.4.3 Renaming Elements or Attributes .....	32
4.4.4 Deleting Elements or Attributes .....	33
4.4.5 Customize XSLT Stylesheet .....	34
4.4.6 Normalize Dates .....	35

4.4.7	Validate Documents against Schema .....	37
4.5	Load .....	37
4.5.1	Selecting the Database .....	38
4.5.2	Document Settings .....	38
4.5.2.1	Configuring the URI Structure .....	39
4.5.2.2	Configuring Document Access Permissions .....	42
4.5.2.3	Configuring Collections .....	43
4.5.2.4	Configuring Quality Boost .....	44
4.6	Launching Ingestion and Tracking Status .....	45
4.7	Deleting a Flow .....	46
5.0	Scripting Information Studio Tasks .....	47
5.1	The info API .....	47
5.2	Creating a Database .....	47
5.3	Configuring the Database Text Indexes .....	48
5.4	Loading Data into Databases .....	49
5.5	Establishing Ingestion Policies .....	50
5.6	Applying Ingestion Policies .....	55
5.7	Ingestion Policies and Multiple Load Operations .....	56
6.0	Creating Custom Collectors and Transformers .....	59
6.1	The infodev and plugin APIs .....	59
6.2	Information Studio Plugin Framework .....	59
6.3	Creating Custom Collectors .....	61
6.3.1	Types of Collectors .....	61
6.3.2	Collector Capabilities and Function Signatures .....	61
6.3.3	Collector Interaction with Information Studio .....	65
6.3.3.1	One-Shot Collectors .....	65
6.3.3.2	Long-Running Collectors .....	66
6.3.4	Collector Type and MarkLogic Server Restart .....	67
6.3.5	An Example Collector .....	68
6.4	Creating Custom Transformers .....	73
6.4.1	Transformer Capabilities and Function Signatures .....	73
6.4.2	Transformer Interaction with Information Studio .....	75
6.4.3	An Example Transformer .....	76
7.0	Technical Support .....	81

## 1.0 Introduction to Information Studio

The MarkLogic Server Application Services suite includes Information Studio, which is a browser-based Interface and XQuery API that enables you to quickly create MarkLogic Server databases and load them with content. Information Studio simplifies how you load and transform content by enabling you to collect content from different sources, process it with XSLT and built-in transformation logic, and load it into a MarkLogic database. You can customize Information Studio to connect to any data source and create your own solutions for transforming content as it is collected and loaded into the database.

This chapter includes the following sections.

- [Information Studio Components](#)
- [Application Services App Server and Databases](#)
- [Information Studio APIs](#)
- [Configuring Information Studio for Large-Scale Loading Processes](#)
- [Starting Application Services](#)

### 1.1 Information Studio Components

The following are the definitions for the main Information Studio components:

- A *flow* is a content load configuration that determines which documents are to be loaded into which database, as well as how they are to be loaded into the database. A flow consists of a unique collector, transformer, and ingestion policy. Flows are described in “Creating a New Flow” on page 22.
- A *collector* is an Information Studio plugin used to gather the content to be loaded into the database. There are two basic types of collectors: one-shot and long-running. Specific collector implementations gather content in different ways. For example, a one-shot collector may scan and load files from a filesystem directory and then stop, while a long-running collector may “listen” for documents from some source and load them into the database until it is explicitly stopped by a user. The collectors bundled with Information Studio are described in “Collect” on page 23. You can also create your own custom collectors, as described in “Creating Custom Collectors” on page 61.
- A *transformer* is an Information Studio plugin used to modify content as it is loaded into the database. Specific transformer implementations modify the content in different ways. The transformers bundled with Information Studio are described in “Transform” on page 30. You can also create your own custom transformers, as described in “Creating Custom Transformers” on page 73.
- An *ingestion policy* is a unit of XML configuration, in the form of a stored <options> node, that specifies how to load content into a database. An Information Studio database may have multiple named policies, as well as a default policy. Ingestion policies are

described in “Configuring Ingestion Options” on page 27 and “Establishing Ingestion Policies” on page 50.

- A *ticket* is a mechanism for tracking a database load process and recording any errors that may have occurred. A ticket has a unique ID, which can be used to obtain status reports. Tickets persist in a database until they reach their expiration date or are explicitly deleted by a user.

## 1.2 Application Services App Server and Databases

Information Studio makes use of an HTTP App Server at port 8002, named `App-Services`, which stores data in the `App-Services` database described below. In addition, Information Studio internally makes use of a database named `Fab`.

Database	Purpose
App-Services	The <code>App-Services</code> database stores the Information Studio configuration data for the flows, as well as all of the tickets and log messages generated by the load operations. The <code>App-Services</code> database also serves as the triggers database for both the <code>App-Services</code> and <code>Fab</code> databases.
Fab	<p>The <code>Fab</code> database retains the state information related to the document transformation and distribution processes. Documents that generated errors during a load operation are also retained in the <code>Fab</code> database.</p> <p>If there are transformation steps configured for the flow, the collector loads the documents to the <code>Fab</code> database, where they are processed by a CPF pipeline, which transforms the content and then distributes the resulting documents to the destination database.</p>

When a flow is created, the following two scheduled tasks are created to garbage-collect the content in the Information Studio databases:

- A scheduled task for deleting expired documents from the `Fab` database. By default, this task is scheduled to run in 30 days at 11:59 pm. This start time can be configured programmatically by means of the `fab-retention-duration` element in the ingestion policy, as described in “Establishing Ingestion Policies” on page 50. The task logs a message at the "Debug" level when no documents remain to be removed.
- A scheduled task for deleting expired tickets from the `App-Services` database. By default, this task is scheduled to run in 30 days at 11:59 pm. This start time can be configured programmatically by means of the `ticket-retention-duration` element in the ingestion policy, as described in “Establishing Ingestion Policies” on page 50. The task logs a message at the "Debug" level for every ticket it deletes as well as a final message when complete, or a default message indicating that the task has run and no tickets were deleted.

**Note:** See [Upgrade Installation](#) in the *Application Builder Developer's Guide* for details on what to expect when upgrading from MarkLogic Server versions 4.0 or 4.1 to version 4.2.

### 1.3 Information Studio APIs

The `info` and `infodev` APIs allow you to programmatically configure and use Information Studio and to create custom collector and transformer plugins. For reference documentation on each function, see the *MarkLogic XQuery and XSLT Function Reference*.

The purpose of the `info` and `infodev` APIs are shown in the table below.

Module	Purpose
<code>info</code>	<p>The API provided by the <code>info</code> module allows you to script the processes reflected in the Information Studio Interface. These processes include creating, configuring, and deleting databases; loading content; setting policy; getting status information (via <code>info:ticket</code>) and error information (<code>info:ticket-errors</code>), and running a flow configured in the Information Studio Interface.</p> <p>The use of the <code>info</code> API is described in “Scripting Information Studio Tasks” on page 47.</p>
<code>infodev</code>	<p>The API provided by the <code>infodev</code> module is specifically for developers creating custom collector and transformer plugins. The functions in this API provide the hooks into the plugin framework.</p> <p>The use of the <code>infodev</code> API is described in “Creating Custom Collectors and Transformers” on page 59.</p>

### 1.4 Configuring Information Studio for Large-Scale Loading Processes

As described in “Application Services App Server and Databases” on page 6, Information Studio makes use the `App-Services` and `Fab` databases to temporarily store collected documents and to retain configuration and state information for the Information Studio flows.

If you plan to load a large amount of content with an Information Studio flow, consider mounting the `App-Services` and `Fab` databases on a different volume as the destination database.

If you initially configured all of your MarkLogic Server databases on one volume, you can delete the forests for the `App-Services` and `Fab` databases, create new forests on a different volume, and attach the new forests to the `App-Services` and `Fab` databases.

If you want to retain the exiting data in the `App-Services` and `Fab` databases, you can move the forests. The following procedure assumes there is no activity on the forests being moved. If there are any updates to the forests being moved, they might end up in different states. Note that this should not be done on active systems, as there will be a short outage period between detaching the old forest and attaching the new forest.

To move an existing forest to another volume, perform the following steps:

1. Use the forest backup/restore page of the Admin Interface to backup the `App-Services` and `Fab` forests to a directory, as described in [Making Backups of a Forest](#) in the *Administrator's Guide*.
2. On the other volume, create new `App-Services` and `Fab` forests, as described in [Creating a Forest](#) in the *Administrator's Guide*.
3. For the new `App-Services` and `Fab` forests, restore the forests from the backups made in step 1, as described in [Restoring a Forest](#) in the *Administrator's Guide*.
4. Detach your original `App-Services` and `Fab` forests from their respective databases and attach the newly restored public forest to the database, as described in [Attaching and/or Detaching Forests to/from a Database](#) in the *Administrator's Guide*.
5. Delete the original private forests.

## 1.5 Starting Application Services

Information Studio is bundled as part of the Application Services suite of applications. To start Application Services, open the following URL in a browser window:

```
http://localhost:8002
```

If your instance of MarkLogic Server is running on a different host, or if Information Studio is configured on a different port, substitute the appropriate values for host and port.

In order to use Information Studio, you must have the `infostudio-user` role assigned to your login account. To use Application Builder, you must have the `app-builder` role. Users with the `admin` role have access to both applications.



## 2.0 Controlling Access to Information Studio

Information Studio allows you to create and configure databases, as well as load documents into databases. This chapter describes the security roles needed to run Information Studio, and includes the following parts:

- [Predefined Roles for Information Studio](#)
- [The infostudio-admin User](#)

### 2.1 Predefined Roles for Information Studio

There are two predefined roles that are used by Information Studio:

- [infostudio-user](#)
- [infostudio-admin-internal](#)

For details about the MarkLogic Server security model and about configuring users and roles, see *Understanding and Using Security Guide* and [Security Administration](#) in the *Administrator's Guide*.

#### 2.1.1 infostudio-user

The `infostudio-user` role is a minimally privileged role that is needed to use Information Studio. You must grant this role to all users who are allowed to access Information Studio.

The `infostudio-user` role has the following execute privileges:

- `infostudio` (<http://marklogic.com/xdmp/privileges/infostudio>)
- `unprotected-collections`

#### 2.1.2 infostudio-admin-internal

The `infostudio-admin-internal` role is used by Information Studio to amp certain functions that Information Studio performs. You should not explicitly grant the `infostudio-admin-internal` role to any user; it is only for internal use by Information Studio.

### 2.2 The infostudio-admin User

The `infostudio-admin` user is a preconfigured user that handles CPF restart and resumes unfinished Information Studio tasks in the event of an unexpected shutdown and restart of MarkLogic Server. When MarkLogic Server is restarted, long-running collectors resume loading documents in the database, as described in “Collector Type and MarkLogic Server Restart” on page 67. In this situation, the original user that started the collector is unknown, so the purpose of the `infostudio-admin` user is to resume control of the collector.

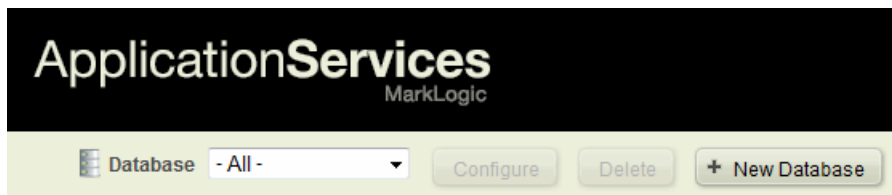
## 3.0 Creating and Configuring Databases

This chapter describes how to create and configure a database in Information Studio. You must have the `infostudio-role` to accomplish these tasks in Information Studio.

- [Database section of the Application Services Page](#)
- [Creating a New Database](#)
- [Configuring a Database](#)
- [Deleting a Database](#)

### 3.1 Database section of the Application Services Page

When you start Application Services, the Application Services page opens. At the top of the Application Services page is the Database section.



The Database section provides you with access to all of the databases in your MarkLogic Server and allows you to create a new database, configure a database, or delete a database. Select a database from the Database pull-down menu and click a button to the right to perform the desired operation.

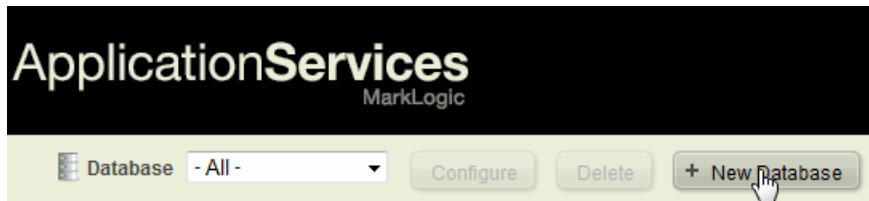
The following table lists the actions on the Database section of the Application Services page:

Action	Description
New Database	Creates a new database, as described in “Creating a New Database” on page 11.
Configure	Configure a database, as described in “Configuring a Database” on page 12.
Delete	Delete a database, as described in “Deleting a Database” on page 20.

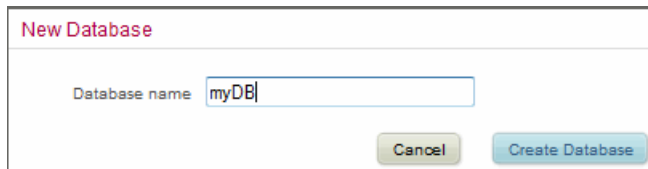
## 3.2 Creating a New Database

This section describes how to create a new database. Creating a new database creates a database and its respective forest.

1. To create a new database, click New Database at the top of the Application Services page:



2. In the New Database window, enter the name of the new database and click Create Database:

A screenshot of the 'New Database' dialog box. It has a title bar that says 'New Database'. Inside, there is a text input field labeled 'Database name' containing the text 'myDB'. At the bottom right of the dialog are two buttons: 'Cancel' and 'Create Database'.

### 3.3 Configuring a Database

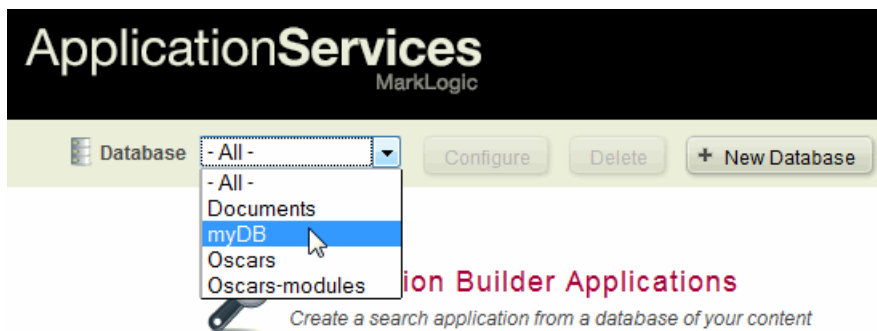
This section describes how to configure a database. The main topics are:

- [Navigating to the Database Settings Page](#)
- [Configuring Text Indexes](#)
- [Configuring Database Range Indexes](#)
- [Configuring Database Fields](#)

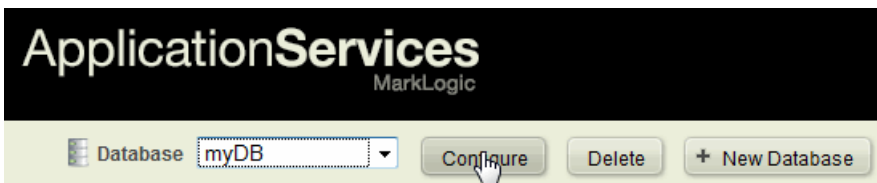
#### 3.3.1 Navigating to the Database Settings Page

To open the Database Settings page, do the following:

1. To configure a database, select the database from the Database pull down window at the top of the Application Services page:



2. After selecting the database, click the Configure button:



3. The Database Settings page appears:

**Database Settings**  
[Application Services](#) > Database settings (myDB)

**Enable Indexes**

☐ Wildcards  
☐ Positions  
☐ Collection Lexicon

**Range Indexes**  
Create range indexes to enable fast searches or sorting on specific elements or attributes.

Name	Type
------	------

[+ Add New](#)

**Fields**  
Create fields for more specific searches.

Name	Root	Include
------	------	---------

[+ Add New](#)

### 3.3.2 Configuring Text Indexes

Information Studio provides a subset of the text indexing options provided by the Admin Interface, which are described in the [Text Indexing](#) chapter in the *Administrator's Guide*. This section describes how to configure some of the more common text indexing options using Information Studio.

The text indexing settings are described below. Each setting is described in detail in the [Understanding the Text Index Settings](#) section in the *Administrator's Guide*. These settings enable more efficient searches of the documents in your database. The cost of more efficient searches is slower loading times of the content into the database, as well as an increased use of system resources, such as memory and disk space.

- Checking Wildcards enables Three Character Searches and Codepoint Word Lexicon indexing. Use this setting for more efficient wildcard searches on the documents in your database.
- Checking Positions enables Word Positions indexing. Use this setting for more efficient phrase searches on the documents in your database.
- Checking both Wildcard and Positions enables Three Character Word Position indexing. Use this setting for better performance when wildcards are used in phrase searches on the documents in your database.
- Checking Collection Lexicon enables Collection Lexicon indexing. Use this setting for more efficient searches that constrain on collections.

#### Database Settings

[Application Services](#) > Database settings (myDB)

#### Enable Indexes

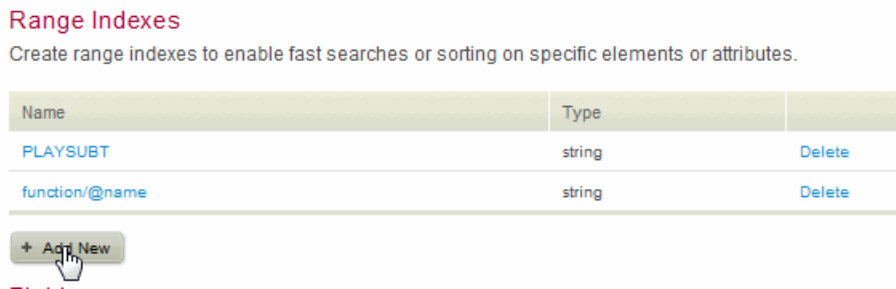
- ☒ Wildcards
- ☒ Positions
- ☐ Collection Lexicon

### 3.3.3 Configuring Database Range Indexes

MarkLogic Server maintains an index for every database to enable search applications to rapidly search the text, structure, and combinations of the text and structure in XML documents. Defining a range index on an element or attribute enables more efficient range query search operations, such as those described in the [Search Page](#) section in the *Application Builder Developer's Guide*. For documents that contain numeric or date information, queries may include search conditions based on inequalities (for example, `price < 100.00` or `date ≥ thisQtr`).

Range indexes are described in detail in the [Element and Attribute Range Indexes and Lexicons](#) chapter in the *Administrator's Guide*. This section describes how to create them using Information Studio.

To create either an attribute range index or an element range index, select Add New at the bottom of the Range Indexes portion of the Database Settings window:



### 3.3.3.1 Creating an Attribute Range Index

This section describes how to create an Attribute Range Index.

1. After clicking Add New, a selection box appears. Click Attribute Index:

**Attribute Index**  
 Create an index for a particular attribute.

**Element Index**  
 Create an index for a particular element.

2. Select the data type of the attribute from the pull-down menu and enter the collation at the top of the page. Enter the name of the attribute to be indexed and click Find. If content similar to that being loaded is already present in the database, all of the elements that contain the attribute appear. Select the element to be indexed from the list.

CONFIGURE ELEMENT ATTRIBUTE RANGE INDEX Close X

Current selection function/@category  
 Data type string  
 Collation http://marklogic.com/collation/

Attribute category Find

Find matching attributes in the source content.

	Element		Attribute		Path
	Name	Namespace	Name	Namespace	
<input type="radio"/>	sum	http://marklogic.com/xdmp/apidoc	category	no namespace	N/A
<input type="radio"/>	module	http://marklogic.com/xdmp/apidoc	category	no namespace	N/A
<input type="radio"/>	summary	http://marklogic.com/xdmp/apidoc	category	no namespace	N/A
<input checked="" type="radio"/>	function	http://marklogic.com/xdmp/apidoc	category	no namespace	N/A
<input type="radio"/>	Other				

Done



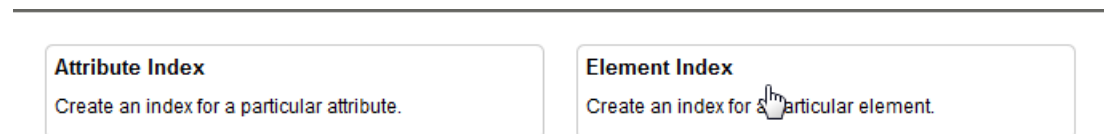
Field	Description
Data Type	The type of attribute data. Each of the types correspond with an XQuery type.
Collation	If the attribute data type is string, then you can specify the URI for the collation to use. Collations specify the order in which strings are sorted and how they are compared. The UCA Root Collation is set by default.
Attribute	The name of the attribute you want to index.

### 3.3.3.2 Creating an Element Range Index

This section describes how to create an Element Range Index. An Element Range Index accelerates queries for comparisons within a specified type. Each Element Range Index keeps track of the values appearing in all elements with a given name, type, and collation. Element Range Indexes also allow you to use the `cts:element-values` family of lexicon APIs and to use the `cts:element-range-query` constructor in searches.

To create an Element Range Index, do the following:

1. After clicking Add New, a selection box appears. Click Element Index:



2. Select the data type of the element from the pull-down menu and enter the collation at the top of the page. Enter the name of the element to be indexed and click Find. If content similar to that being loaded is already present in the database, all of the elements that match the specified name appear. Select the element to be indexed below.

CONFIGURE ELEMENT RANGE INDEX

Close X

Current selection: function

Data type

string

Collation

http://marklogic.com/collation/

---

Element

function

Find

Find matching elements in the source content.

Element	Path
Name      Namespace	
<input checked="" type="radio"/> <div>function</div> <div>http://marklogic.com/xdmp/apidoc</div>	/*:module/*:function[1]
<input type="radio"/> <div>Other</div>	

Done

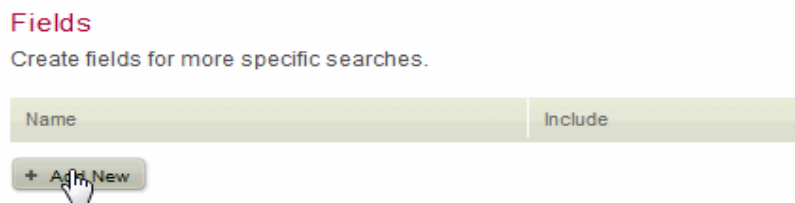
Field	Description
Data Type	The type of element data. Each of the types correspond with an XQuery type.
Collation	If the element data type is string, then you can specify the URI for the collation to use. Collations specify the order in which strings are sorted and how they are compared. TheUCA Root Collation is set by default.
Element	The name of the element you want to index.

### 3.3.4 Configuring Database Fields

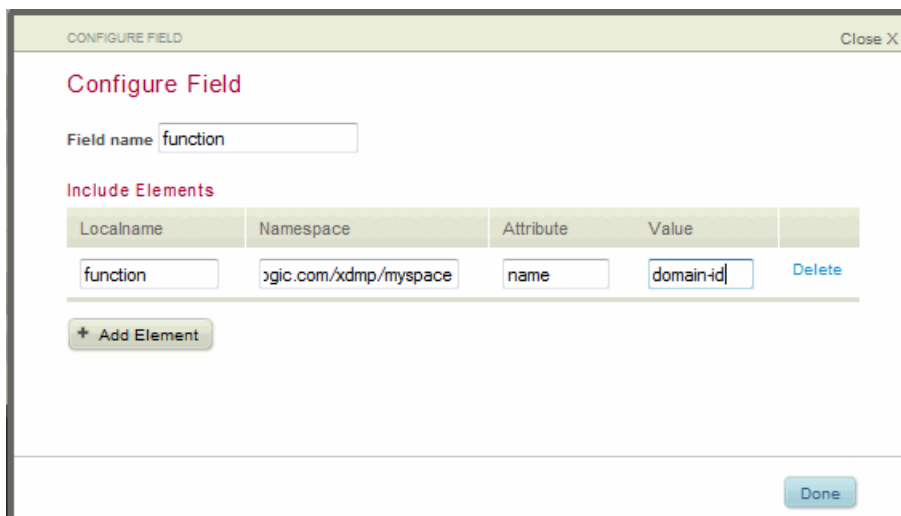
You can group elements into fields and then formulate searches that are constrained to those fields, as described in the [Add/Modify Word Constraint](#) section in the *Application Builder Developer's Guide*. A field can explicitly include elements, depending on their relevance when searching the content. Searches on the included elements in a field can be further constrained by the element's attributes and attribute values.

Fields are described in detail in the [Fields Database Settings](#) chapter in the *Administrator's Guide*. This section describes how to create fields using Information Studio.

1. To create a field, select Add New at the bottom of the Fields portion of the Database Settings window:



2. In the Configure Field page, specify the Field Name and the Localname of the element to be included in the field. You can optionally specify the element's Namespace, Attribute, and attribute Value.



- To include additional elements in the field, click Add New under the Include Elements section and fill in the element information. You can also delete an element from the field by clicking Delete. When finished, click Done.

CONFIGURE FIELD Close X

**Configure Field**

Field name

**Include Elements**

Localname	Namespace	Attribute	Value	
<input type="text" value="function"/>	<input type="text" value="ogic.com/xdmp/myspace"/>	<input type="text" value="name"/>	<input type="text" value="domain-id"/>	<a href="#">Delete</a>
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<a href="#">Delete</a>

[+ Add Element](#)

[Done](#)

- The configured field will appear in the Fields list. To delete the field, click Delete.

**Fields**

Create fields for more specific searches.

Name	Include	
<a href="#">function</a>	<a href="#">function</a>	<a href="#">Delete</a>

[+ Add New](#)

### 3.4 Deleting a Database

To delete a database, select the database from the Database pull-down menu and click Delete. Click OK in the ‘Are you sure you'd like to delete this database?’ popup window. Both the database and its respective forest are deleted.

**Note:** If the database is currently being used by an App Server, the Delete button will be inactive. You must first change the database used by the App Server or remove the App Server before you can delete its database.

**ApplicationServices**  
MarkLogic

Database  [Configure](#) [Delete](#) [+ New Database](#)

## 4.0 Creating and Configuring Flows

This chapter describes how to use Information Studio to create flows that load content into a database.

The main topics are:

- [Information Studio section of the Application Services Page](#)
- [Creating a New Flow](#)
- [Collect](#)
- [Transform](#)
- [Load](#)
- [Launching Ingestion and Tracking Status](#)
- [Deleting a Flow](#)

### 4.1 Information Studio section of the Application Services Page

When you start Application Services, the Application Services page opens. At the bottom of the Application Services page is the Information Studio section.

#### Information Studio Flows

Flow Name	Database	Last Run	Collected / Loaded / Errors	
<a href="#">Load myDB</a>	myDB	03 Jun 2010 12:14:40	84 / 84 / 0	<a href="#">Delete</a>
+ New Flow				

The Information Studio section lists all of the flows in the `App-Services` database and allows you to create a new flow, modify an existing flow, or delete a flow. You can click the flow name to modify an existing flow.

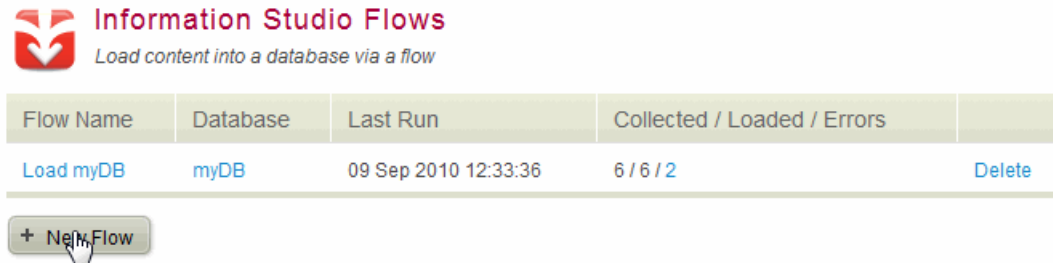
The following table lists the actions on the Information Studio Flows section of the Application Services page:

Action	Description
New Flow	Creates a new flow, as described in “Creating a New Flow” on page 22.
Edit	Click on the flow name to configure an existing flow.
Delete	Permanently delete a flow from the <code>App-Services</code> database, as described in “Deleting a Flow” on page 46.

## 4.2 Creating a New Flow

This section describes how to create a new flow.

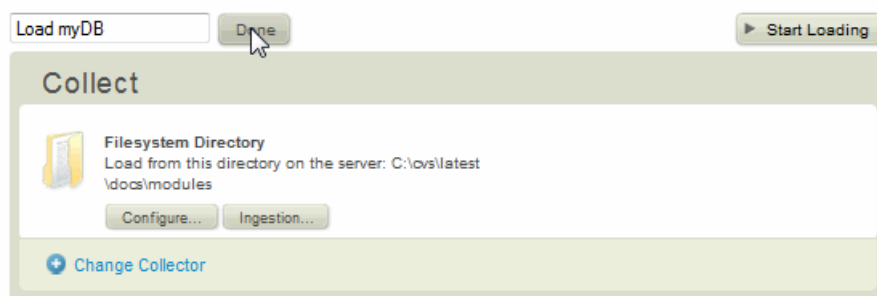
1. To create a new flow, click New Flow in the Information Studio section of the Application Services window:



2. In the Flow Editor window, name the flow by clicking Edit next to the flow name:



3. Enter the name of the flow in the text field and click Done:



## 4.3 Collect

Collectors allow you to specify how the files are to be loaded into the database. The topics in this section are as follows:

- [Selecting a Collector](#)
- [Using the Filesystem Directory Collector](#)
- [Using the Browser Drop-Box Collector](#)
- [Configuring Ingestion Options](#)

### 4.3.1 Selecting a Collector

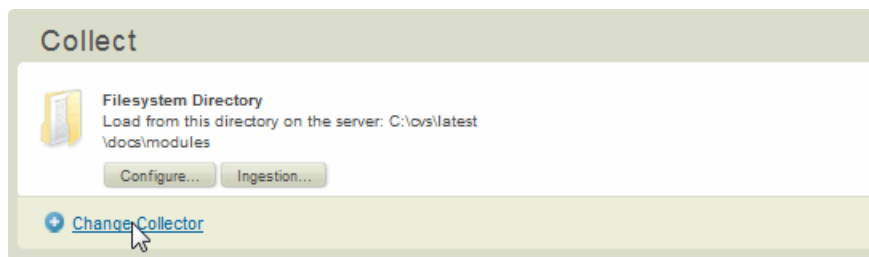
A collector is a plugin component that accumulates content to be loaded into a MarkLogic Server database. Specific collector implementations gather content in different ways. For example, one collector may scan and load files from a filesystem directory in a single pass while another may be configured to monitor and mirror a directory.

Two collectors are shipped with MarkLogic Server:

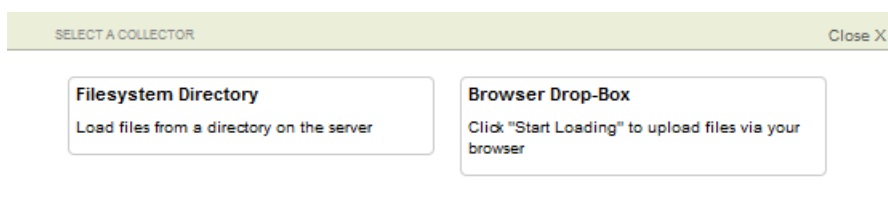
- Filesystem Directory — Load all of the files from a specified directory
- Browser Drop-Box — Load all of the files dropped into a browser window

How to use each of these collectors is described below. You can also create custom collectors, as described in “Creating Custom Collectors” on page 61.

1. By default, the Filesystem Directory collector is set. You can change your collector by clicking Change Collector at the bottom of the Collect section of your flow:



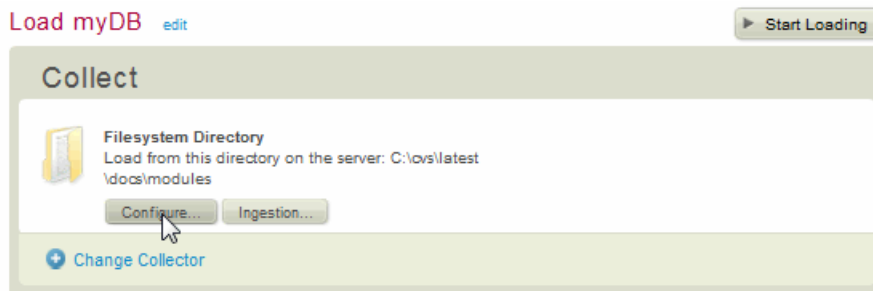
2. Select the type of collector you want to use:



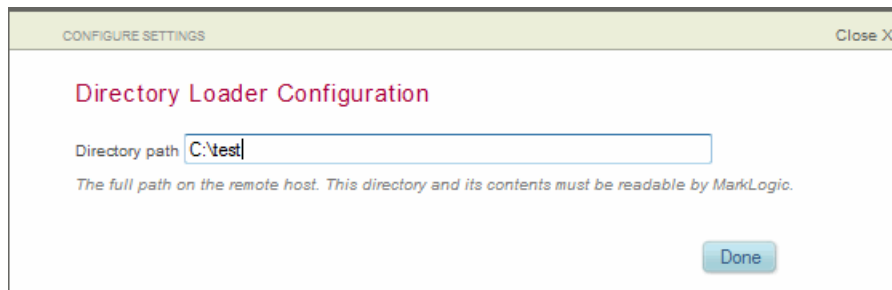
### 4.3.2 Using the Filesystem Directory Collector

The Filesystem Directory collector allows you to load all of the files from a specified directory into the database. The Collector Options specify where the ingested documents are to be found in the filesystem. The Ingestion Options specify the manner in which the documents are to be ingested into the database, as described in “Configuring Ingestion Options” on page 27.

1. To configure the Collector Options, select Configure in the Collect box:



2. Enter a Directory Path to specify the location of the documents in the filesystem to be ingested into the database. All of the files in the specified directory and its subdirectories will be ingested. When finished, click Done:

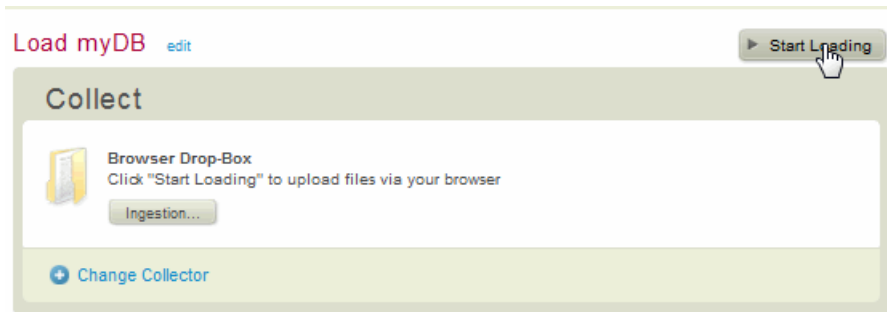




### 4.3.3 Using the Browser Drop-Box Collector

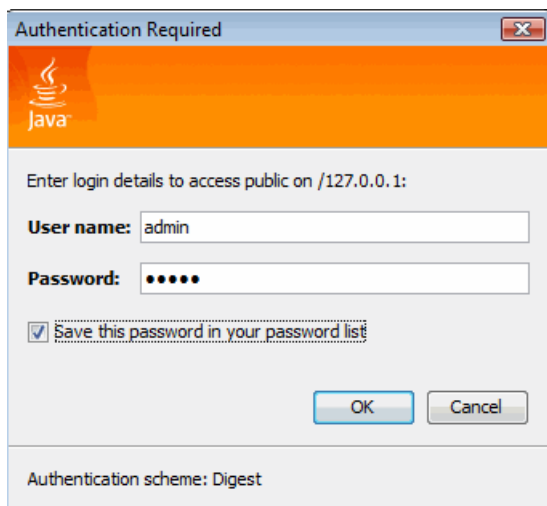
The Browser Drop-Box collector allows you to load all of the files dropped into a browser window into the database. The Ingestion Options specify the manner in which the documents are to be ingested into the database, as described in “Configuring Ingestion Options” on page 27.

1. To begin loading files into the database, click the Start Loading button:

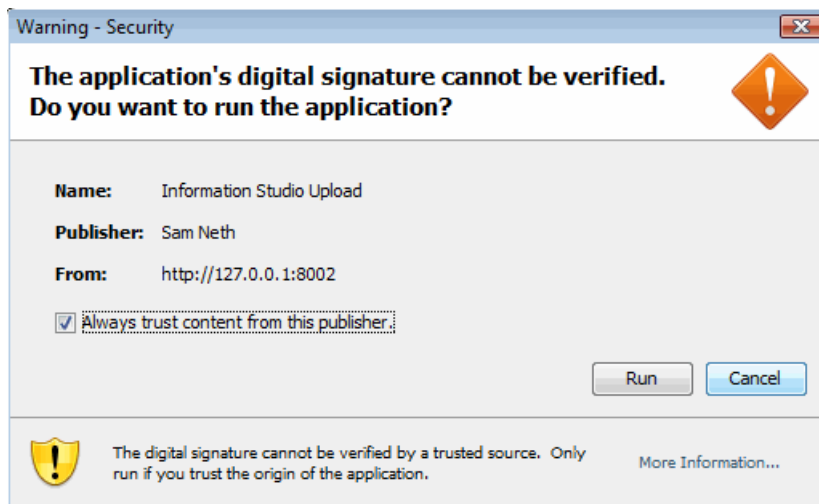


2. Initially, you will be prompted to log into MarkLogic Server and accept a certificate: Enter the same login credentials as you used to log in to Information Studio.

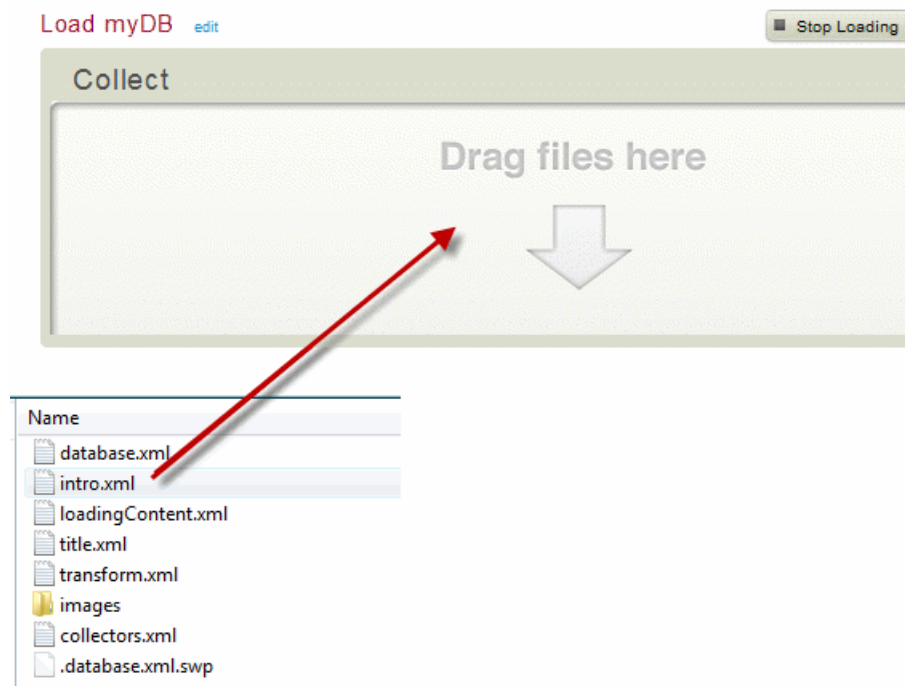
**Note:** If you are running MarkLogic Server on Mac OS, do not check the box that instructs Java to remember your login information if you want the option to use the Drop-Box Collector with different credentials.



3. When accepting the certificate from MarkLogic Server, click “Always trust content from this publisher” if you don’t want to be prompted again to accept the certificate when doing future uploads. Click Run:

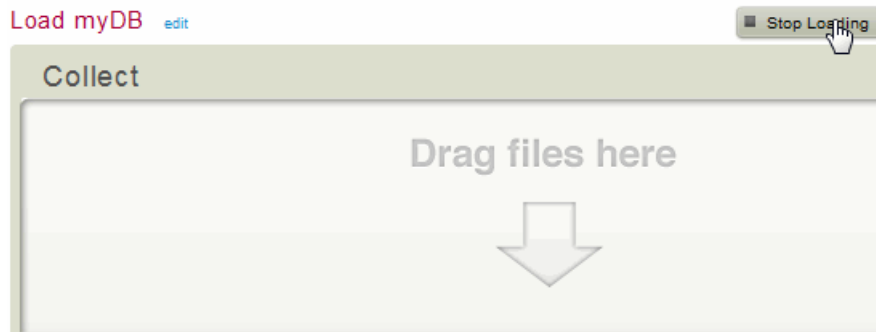


4. When you are ready to upload your files, select the files from your filesystem (for example, from an Explorer window) and drag them into the Drag Files Here area of the Collector:



**Note:** Navigating away from the Information Studio page while uploading will cancel the upload operation.

5. When you have finished uploading your files, click the Stop Loading button:



#### 4.3.4 Configuring Ingestion Options

The Ingestion Options specify the manner in which the documents are to be ingested into the database and under what URI they are to be stored. Here, you can fine-tune which documents are ingested from the Directory Path specified in your Collector Options by document format, encoding type, language, namespace, and a regular expression.

1. To configure the Ingestion Options, select Ingestion in the Collect box:



2. Configure the fields in the Ingestion Configuration window, as described in the table below.

**Note:** Most of the options in the following table are the same as the options passed to the `xmdp:document-load` function.

Field	Description
Documents per transaction	The maximum number of documents to be ingested in a single transaction. If ingesting more than the maximum, the ingest operation will be scheduled as more than one transaction.
Filtering	The filter used to select the documents in the filesystem. This can be any XQuery regular expression. The default regular expression specifies all documents in the directory and subdirectories.
Repair XML documents	Check this box to attempt to repair malformed XML content on each document during ingestion. If the box is left unchecked, malformed XML content is rejected and the document will generate an error.

Field	Description
Format	Ingest documents as a particular format, such as XML, Text, or Binary. Default indicates to ingest documents as any format. Documents that are not originally of the specified format will be converted to that format.
Encoding	Ingest documents as a particular encoding type, such as UTF-8, ASCII, and so on. See the <i>Search Developer's Guide</i> for a list of character set encodings by language. All encodings will be translated into UTF-8 from the specified encoding. The string specified for the encoding option will be matched to an encoding name according to the Unicode Charset Alias Matching rules ( <a href="http://www.unicode.org/reports/tr22/#Charset_Alias_Matching">http://www.unicode.org/reports/tr22/#Charset_Alias_Matching</a> ). Auto indicates to use an automatic encoding detector. If no encoding can be detected, the encoding defaults to UTF-8.
Language	Add an <code>xml:lang</code> attribute to the root element node on all ingested documents to indicate they are written in a particular language, such as English or French. Default indicates to not tag ingested documents with an <code>xml:lang</code> attribute.
Default namespace	<p>The namespace to use if there is no namespace at the root node of the document. The default value is "".</p> <p><b>Warning</b> If you specify a default namespace in a collector, you must also specify the same namespace in the transformers you use in the flow. Otherwise the transformer will not work.</p>

## 4.4 Transform

Transformation allow you to modify the content of files as they are loaded into the database. The topics in this section are as follows:

- [Selecting a Transformer](#)
- [Renaming Elements or Attributes](#)

### 4.4.1 Selecting a Transformer

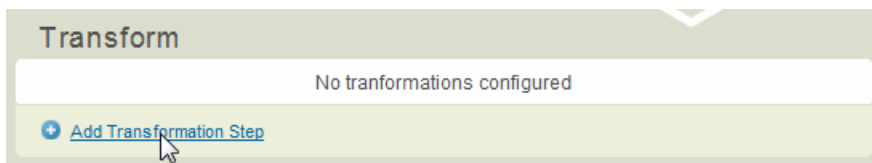
Transformers are plugins that modify your documents as they are loaded into the database. The following transformers are shipped with MarkLogic Server:

- XQuery — Allows you to add a custom XQuery to transform the documents.
- Rename — Renames a specific element or attribute in the loaded documents.
- XSLT — Allows you to define a custom XSLT stylesheet.
- Normalize Dates — Change dates in an element in the loaded documents.
- Schema Validation — Sets the schema validation level.
- Delete — Deletes a specific element or attribute from the loaded documents.

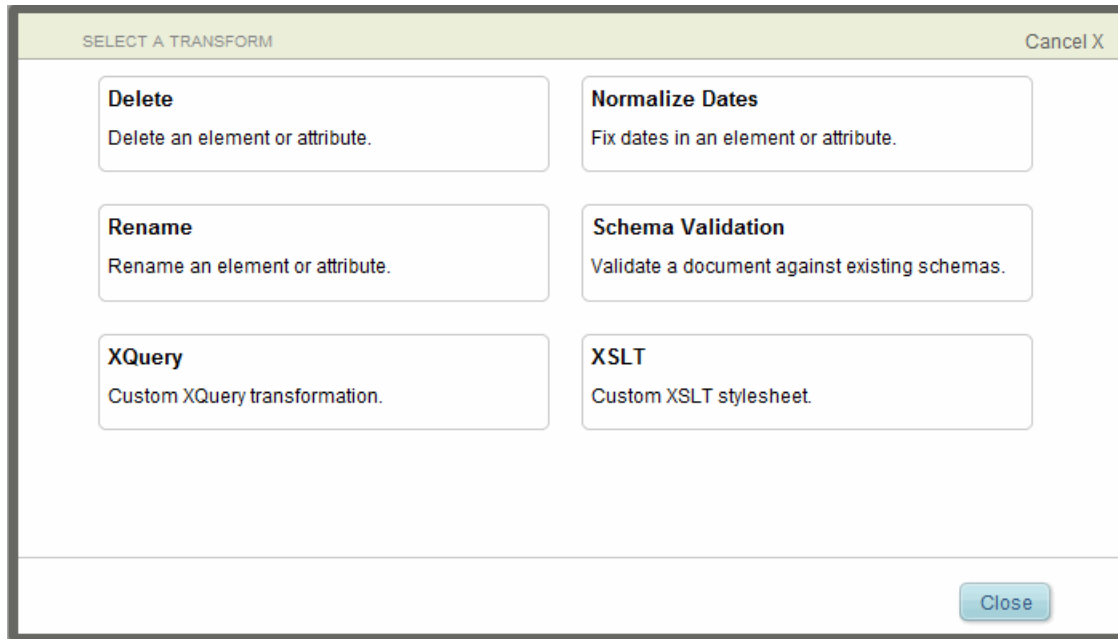
**Note:** There are performance implications when transforming documents during a load operation, so you must weigh the benefits of transforming documents during load against your performance needs.

How to use each of these transformers is described below. You can also create your own custom transformers, as described in “Creating Custom Transformers” on page 73.

1. You can enable a transformer by clicking Add Transformation Step at the bottom of the Transform section of your flow:



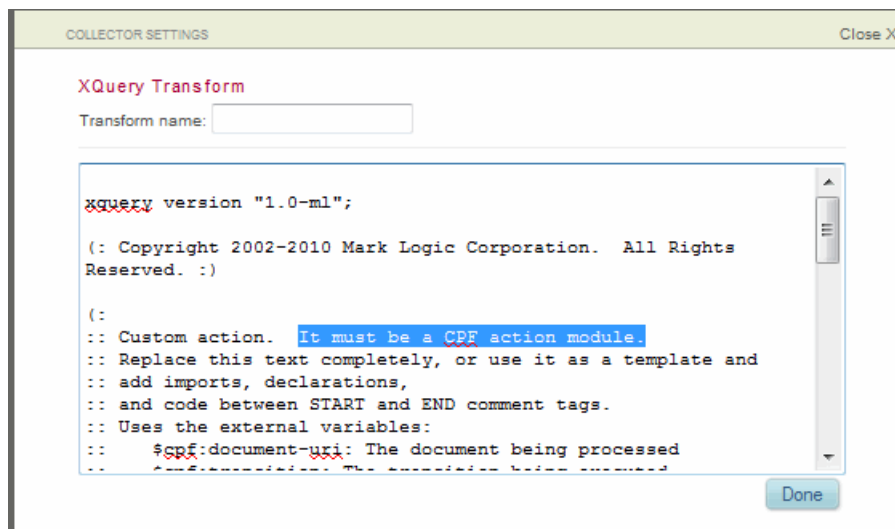
2. Select the type of transformer you want to add to your flow. You can add as many transformers to your flow as you wish.



#### 4.4.2 Custom XQuery

The Custom XQuery transformer allows you add a Content Processing Framework (CPF) action module to modify the loaded documents. The settings window provides a basic template for the CPF action module and directions on where to add your code. Alternatively, you can simply paste in your completed action module.

**Note:** The module you provide must be a valid CPF action module, or the pipeline will not function properly.



### 4.4.3 Renaming Elements or Attributes

The Rename transformer allows you can change the name of an element or attribute in one or more namespaces. Enter the name of the element to change in the Old Nodes Match field and its new name in the New QName field. Click Submit Query.

**CONFIGURE SETTINGS** Close X

**Rename Transform**

Transform name:

---

Match elements or attributes with this pattern:

Test

+ Add a namespace binding

Change the name of matching nodes to this name:

+ Add a namespace binding

Done

---

**Match patterns:**  
 Elements or attributes are selected with XSLT match patterns. To match elements or attributes in a namespace, add namespace bindings and use prefixes.

**Examples:**  
*oldname* - matches an element named *oldname*  
*@avalue* - matches an attribute named *avalue*; more complex patterns are possible  
*topic[contains(@class,'task')]* - matches an element named *topic* that has an attribute named *class* that contains the string "task".

If you have specified a default namespace in your collector, click ‘Add a namespace binding’ and specify the same default namespace:

**CONFIGURE SETTINGS**

**Rename Transform**

Transform name:

---

Match elements or attributes with this pattern:

Test

The following namespace bindings are in scope for the match pattern:

Prefix	Namespace URI
mns	http://myco.com/namespace/mynamespace

+ Add a namespace binding

Change the name of matching nodes to this name:

The following namespace binding is in scope for the new name:

Prefix	Namespace URI
mns	http://myco.com/namespace/mynamespace

Done



#### 4.4.4 Deleting Elements or Attributes

The Delete transformer allows you can remove an element or attribute. You can also isolate the element or attribute in a specific namespace to be deleted. Enter the name of the element to delete in the Element Localname field. If deleting an attribute, enter the name of its element in the Element Localname field and the name of the attribute in the Attribute localname field. Click Submit Query.

The screenshot shows the 'Delete Transform' configuration window. At the top, there's a title bar with 'CONFIGURE SETTINGS' and a 'Close X' button. Below the title bar, the section is titled 'Delete Transform'. There is a text input field for 'Transform name:'. Below that, a section titled 'Delete elements or attributes that match this pattern:' contains a text input field with 'match-expression' and a 'Test' button. Below this is a '+ Add a namespace binding' button. At the bottom right is a 'Done' button. Below the 'Test' button, there is a section titled 'Match patterns:' with explanatory text: 'Elements or attributes are selected with XSLT match patterns. To match elements or attributes in a namespace, add namespace bindings and use prefixes.' and 'Examples: oldname - matches an element named oldname; @avalue - matches an attribute named avalue; more complex patterns are possible; db:phrase[@revisionflag='deleted'] - matches an element named db:phrase that has an attribute named revisionflag with the value "deleted". The binding for "db" must be provided.'

If you have specified a default namespace in your collector, click ‘Add a namespace binding’ and specify the same default namespace:

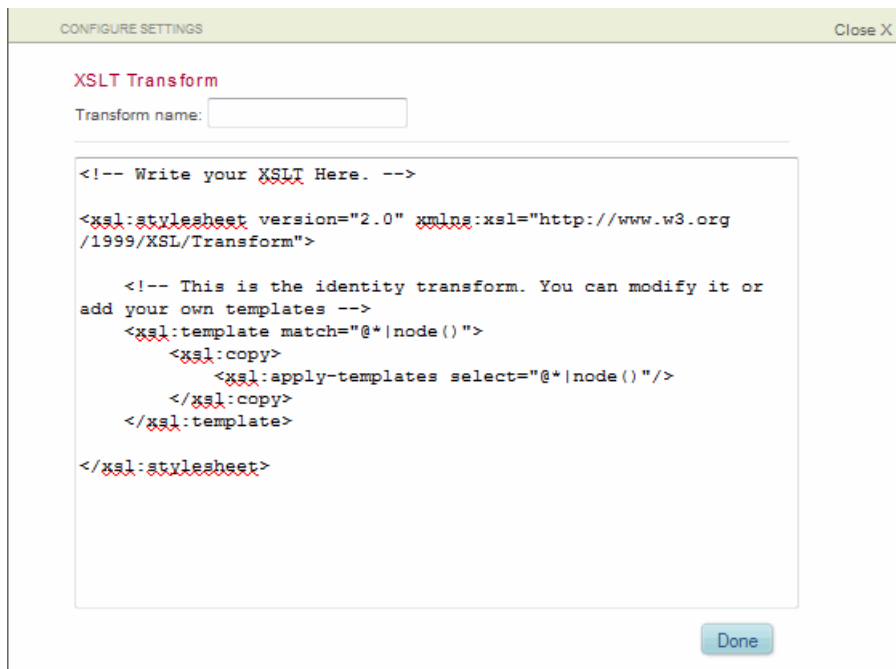
This screenshot shows the 'Delete Transform' configuration window with a namespace binding added. The 'Transform name:' field is empty. The 'Delete elements or attributes that match this pattern:' section has a text input field with 'myelement' and a 'Test' button. Below this, a section titled 'The following namespace bindings are in scope:' contains a table with two columns: 'Prefix' and 'Namespace URI'. The table has one row with 'mns' in the 'Prefix' column and 'http://myco/namespaces/mynamespace' in the 'Namespace URI' column. To the right of the table is a green 'X' icon. Below the table is a '+ Add a namespace binding' button. At the bottom right is a 'Done' button.

Prefix	Namespace URI
mns	http://myco/namespaces/mynamespace

### 4.4.5 Customize XSLT Stylesheet

The XSLT transformer allows you to create a custom XSLT stylesheet to apply to the loaded documents. The XSLT stylesheet allows you to modify document content, properties, permissions, and collections.

**Note:** XSLT cannot be used on binary documents, so they are ignored by this transformer and are passed through unchanged.



The screenshot shows a 'CONFIGURE SETTINGS' dialog box with a 'Close X' button in the top right corner. The title of the dialog is 'XSLT Transform'. Below the title is a 'Transform name:' label followed by an empty text input field. The main area of the dialog contains a text editor with the following XSLT code:

```
<!-- Write your XSLT Here. -->

<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org
/1999/XSL/Transform">

  <!-- This is the identity transform. You can modify it or
add your own templates -->
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()" />
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

At the bottom right of the dialog is a 'Done' button.

If you are importing an XSLT stylesheet, the stylesheet must have read permission for the `infostudio-user` role and be stored should be under the `/actions` root in the `App-Services` database. Information Studio expects XSLT stylesheets to be located in the `/actions` directory, so it's not necessary to specify the root directory in your import statement.

For example, to import the stylesheet, `/actions/a.xsl`, the import should look like:

```
<xsl:import href="a.xslt"/>
```

### 4.4.6 Normalize Dates

The Normalize Dates transformer allows you to specify a text transformation into an `xs:date` or `xs:dateTime` value on certain elements in the documents.

**Normalize Dates Transform**

Transform name:

Normalize **DD/MM/YYYY** ▼ dates  
in elements or attributes that match:

Test

+ Add a namespace binding

Normalize by **replacing the value** ▼ Done

**Match patterns:**  
Elements or attributes are selected with XSLT match patterns. To match elements or attributes in a namespace, add namespace bindings and use prefixes. The prefixes that you define apply to both the match pattern and the new attribute name, if you specify that the normalized date should appear in a new attribute.

**Examples:**  
 pubdate - matches an element named pubdate  
 @date - matches an attribute named date; more complex patterns are possible  
 revdate[@status='approved' and not(author)] - matches an element named revdate that has an attribute named status with the value approved and does not have a child element named author.

Specify the date format

**Normalize Dates Transform**

Transform name:

Normalize **DD/MM/YYYY** ▼ dates in elements or attributes that match:

Test

+ Add a namespace binding

Normalize by **replacing the value** ▼ Done

**Match patterns:**  
Elements or attributes are selected with XSLT match patterns. To match elements or attributes in a namespace, add namespace bindings and use prefixes. The prefixes that you define apply to both the match pattern and the new attribute name, if you specify that the normalized date should appear in a new attribute.

**Examples:**  
 pubdate - matches an element named pubdate  
 @date - matches an attribute named date; more complex patterns are possible

You can add the date as a new element, attribute to an existing element, or overwrite and value of an existing element or attribute with the date.

**Normalize Dates Transform**

Transform name:

---

Normalize  dates  
in elements or attributes that match:

+ Add a namespace binding

Normalize by

replacing the value  
adding a new attribute

If you have specified a default namespace in your collector, click ‘Add a namespace binding’ and specify the same default namespace for your element and attribute (if applicable):

**Normalize Dates Transform**

Transform name:

---

Normalize  dates  
in elements or attributes that match:

The following namespace bindings are in scope for the match pattern:

Prefix	Namespace URI
mns	http://myco/namespaces/mynamespace

+ Add a namespace binding

Normalize by  with this name:

The following namespace binding is in scope for the new attribute name:

Prefix	Namespace URI
mns	http://myco/namespaces/mynamespace

### 4.4.7 Validate Documents against Schema

The Schema Validation transformer specifies the level at which the XML of loaded documents are to be validated against the schema. Specify `strict` to make validation failures fatal or `lax` to produce a warning and continue processing. The default setting is `strict`. For details on the validation levels, see [Validate Expression](#) in the *XQuery and XSLT Reference Guide*.

The screenshot shows a 'CONFIGURE SETTINGS' dialog box with a 'Close X' button in the top right corner. The title is 'Schema Validation Transform'. Below the title, there is a 'Transform name:' label followed by a text input field containing 'Check Schema'. Underneath, the 'Validation mode:' label is followed by a dropdown menu currently set to 'strict'. A mouse cursor is hovering over the dropdown, which shows 'strict' and 'lax' as options. To the right of the dropdown is a 'Done' button. Below the dropdown, the text 'Validation mode:' is followed by a description: 'Validation mode specifies the initial validation mode:'. This is followed by two entries: 'strict' with the description 'In strict mode, there must be a schema for the document element and the entire document must be valid.' and 'lax' with the description 'In lax mode, the server will validate elements if it can find schema information about them, but will otherwise silently accept them.' At the bottom, a 'Note:' section states 'Validation is performed using schemas from the 'Schemas' database.'

## 4.5 Load

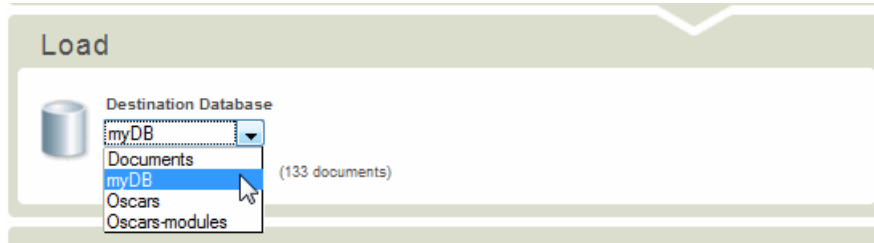
The Load section of the Flow Editor window allows you to configure which database the flow is to load documents. In addition, you can configure the URI structure under which the documents are to be loaded, their access permissions, and the collections under which the documents are to be grouped. The topics in this section are:

- [Selecting the Database](#)
- [Document Settings](#)

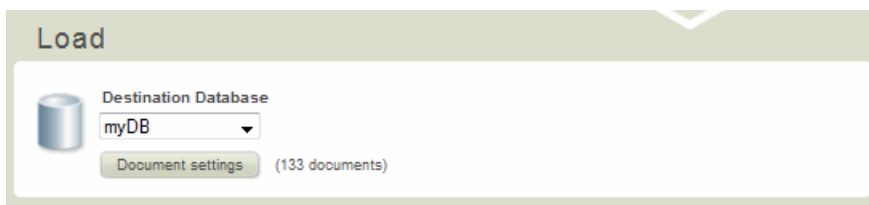
### 4.5.1 Selecting the Database

To select the database for the flow, do the following:

1. Select the database into which you want this flow to load the content:

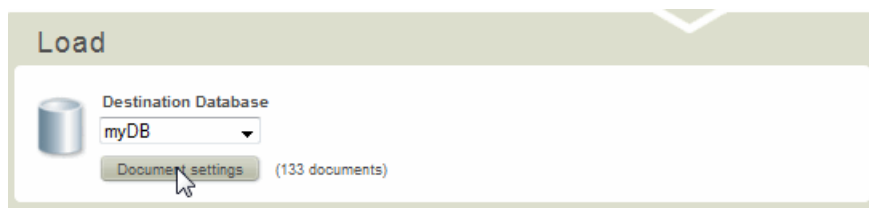


2. The total number of documents currently in the database are displayed on the right:



### 4.5.2 Document Settings

The Document Settings allow you to assign specific attributes to the documents that are loaded into the database.



The Document Settings options are:

- [Configuring the URI Structure](#)
- [Configuring Document Access Permissions](#)
- [Configuring Collections](#)
- [Configuring Quality Boost](#)

### 4.5.2.1 Configuring the URI Structure

The URI Structure Configuration window allows you to configure the URIs of your loaded documents.

1. To configure the URI structure, select the URI tab in the Document Settings window.
2. In the URI Structure Configuration window, you can specify the URI structure of incoming documents and how to handle conflicts between incoming documents and documents that already exist in the database:

The screenshot shows the 'DOCUMENT SETTINGS' window with the 'URI' tab selected. The 'URI Structure Configuration' section contains a text field for the URI template, currently set to `/content{ $path}/{ $filename}. { $ext}`. Below this, there are three radio buttons for handling conflicts: 'Ignore Incoming document', 'Replace' (which is selected), and 'Error'. Further down, there is a section for 'URI substitutions' with definitions for `{ $path}`, `{ $filename}`, `{ $ext}`, and `{ $guid}`. Below that, there are 'Examples' showing two URI templates: `/content{ $path}/{ $filename}. { $ext}` and `/category{ $ext}/{ $guid}/{ $filename}. { $ext}`. A 'Done' button is located at the bottom right of the window.

DOCUMENT SETTINGS Close X

URI Permissions Collections Quality Boost

**URI Structure Configuration**

URI

If a document already exists at this URI

☐ Ignore Incoming document

☒ Replace

☐ Error

**URI substitutions:**

*{ \$path} - represents the directory path not including the file name, for example /space/content*

*{ \$filename} - represents the base name of the file, for example invoice*

*{ \$ext} - represents the file extension, for example xml*

*{ \$guid} - represents a generated globally unique ID*

**Examples:**

*/content{ \$path}/{ \$filename}. { \$ext}*

*/category{ \$ext}/{ \$guid}/{ \$filename}. { \$ext}*

Done

3. The URI setting defines the structure of the URI under which files are loaded into the database. A URI can be made up of some or all the following elements and they can be organized in any order:

Element	What it is	Example
{ \$guid }	Globally Unique ID. This specifies to generate a globally unique ID for each file loaded into the database.	15908936213503297716
{ \$path }	The directory path to the file. By default, this is the same path under which the file is located in the filesystem.	C:/latest/docs
{ \$path strip-prefix=" " }	You can add a <code>strip-prefix</code> inside { \$path } to remove a prefix from the upload directory. (See example below.)	
{ \$filename }	The name of the file.	myfile
{ \$ext }	The file extension. Note that the dot (.) must be specified in the URI structure as a literal.	xml

To change the URI structure, simply modify the structure of the URI in the URI field.

For example, the default URI is:

```
/content{ $path }/{ $filename }. { $ext }
```

This means a file named, `C:/mydir/mydocument.xml` will be loaded with the following URI:

```
/contentC:/mydir/mydocument.xml
```

Changing the URI structure to:

```
/content{ $path strip-prefix="C:/mydir" }/{ $filename }. { $ext }
```

results in the URI:

```
/content/mydocument.xml
```



Changing the URI structure to:

```
/http://mydir/{$filename}.${$ext}
```

results in the URI:

```
/http://mydir/mydocument.xml
```

Changing the URI structure to:

```
/mydir/{$filename}
```

results in the URI:

```
/mydir/mydocument
```

4. The 'If a document already exists at this URI' buttons provide the following options for handling incoming documents that have the same URI as an existing document in the database:

Option	Description
Ignore Incoming document	Do not update an existing document in the database with the incoming document.
Replace	Replace the existing document in the database with the incoming document.
Error	Generate an error if an existing document in the database has the same URI as the incoming document.

### 4.5.2.2 Configuring Document Access Permissions

As described in [Permissions on Documents](#) in the *Understanding and Using Security Guide*, you can specify permissions on the ingested documents to control which users can access them and in what manner.

1. To set the permissions for the documents ingested by this flow, select the Permissions tab in the Document Settings window.
2. Enter the Role for which you want to set a permission, then select the permission to assign to the role from pull-down menu:

DOCUMENT SETTINGS

URI Permissions Collections Quality Boost

Destination Permissions

Role app-user Read X

Role app-user Insert X

Role app-user Read X

+ New permission

Done

3. To add a new permission, click New Permission and repeat the procedure described above. You can add as many permissions for as many roles as you like:

DOCUMENT SETTINGS

URI Permissions Collections Quality Boost

Destination Permissions

Role app-user Read X

Role app-user Insert X

Role app-user Execute X

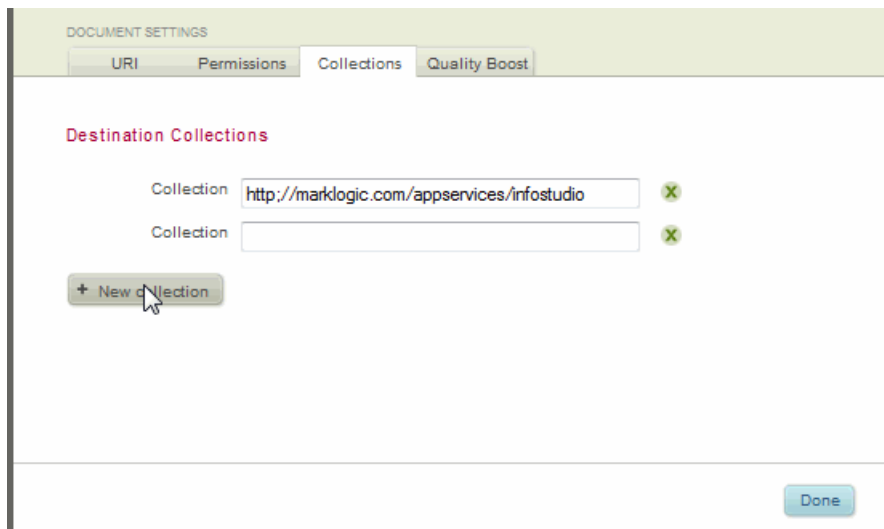
+ New permission

Done

### 4.5.2.3 Configuring Collections

Collections are described in detail in [Protected Collections](#) in the *Administrator's Guide*. This section describes how to specify the collections to be associated with the documents ingested by this flow.

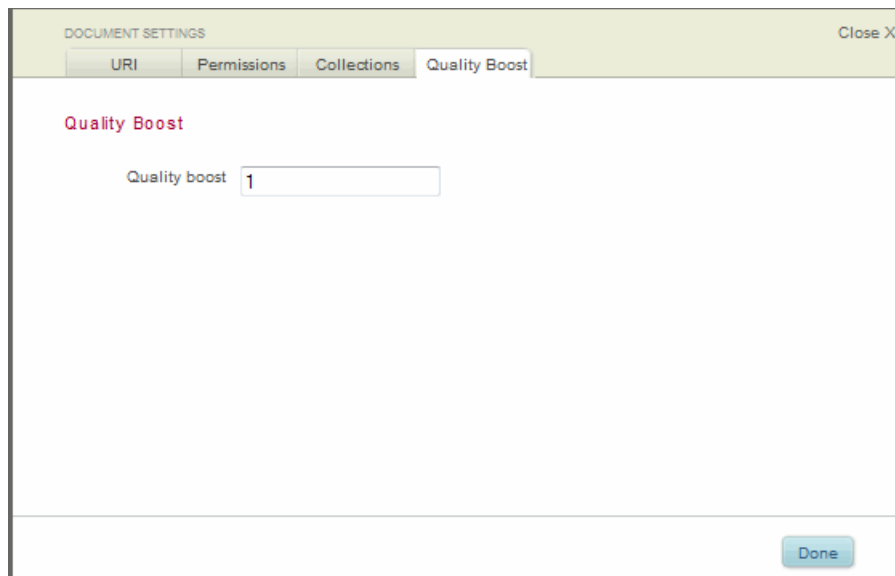
1. To set the collections for the documents ingested by this Flow, select the Collections tab in the Document Settings window.
2. You can add or remove collections in the Destination Collections window:



#### 4.5.2.4 Configuring Quality Boost

Quality Boost associates all ingested documents with the specified quality value. A positive value increases the relevance score of the document in text search functions. The converse is true for a negative value. Leaving this field blank specifies the default document quality, which is 0.

To set the quality boost for the documents ingested by this Flow, select the Quality Boost tab in the Document Settings window:



The screenshot shows the 'DOCUMENT SETTINGS' window with the 'Quality Boost' tab selected. The window has a title bar with 'DOCUMENT SETTINGS' and a 'Close X' button. Below the title bar are four tabs: 'URI', 'Permissions', 'Collections', and 'Quality Boost'. The 'Quality Boost' tab is active, showing a red heading 'Quality Boost' and a text input field labeled 'Quality boost' with the value '1'. A 'Done' button is located at the bottom right of the window.

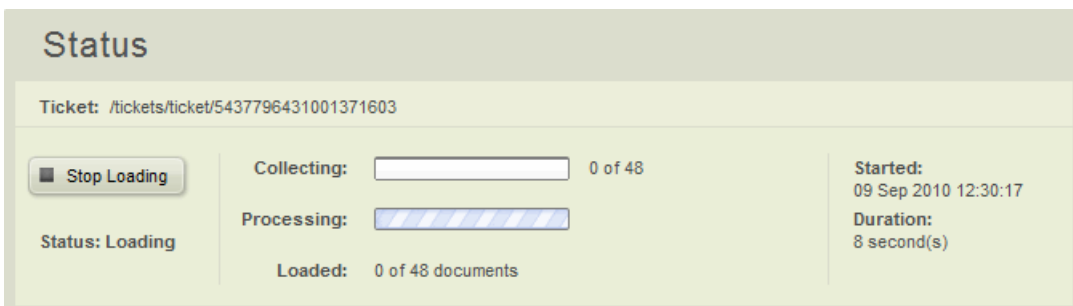
## 4.6 Launching Ingestion and Tracking Status

The Status portion of the flow shows the status of the ingest operations and any resulting errors when ingestion has completed.

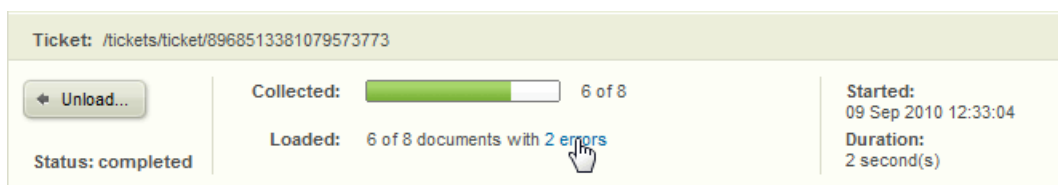
1. When you have finished configuring your flow and are ready to begin ingesting documents into the database, click the Start Load button:



2. The Status portion of your flow displays the ticket status and progress of the documents at the Collecting, Processing, and Loaded stages of the flow:



3. When the ingestion has completed, the Status indicates the ticket status as 'completed' and the number of documents that were successfully ingested into the database. If there were errors, you can click on the errors link.




4. The errors window includes both collection errors and processing errors. Click on any error for more detail.

Collection Errors (2)		Processing Errors (0)	
The following documents were unable to be processed and have not been loaded into the database			
Document	Error	Message	Time
c:\test\wizard.xml	<a href="#">XDMP-DOCENTITYREF</a>	Invalid entity reference	03 Jun 2010 14:35:54
c:\test\foo.xml	<a href="#">XDMP-DOCENTITYREF</a>	Invalid entity reference	03 Jun 2010 14:35:54

5. If you wish to remove the loaded documents from the database, Click Unload:

Ticket: /tickets/ticket/8968513381079573773

← Unload...  
Status: completed

Collected:  6 of 8


Loaded: 6 of 8 documents with 2 errors

Started:  
09 Sep 2010 12:33:04

Duration:  
2 second(s)

## 4.7 Deleting a Flow

To delete a flow, click the Delete link associated with the flow in the Information Studio Flows section of the Application Services page. Click OK in the ‘Are you sure you'd like to delete this flow?’ popup window.

 **Information Studio Flows**  
*Load content into a database via a flow*

Flow Name	Database	Last Run	Collected / Loaded / Errors	
<a href="#">Load myDB</a>	myDB	09 Sep 2010 12:33:36	6 / 6 / 2	<a href="#">Delete</a>

+ New Flow

## 5.0 Scripting Information Studio Tasks

You can use the `info` API to programmatically accomplish the same tasks as described for the Information Studio interface in chapters “Creating and Configuring Databases” on page 10 and “Creating and Configuring Flows” on page 21.

This chapter describes:

- [The info API](#)
- [Creating a Database](#)
- [Loading Data into Databases](#)
- [Establishing Ingestion Policies](#)
- [Applying Ingestion Policies](#)
- [Ingestion Policies and Multiple Load Operations](#)

### 5.1 The info API

The `info` API provides functions that allow you to easily create databases and load them with data. The `info` API functions that manage databases are built on top of the `admin` API described in the *Scripting Administrative Tasks Guide*. In addition, the `info` API provides functions that greatly simplify and enhance database load operations.

### 5.2 Creating a Database

The `info:database-create` function greatly simplifies the task of programmatically creating a forests and databases. The task of creating forests and databases using the `admin` API is described in [Creating and Configuring Forests and Databases](#) in the *Scripting Administrative Tasks Guide*. When using the `info:database-create` function, you are trading the finer-level control provided by the `admin` functions for simplicity.

For example, the following function creates a new database, named `Sample-Database`, with two forests per host. By default, the database is located in the `Default` group and the forest data is placed in the default location (`/MarkLogic/Data/Forests`) on each host in the `Default` group. Each forest is given a name like `Sample-Database-<unique-id>`, where `<unique-id>` is a unique number generated by the API. The `Sample-Database` database is configured with the default security and schema databases, `Security` and `Schemas`.

```
xquery version "1.0-ml";

import module namespace info =
  "http://marklogic.com/appservices/infostudio"
  at "/MarkLogic/appservices/infostudio/info.xqy";

info:database-create("Sample-Database", 2)
```

The `info:database-create` function provides optional parameters to control the location of your forest data, as well as which databases to use to manage security, schema and trigger data. The function also accepts a `group` parameter. The `info` API determines which hosts are in the group and creates the specified number of forests for each host in the group.

For example, the following function creates a `Sample-Database` with three forests per host. The database is located in the `MyGroup` group and the forest data is placed in the `c:\myData` directory on each host in the `MyGroup` group. The security database is `MySecurity`, the schema database is `MySchemas` and the triggers database is `MyTriggers`.

```
xquery version "1.0-ml";

import module namespace info =
  "http://marklogic.com/appservices/infostudio"
  at "/MarkLogic/appservices/infostudio/info.xqy";

info:database-create(
  "Sample-Database",
  3,
  "MyGroup",
  "c:\myData",
  "MySecurity",
  "MySchemas",
  "MyTriggers")
```

### 5.3 Configuring the Database Text Indexes

You can configure the database Text Indexes by means of the `info:database-set-feature` function. This function allows you to configure the database in a manner similar to that described in “Configuring Text Indexes” on page 14.

For example, the following query enables both Wildcards and Positions:

```
xquery version "1.0-ml";

import module namespace info =
  "http://marklogic.com/appservices/infostudio"
  at "/MarkLogic/appservices/infostudio/info.xqy";

let $settings :=
<settings xmlns="http://marklogic.com/appservices/infostudio">
  <wildcard>true</wildcard>
  <position>true</position>
  <reverse>false</reverse>
</settings> )

return
  info:database-set-feature("Sample-Database", $settings)
```



The following table lists the possible elements in a database `settings` node, as well as their purpose and possible values:

Element	Description	Possible Values
<code>wildcard</code>	Enables Three Character Searches and Codepoint Word Lexicon indexing. Use this setting for more efficient wildcard searches on the documents in your database.	true false
<code>position</code>	Enables Word Positions indexing. Use this setting for more efficient phrase searches on the documents in your database.	true false
<code>reverse</code>	Enables Fast Reverse Searches. Use this setting to index saved queries in order to speed up reverse query searches. This option requires a special license.	true false

## 5.4 Loading Data into Databases

The `info` API allows you to script the operations described in “Creating and Configuring Flows” on page 21.

When a database load operation is initiated, Information Studio immediately returns a ticket URI. You can pass the ticket URI to the `info:ticket` function to return the contents of the ticket, which includes the status of the load and any errors encountered. Load operations are asynchronous, so the ticket is returned before the load operation has completed. Information Studio updates the status of the ticket during the load operation. Initially, the ticket status is ‘active.’ When the load has completed, the ticket status is updated to ‘completed.’ Under special circumstances, other statuses can be set on the ticket. These will be described later in this chapter.

The simplest way to load data into the database is by calling the `info:load` function. The following example loads the files from the `C:\mydocs` directory into the `Sample-Database`:

```
xquery version "1.0-ml";

import module namespace info =
  "http://marklogic.com/appservices/infostudio"
  at "/MarkLogic/appservices/infostudio/info.xqy";

info:load("C:\mydocs", (), (), "Sample-Database")
```

The `info:load` function also allows you to specify an Ingestion Policy and/or deltas for an Ingestion Policy to fine-tune how documents are to be loaded into the database. Ingestion Policies are discussed in “Establishing Ingestion Policies” on page 50.

## 5.5 Establishing Ingestion Policies

When using the Information Studio interface, you establish an Ingestion Policy by means of the Ingestion Settings described in “Configuring Ingestion Options” on page 27. This section describes how to programmatically establish and use Ingestion Policies.

Ingestion Policies control which documents are loaded into the database and to what URI location, as well as the permissions, collections, and format type to be assigned to each document. If you are bulk loading a large number of files into a database, you may want to break the load operation into multiple transactions. Ingestion policies allow you to control the maximum number of files to be loaded during a single transaction. Ingestion policies also allow you to control whether to overwrite existing files in the database or generate an error when an attempt is made to overwrite an existing file.

You can create a ‘default’ policy to be used in the event no policy is specified for a load operation. When you set a policy, you need only specify the options you want to change. Information Studio will then merge your changes with the global default policy settings.

The following is an example of a simple ‘default’ Ingestion Policy:

```
let $policy :=
<options name="default" xmlns="http://marklogic.com/appservices/
infostudio">
  <collection>http://marklogic.com/appservices/infostudio</collection>
  <error-handling>continue-with-warning</error-handling>
  <fab-retention-duration>P30D</fab-retention-duration>
  <file-filter>^[^\.]</file-filter>
  <max-docs-per-transaction>100</max-docs-per-transaction>
  <overwrite>overwrite</overwrite>
  <ticket-retention-duration>P30D</ticket-retention-duration>
  <uri>
    <literal>/content</literal>
    <filename/>
    <literal>.</literal>
    <ext/>
  </uri>
</options>
```

You can ‘set’ the Ingestion Policy as the ‘default’ policy by calling the `info:policy-set` function, as follows:

```
info:policy-set("default", $policy)
```

The following table lists all of the possible elements in an Ingestion Policy, as well as their purpose and possible values:

Element	Description	Possible Values and Default Value
<code>annotation</code>	A description of the policy, or any other notation.	Any string  Default: None
<code>overwrite</code>	Specify how to manage files that already exist in the database.  Specify <code>overwrite</code> to overwrite existing files in the database; <code>skip</code> to not overwrite the files, but continue with the load, or <code>error</code> to not overwrite the files and generate an error.	<code>overwrite</code> <code>skip</code> <code>error</code>  Default: <code>overwrite</code>
<code>error-handling</code>	How to handle load errors. Specify <code>continue-with-warning</code> to continue the load or <code>error</code> to abort the load when an error is encountered.	<code>continue-with-warning</code> <code>error</code>  Default: <code>continue-with-warning</code>
<code>collection</code>	The URI of a collection.  By default, any existing collections will be overridden by the specified collection. You can use the <code>add</code> attribute to add the collection to any existing collections, rather than overriding them.	The collection URI.  Default: None
<code>max-docs-per-transaction</code>	The maximum number of documents to be ingested in a single transaction. If ingesting more than the maximum, the ingest operation will be scheduled as more than one transaction.	Any <code>xs:unsignedInt</code>  Default: 100
<code>file-filter</code>	The filter used to select the documents in the filesystem. This can be any XQuery regular expression. The default regular expression specifies all documents in the directory and its subdirectories except for those that start with a dot, such as <code>.mydoc</code> .	Any valid XQuery regular expression  Default: <code>^[^\.]</code>

Element	Description	Possible Values and Default Value
<code>repair</code>	Specify <code>full</code> to attempt to repair malformed XML content on each document during ingestion. Specifying no value or <code>none</code> will cause documents containing malformed XML content to be rejected with an error.	none full  Default: None
<code>format</code>	Ingest documents as a particular format, such as XML, Text, or Binary. No value indicates to ingest documents as any format. Documents that are not of the specified format will generate an error.	xml text binary  Default: None
<code>default-namespace</code>	Apply a default namespace to all the nodes that do not have an associated namespace.	The namespace URI.  Default: None
<code>default-language</code>	Add an <code>xml:lang</code> attribute to the root element node on all ingested documents to indicate they are written in a particular language, such as English or French. Default indicates to not tag ingested documents with an <code>xml:lang</code> attribute.	ar de en es fa fr it ko nl pt ru zh zh-Hant  Default: None
<code>uri</code>	The URI structure for the ingested documents in the database. For a complete discussion, see “Configuring the URI Structure” on page 39.	<pre>&lt;literal/&gt; &lt;path @[strip-prefix]/&gt; &lt;guid/&gt; &lt;filename/&gt; &lt;ext/&gt;</pre> Default: <pre>&lt;literal&gt;   /content &lt;/literal&gt; &lt;path/&gt; &lt;literal&gt;.&lt;/literal&gt; &lt;filename/&gt; &lt;literal&gt;.&lt;/literal&gt; &lt;ext/&gt;</pre>

Element	Description	Possible Values and Default Value
encoding	Ingest documents as a particular encoding type, such as UTF-8, ASCII, and so on. See the <i>Search Developer's Guide</i> for a list of character set encodings by language. All encodings will be translated into UTF-8 from the specified encoding. The string specified for the encoding option will be matched to an encoding name according to the Unicode Charset Alias Matching rules ( <a href="http://www.unicode.org/reports/tr22/#Charset_Alias_Matching">http://www.unicode.org/reports/tr22/#Charset_Alias_Matching</a> ). Auto indicates to use an automatic encoding detector. If no encoding can be detected, the encoding defaults to UTF-8.	A valid encoding type  Default: UTF-8
filesize-limit-kb	Specifies the maximum size a file can be without generating a load error.	xs:unsignedInt  Default: None
permission	Specifies the permissions to set on the loaded documents. This is expressed in the form:  <pre>&lt;permission&gt;   &lt;role&gt;role&lt;/role&gt;   &lt;capability&gt;permission&lt;/capability&gt; &lt;/permission&gt;</pre>	Possible roles are:  app-user alert-user alert-admin alert-execution dls-admin dls-user flexrep-admin flexrep-user infostudio-user  As well as any custom roles you have created.  Possible permissions are:  read insert update execute  Default: None

Element	Description	Possible Values and Default Value
quality	Associate all ingested documents with the specified quality value. A positive value increases the relevance score of the document in text search functions. The converse is true for a negative value. Leaving this field blank specifies the default document quality.	<code>xs:integer</code> Default: 1
forest	The name of a specific forest in which to load the documents.	<code>xs:string</code> Default: None
ticket-retention-duration	The length of time to keep the state data for tickets in the <code>App-Services</code> database. For an overview of the <code>App-Services</code> database, see “Application Services App Server and Databases” on page 6.	<code>xs:duration</code> Default: <code>P30D</code> (30 days)
fab-retention-duration	The length of time to keep the document ingestion data generated by the load operation in the <code>Fab</code> database. For an overview of the <code>Fab</code> database, see “Application Services App Server and Databases” on page 6.	<code>xs:duration</code> Default: <code>P30D</code> (30 days)

## 5.6 Applying Ingestion Policies

The `info:load` function allows you to name a stored Ingestion Policy to be used for the load operation, as well as a set of specific options (deltas) that selectively override the stored policy. If no Ingestion Policy is specified for a load operation, the ‘default’ policy is used. If no ‘default’ policy has been specified, then a policy consisting of the global defaults are applied to the load operation.

For example, the following query loads the documents from the `C:\mydocs` directory into the `Sample-Database` using the above ‘default’ ingestion policy:

```
xquery version "1.0-ml";

import module namespace info =
  "http://marklogic.com/appservices/infostudio"
  at "/MarkLogic/appservices/infostudio/info.xqy";

return
  info:load("C:\test", (), (), "Sample-Database")
```

To change the URI to `http://docs/mydocs`, you can define a delta that changes the `literal` value in the URI. This delta change will only apply to this load operation and leaves the URI in the ‘default’ Ingestion Policy unchanged.

```
xquery version "1.0-ml";

import module namespace info =
  "http://marklogic.com/appservices/infostudio"
  at "/MarkLogic/appservices/infostudio/info.xqy";

let $delta :=
  <options name="default"
    xmlns="http://marklogic.com/appservices/infostudio">
    <uri>
      <literal>http://docs/mydocs/</literal>
      <filename/>
      <literal>.</literal>
      <ext/>
    </uri>
  </options>

return
  info:load("C:\test", (), $delta, "Sample-Database")
```

**Note:** When defining deltas, only the children elements of the root element are preserved in the Ingestion Policy. So, in the above example, the children of the `uri` element must be defined in their entirety in order for the filenames and extensions to be included in the URI.

## 5.7 Ingestion Policies and Multiple Load Operations

If you are initiating multiple load operations that require changes to the Ingestion Policy, it is important to understand that load operations are asynchronous. A load operation returns the ticket immediately before loading the files into the database. Any changes applied to an Ingestion Policy after launching a load operation may impact the policy set for the previous load.

For example, the following pseudo query changes the policy between loads, which may produce unexpected results:

```
let $mypolicy := info:policy-set("mypolicy", set options)
return info:load($dirpath, "mypolicy", (), $database),

let $mypolicy := info:policy-set("mypolicy", change options)
return info:load($dirpath, "mypolicy", (), $database),

let $mypolicy := info:policy-set("mypolicy", change options)
return info:load($dirpath, "mypolicy", (), $database)
```

The solution to this is to define unique deltas that define the changes to the policy and pass them to the `info:load` function, as shown in the pseudo query below:

```
let $mypolicy := info:policy-set("mypolicy", set options...)
return info:load($dirpath, "mypolicy", (), $database),

let $delta1 := change options
return info:load($dirpath, "mypolicy", $delta1, $database),

let $delta2 := change options
return info:load($dirpath, "mypolicy", $delta2, $database)
```



For example, you want to load some modules into one URI and some 4.1 and 4.2 scripts into their own unique URIs. This could be done by defining a policy with the correct URI for the modules and then defining a delta to change the URI for each set of scripts:

```
xquery version "1.0-ml";

import module namespace info =
  "http://marklogic.com/appservices/infostudio"
  at "/MarkLogic/appservices/infostudio/info.xqy";

(: Create a policy with a URI for the modules :)
let $mypolicy := info:policy-set(
  "mypolicy",
  <options xmlns="http://marklogic.com/appservices/infostudio">
    <collection>
      http://marklogic.com/appservices/infostudio
    </collection>
    <error-handling>continue-with-warning</error-handling>
    <fab-retention-duration>P30D</fab-retention-duration>
    <file-filter>^[^\.]</file-filter>
    <max-docs-per-transaction>100</max-docs-per-transaction>
    <overwrite>overwrite</overwrite>
    <ticket-retention-duration>P30D</ticket-retention-duration>
    <uri>
      <literal>http://pubs/modules/actions/</literal>
      <filename/>
      <literal>.</literal>
      <ext/>
    </uri>
  </options>)

(: Define a delta to change the URI for the 4.2 scripts :)
let $delta1 :=
  <options xmlns="http://marklogic.com/appservices/infostudio">
    <uri>
      <literal>http://pubs/42scripts/</literal>
      <filename/>
      <literal>.</literal>
      <ext/>
    </uri>
  </options>

(: Define a delta to change the URI for the 4.1 scripts :)
let $delta2 :=
  <options xmlns="http://marklogic.com/appservices/infostudio">
    <uri>
      <literal>http://pubs/41scripts/</literal>
      <filename/>
      <literal>.</literal>
      <ext/>
    </uri>
  </options>
```

```
(: Load actions into the database :)
let $ticket1 := info:load(
  "C:\cvs\latest\myapp\scripts\actions",
  "mypolicy",
  (),
  "Sample-Database")

(: Load 4.2 scripts into the database :)
let $ticket2 := info:load(
  "C:\cvs\latest\myapp\scripts\4.2scripts",
  "mypolicy",
  $delta1,
  "Sample-Database")

(: Load 4.1 scripts into the database :)
let $ticket3 := info:load(
  "C:\cvs\latest\myapp\scripts\4.1scripts",
  "mypolicy",
  $delta2,
  "Sample-Database")

return (
  "Loaded files from",
  fn:data(info:ticket($ticket1)//directory,
  fn:data(info:ticket($ticket2)//directory,
  fn:data(info:ticket($ticket3)//directory )
```

## 6.0 Creating Custom Collectors and Transformers

Information Studio is shipped with built-in collectors and transformers. These collectors and transformers are implemented as plugins, which are located in the `/MarkLogic/Plugins` directory and described in “Collect” on page 23 and “Transform” on page 30.

Information Studio provides a plugin framework that enables you to create custom collectors and transformers to meet your own special needs. The main topics in this chapter are:

- [The infodev and plugin APIs](#)
- [Information Studio Plugin Framework](#)
- [Creating Custom Collectors](#)
- [Creating Custom Transformers](#)

### 6.1 The infodev and plugin APIs

The `infodev` and `plugin` APIs provide functions that allow you to create custom collector and transformer plugins. Both APIs are documented in detail in the *MarkLogic XQuery and XSLT Function Reference*.

### 6.2 Information Studio Plugin Framework

Information Studio provides a graphical interface for configuring and running collectors and transformers. The Plugin Framework described in this chapter provides a set of tools for creating your own custom plugins for use by Information Studio. The general Plugin Framework is described in the [Plugin Framework](#) chapter in the *Application Developer's Guide*.

After creating a plugin, simply move the file to the `/MarkLogic/Plugins` directory and it will be ready to use. Place any modules referenced by your plugin in the `MarkLogic/Modules` directory

**Warning** An error in a plugin will disable all App Servers, including the Admin interface. Always test your plugins on a non-production server before installing them on a production server.

Every collector or transformer plugin contains a map of function pointers to capabilities. The plugin module implements the functions defined in the capabilities map and registers them by calling the `plugin:register` function. The available collector capabilities are described in “Collector Capabilities and Function Signatures” on page 61. The available transformer capabilities are described in “Transformer Capabilities and Function Signatures” on page 73.

A function pointer takes the form:

```
xdmp:function(xs:QName("prefix:functionName"))
```

A collector capability takes the form:

```
"http://marklogic.com/appservices/infostudio/collector/capability"
```

A transformer capability takes the form:

```
"http://marklogic.com/appservices/infostudio/transformer/capability"
```

For example, the plugin portion of a very basic collector, named `mycollector`, might look something like the following:

```
declare namespace my =
  "http://marklogic.com/extension/plugin/mycollector";

declare function my:capabilities()
as map:map
{
  let $map := map:map()

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/infostudio/collector/model",
    xdmp:function(xs:QName("my:model")) )

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/infostudio/collector/start",
    xdmp:function(xs:QName("my:start")) )

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/string",
    xdmp:function(xs:QName("my:string")) )

  return $map
};

(: ... implement the functions in the capability map here ... :)
```

Use the `plugin:register` function to register the capabilities map for use by Information Studio:

```
plugin:register(my:capabilities(), "mycollector.xqy")
```

## 6.3 Creating Custom Collectors

This section describes how to create custom collectors. For example, you might want to write a collector that unzips zip files and ingests the contents into the database or a collector that ingests RSS feeds. Another type of collector might recognize a particular user and set the ingestion policy specifically for that user or extract files from one database and ingest them into another database.

The main topics in this section are:

- [Types of Collectors](#)
- [Collector Capabilities and Function Signatures](#)
- [Collector Interaction with Information Studio](#)
- [Collector Type and MarkLogic Server Restart](#)
- [An Example Collector](#)

### 6.3.1 Types of Collectors

There are two basic types of collectors:

- One-shot
- Long-running

An example of a one-shot collector is the Filesystem Directory collector, which is started, completes a specific ingestion operation, and then automatically stops. An example of a long-running collector is the Browser Drop-Box collector, which once started, continues to “listen” for input until it is explicitly stopped by a user.

The collector type impacts how you implement the collector, as described in “Collector Interaction with Information Studio” on page 65, as well as how the collector behaves in the event of a MarkLogic Server restart, as described in “Collector Type and MarkLogic Server Restart” on page 67.

### 6.3.2 Collector Capabilities and Function Signatures

At a minimum, a collector must do the following:

- Define a data model that specifies the data to be passed into the start function.
- Define a start function that initiates the load.
- Define a string function that specifies all of the labels needed for display.
- Call the `infodev:ingest` function to ingest the files into the database.

If the collector is long-running it must also define a function that returns a `plugin:listener-view` element.

The following table describes all of the available collector capabilities and the function signatures used by plugins to implement the capabilities.

Capability	Description
model	<p>The model for the data to be passed into the <code>plugin:start</code> function. Currently, model is not used by Information Studio. However future versions will require this capability, so it is recommended that you implement a model to ensure forward compatibility of your plugin.</p> <p>Function signature:</p> <pre>plugin:model (   ) as element (plugin:plugin-model)</pre>
start	<p>Starts the plugin.</p> <p>Function signature:</p> <pre>plugin:start (   \$model as element(),   \$ticket-id as xs:string,   \$policy-deltas as element(info:options)? ) as empty-sequence()</pre>
config-view	<p>The view to display the collector configuration window in the Information Studio Interface. The strings displayed are defined in the <code>plugin:string</code> function.</p> <p>Note that the <code>\$model</code> parameter is optional, so the plugin needs to either return a view for a user-defined model passed from the Information Studio Interface to the <code>plugin:config-view</code> function or return a view for the generic model defined in the <code>plugin:model</code> function, when no model is passed in from the Information Studio Interface.</p> <p>Function signature:</p> <pre>plugin:config-view (   \$model as element (plugin:plugin-model)?,   \$lang as xs:string,   \$submit-here as xs:string ) as element (plugin:config-view)</pre>

Capability	Description
listener-view	<p>The listener view for long-running collectors.</p> <p>Function signature:</p> <pre> plugin:listener-view(   \$upload-here as xs:string ) as element(plugin:listener-view) </pre>
handle-post	<p>Handles the ingestion of newly posted documents into the <code>listener-view</code> of a long-running collector.</p> <p>Function signature:</p> <pre> plugin:handle-post(   \$document as node(),   \$source-location as xs:string,   \$tid as xs:string?,   \$policy-deltas as element(info:options)?,   \$properties as element()* ) as xs:string+ </pre>
cancel	<p>Sets the ticket status to ‘cancelled’.</p> <p>Function signature:</p> <pre> plugin:cancel(   \$ticket-id as xs:string ) as empty-sequence() </pre>
complete	<p>Sets the ticket status to ‘completed’.</p> <p>Function signature:</p> <pre> plugin:complete(   \$ticket-id as xs:string ) as empty-sequence() </pre>
abort	<p>Sets the ticket status to ‘aborted’.</p> <p>Function signature:</p> <pre> plugin:abort(   \$ticket-id as xs:string ) as empty-sequence() </pre>

Capability	Description
validate	<p>Validates the data in the model.</p> <p>Function signature:</p> <pre>plugin:validate(   \$model as element(plugin:plugin-model) ) as element(plugin:report)*</pre>
string	<p>Defines the labels displayed by the collector in the Information Studio Interface.</p> <p>Function signature:</p> <pre>plugin:string(   \$key as xs:string,   \$model as element(plugin:plugin-model)?,   \$lang as xs:string) as xs:string?</pre>



### 6.3.3 Collector Interaction with Information Studio

The interaction between Information Studio and a collector is different depending on whether the collector is one-shot or long-running. The following sections describe the interaction between Information Studio and each type of collector.

#### 6.3.3.1 One-Shot Collectors

The interaction between Information Studio and a one-shot collector is as follows:

1. When the user creates a flow, a new flow document is created in the App-Services database.
2. When the user selects a collector for the flow in the Information Studio Interface, Information Studio renders the collector's view defined in the `plugin:config-view` function as an HTML iframe in the Information Studio Interface.
3. The iframe describes the elements to be defined in the data model. When the user finishes filling in the iframe fields and clicks the Done button, the flow stored in the App-Services database is updated with the completed data model.
4. Information Studio displays the completed collector configuration in the Information Studio Interface using the labels in the `plugin:string` function.
5. The collector is in the quiescent state until the user clicks Start Loading in the Information Studio Interface. At this point, Information Studio asks for the listener-view capability in the plugin. In this case, the collector does not have a listener-view capability, so it is identified as one-shot in the ticket.
6. Information Studio updates the ticket in the App-Services database with the collector type and start time and sets its status to 'active.'
7. The `plugin:start` function reads the ticket, data model, and any policy deltas from the flow stored in the App-Services database and calls the `infodev:ingest` function for each document to ingest into the database.
8. When Information Studio has finished ingesting the documents, it updates the ticket with the 'completed' status.

### 6.3.3.2 Long-Running Collectors

The interaction between Information Studio and a long-running collector is as follows:

1. When the user creates a flow, a new flow document is created in the App-Services database.
2. When the user selects a collector for the flow in the Information Studio Interface, Information Studio displays the collector configuration in the Information Studio Interface using the labels in the `plugin:string` function.
3. The collector is in the quiescent state until the user clicks Start Loading in the Information Studio Interface. At that point, Information Studio asks for the listener-view capability in the plugin. In this case, the collector has a listener-view capability, so Information Studio creates a ticket that identifies the plugin as long-running.
4. Information Studio updates the ticket in the App-Services database with the collector type and start time and sets its status to 'active.'
5. Information Studio activates the listener-view defined in the `plugin:listener-view` function in the Information Studio Interface.

**Note:** The Browser Drop-Box collector implements the listener-view as an UploadApplet, but anything that can post multi-part can be used to implement the listener-view.

6. From this point, as long as the ticket status is 'active', any documents put into listener are passed to the `plugin:handle-post` function. If multiple documents are put into the listener, the listener ensures that one document at a time is passed to the `plugin:handle-post` function, which calls the `infodev:ingest` function to ingest the document into the database.
7. The listener-view stays active until the user clicks the Stop Loading, at which time Information Studio terminates the listener-view and sets the ticket status to 'completed.'

### 6.3.4 Collector Type and MarkLogic Server Restart

There are special considerations when designing a collector that impact its behavior in the event that MarkLogic Server shuts down and restarts before a collector has completed its ingestion operation.

Whether or not a collector is long-running has implications should MarkLogic Server restart before the collector has completed its ingestion operation. When the collector plugin is called by the user, a ticket is created containing the server start time and an annotation that notes whether the collector is long-running or not long-running. Should MarkLogic Server restart before a collection operation has completed, a trigger on the Database Online event for the App-Services database will check all active tickets. If a ticket is annotated as long-running, no action will be taken, so the ticket remains active and the ingestion operation is resumed. If a ticket is not annotated as long-running and if the server start time is later than the start time recorded in the ticket, then the ticket status is set to 'aborted'.

### 6.3.5 An Example Collector

This section walks through a simple custom collector that inserts a single, fixed document that is specified in the plugin. The collector loads a “document” with the following content into a database and URI specified by the user:

```
<root attribute="value">
  <child>content</child>
  <namespace xmlns="http://marklogic.com">content</namespace>
</root>
```

The code for the collector is as follows:

```
xquery version "1.0-ml";

(: Copyright 2002-2010 MarkLogic Corporation. All Rights Reserved. :)

(: Declare and import namespaces :)

declare namespace testdoc =
  "http://marklogic.com/extension/plugin/testdoc";

(: Declare the namespaces for the Application Services label and tag
   libraries. :)

declare namespace lbl="http://marklogic.com/xqutils/labels";
declare namespace ml="http://marklogic.com/appservices/mlogic";

import module namespace plugin =
  "http://marklogic.com/extension/plugin"
  at "/MarkLogic/plugin/plugin.xqy";

import module namespace infodev=
  "http://marklogic.com/appservices/infostudio/dev"
  at "/MarkLogic/appservices/infostudio/infodev.xqy";

import module namespace info =
  "http://marklogic.com/appservices/infostudio"
  at "/MarkLogic/appservices/infostudio/info.xqy";

declare default function namespace
  "http://www.w3.org/2005/xpath-functions";
```

```
(: Define the plugin capabilities map :)

declare function testdoc:capabilities()
as map:map
{
  let $map := map:map()

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/infostudio/collector/model",
    xdmp:function(xs:QName("testdoc:model")))

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/infostudio/collector/start",
    xdmp:function(xs:QName("testdoc:start")))

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/infostudio/collector/config-
view",
    xdmp:function(xs:QName("testdoc:view")))

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/infostudio/collector/cancel",
    xdmp:function(xs:QName("testdoc:cancel")))

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/infostudio/collector/abort",
    xdmp:function(xs:QName("testdoc:abort")))

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/infostudio/collector/validate",
    xdmp:function(xs:QName("testdoc:validate")))

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/string",
    xdmp:function(xs:QName("testdoc:string")))

  return $map
};
```

```
(:~ Implement the data model to be used by config-view in the
Information Studio Interface. :)

declare function testdoc:model(
) as element(plugin:plugin-model)
{
  <plugin:plugin-model>
    <plugin:data>
      <path>/test/document.xml</path>
    </plugin:data>
  </plugin:plugin-model>
};

(:~ Implement the start function that starts the plugin. All
collector start functions accept the same parameters: the
plugin model, the ticket ID, and optional policy deltas. :)

declare function testdoc:start(
  $model as element(plugin:plugin-model),
  $tid as xs:string,
  $policy-deltas as element(info:options)?
) as empty-sequence()
{

  (: The "document" to be ingested :)

  let $doc :=
    <root attribute="value">
      <child>content</child>
      <namespace xmlns="http://marklogic.com">content</namespace>
    </root>

  let $path := $model/plugin:data/path
  let $_ := infodev:ticket-set-total-documents($tid, 1)

  (: Ingest the document into the database and log the ingest event
to the ticket's progress file in the App-Services database. :)

  let $ingestion :=
    try {
      infodev:ingest($doc, $path, $tid, $policy-deltas),
      infodev:log-progress(
        $tid,
        <info:annotation>test doc inserted</info:annotation>,
        1)
    } catch($e) {
      infodev:handle-error($tid, $path, $e)
    }

  (: When ingestion has completed, reset the ticket status. :)

  let $_ := infodev:ticket-set-status($tid, "completed")
  return ()
};
```

(:~ Implement the view function to display the collector popup configuration window in the Information Studio Interface. The strings displayed are defined in the testdoc:string function. The "Done" button comes from the Application Services mlogic tag library.

Note the value of the input type, "{\$model/plugin:data/\*:path}". This is because path is not in the default namespace. :)

```
declare function testdoc:view(
  $model as element(plugin:plugin-model)?,
  $lang as xs:string,
  $submit-here as xs:string
) as element(plugin:config-view)
{
  <config-view xmlns="http://marklogic.com/extension/plugin">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
        <title>iframe plugin configuration</title>
      </head>
      <body>
        <form action="{$submit-here}" method="post">
          <label for="path">
            {testdoc:string("path-label", $model, $lang)}
          </label>
          <input type="text" name="path" id="path"
            value="{$model/plugin:data/*:path}"/>
          <br/>
          <ml:submit label="Done"/>
        </form>
      </body>
    </html>
  </config-view>
};
```

(:~ Implement a function to cancel an active ticket :)

```
declare function testdoc:cancel(
  $ticket-id as xs:string
) as empty-sequence()
{
  infodev:ticket-set-status($ticket-id, "cancelled")
};
```

(:~ Implement a function to abort an active ticket :)

```
declare function testdoc:abort(
  $ticket-id as xs:string
) as empty-sequence()
{
  infodev:ticket-set-status($ticket-id, "aborted")
};
```

```

(:~ Implement a function to validate the plugin model. Return an
empty sequence if a path is specified in the plugin model.
Report an error if the path is empty. :)

declare function testdoc:validate(
  $model as element()
) as element(plugin:report)*
{
  if (normalize-space(string($model/plugin:data/path)) eq "")
  then
    <plugin:report id="db">
      Specified path must not be empty
    </plugin:report>
  else ()
};

(:~ Implement a function to define all of the labels displayed by
the collector. This plugin uses the labels defined in the
Application Services label library. :)

declare function testdoc:string(
  $key as xs:string,
  $model as element(plugin:plugin-model)?,
  $lang as xs:string
) as xs:string?
{
  let $labels :=
    <lbl:labels xmlns:lbl="http://marklogic.com/xqutils/labels">
      <lbl:label key="name">
        <lbl:value xml:lang="en">
          Load an individual document for testing purposes
        </lbl:value>
      </lbl:label>
      <lbl:label key="description">
        <lbl:value xml:lang="en">
          Insert a fixed document at {$model/plugin:data/path}
        </lbl:value>
      </lbl:label>
      <lbl:label key="path-label">
        <lbl:value xml:lang="en">Path:</lbl:value>
      </lbl:label>
    </lbl:labels>
  return $labels/lbl:label[@key eq $key]/lbl:value[@xml:lang eq
$lang]/string()
};

(:~ Register the plugin with the capabilities map and identify the
plugin as "collector-testdoc.xqy". :)

plugin:register(testdoc:capabilities(), "collector-testdoc.xqy")

```



## 6.4 Creating Custom Transformers

This section describes how to create custom transformers. Transformers utilize MarkLogic Server Content Processing Framework (CPF) to modify one document at a time as it is loaded into the database. Information Studio automatically configures CPF domains, pipelines, and triggers on the Fab database for each transformer. A transformation process must be linear, so branching and conditional operations are not possible.

The main topics in this section are:

- [Transformer Capabilities and Function Signatures](#)
- [Transformer Interaction with Information Studio](#)
- [An Example Transformer](#)

### 6.4.1 Transformer Capabilities and Function Signatures

All transformers must do the following:

- Define a data model that specifies the data to be passed to the compile function.
- Define a configuration view to display the transformer configuration window in the UI.
- Define a compile function that modifies the data in the model.
- Define a string function that specifies all of the labels needed for display.

The following table describes all of the available transformer capabilities and the function signatures used by plugins to implement the capabilities.

Capability	Description
model	<p>The data model for the UI. This represents the data to be passed into the <code>plugin:compile</code> function. Currently, model is not used by Information Studio. However future versions will require this capability, so it is recommended that you implement a model to ensure forward compatibility of your plugin.</p> <p>Function signature:</p> <pre>plugin:model(   ) as element(plugin:plugin-model)</pre>

Capability	Description
config-view	<p>The view to display the transformer configuration window in the UI. The strings displayed are defined in the <code>plugin:string</code> function.</p> <p>Note that the <code>\$model</code> parameter is optional, so the plugin needs to either return a view for a user-defined model passed from the UI to the <code>plugin:config-view</code> function or return a view for the generic model defined in the <code>plugin:model</code> function, when no model is passed in from the UI.</p> <p>Function signature:</p> <pre> plugin:config-view(   \$model as element(plugin:plugin-model)?,   \$lang as xs:string,   \$submit-here as xs:string ) as element(plugin:config-view) </pre>
compile	<p>Transforms the document according to the data model.</p> <p>Function signature:</p> <pre> plugin:compile(   \$model as element() ) as element(plugin:compile) </pre>
string	<p>Defines the labels displayed by the transformer in the UI.</p> <p>Function signature:</p> <pre> plugin:string(   \$key as xs:string,   \$model as element(plugin:plugin-model)?,   \$lang as xs:string ) as xs:string? </pre>

### 6.4.2 Transformer Interaction with Information Studio

The interaction between Information Studio and a transformer is as follows:

1. When the user creates a flow, a new flow document is created in the App-Services database.
2. When the user selects a transformer for the flow in the UI, Information Studio renders the transformer's view defined in the `plugin:config-view` function as an HTML iframe in the UI.
3. The iframe describes the elements to be defined in the transformer's data model. When the user finishes filling in the iframe fields and clicks the Done button, the `plugin:compile` function in the transformer updates the flow stored in the App-Services database with the completed data model.
4. Information Studio displays the completed transform step in the UI using the labels in the `plugin:string` function.
5. From this point, each document ingested by the flow is modified against the transformer's data model.

**Note:** Transform steps implemented using XSLT do not work on binary documents. Binary documents will pass through XSLT steps unchanged.

### 6.4.3 An Example Transformer

This section walks through the implementation of the Schema Validation transformer described in “Validate Documents against Schema” on page 37 that allows users to set the level at which the XML of loaded documents are to be validated against the schema.

```
xquery version "1.0-ml";

(: Copyright 2002-2010 MarkLogic Corporation. All Rights Reserved. :)

(: Declare and import namespaces :)

declare namespace transform =
  "http://marklogic.com/extension/plugin/transform";

declare namespace info="http://marklogic.com/appservices/infostudio";

import module namespace plugin =
  "http://marklogic.com/extension/plugin"
  at "/MarkLogic/plugin/plugin.xqy";

import module namespace infodev =
  "http://marklogic.com/appservices/infostudio/dev"
  at "/MarkLogic/appservices/infostudio/infodev.xqy";

import module namespace pipe =
  "http://marklogic.com/appservices/pipeline"
  at "/MarkLogic/appservices/infostudio/pipe.xqy";

(: Declare the namespaces for the Application Services label and tag
   libraries. :)

declare namespace lbl="http://marklogic.com/xqutils/labels";
declare namespace ml="http://marklogic.com/appservices/mlogic";

declare namespace xproc="http://www.w3.org/ns/xproc";
declare namespace html="http://www.w3.org/1999/xhtml";
declare namespace xsl = "http://www.w3.org/1999/XSL/Transform";

declare default function namespace "http://www.w3.org/2005/xpath-
functions";
```

```
(: Define the plugin capabilities map :)

declare function transform:capabilities-validate-with-xml-schema()
as map:map
{
  let $map := map:map()
  let $_ := map:put($map, "http://marklogic.com/appservices/string",
xdmp:function(xs:QName("transform:string-validate-with-xml-schema")))

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/infostudio/transformer/config-
view",
    xdmp:function(xs:QName("transform:schema-validate-with-xml-
schema")))

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/infostudio/transformer/model",
    xdmp:function(xs:QName("transform:model-validate-with-xml-
schema")))

  let $_ := map:put(
    $map,
    "http://marklogic.com/appservices/infostudio/transformer/compile",
    xdmp:function(xs:QName("transform:compile-step-validate")))

  return $map
};

(:~ Implement the data model to be used by config-view in the UI. :)

declare function transform:model-validate-with-xml-schema()
as element(plugin:plugin-model)
{
  <plugin:plugin-model>
    <plugin:data>
      <name/>
      <mode>strict</mode>
    </plugin:data>
  </plugin:plugin-model>
};
```

(:~ Implement the view function to display the transformer popup configuration window in the UI. The data model defined in the transform:model-validate-with-xml-schema function is used to display the default setting of 'strict'. The strings displayed are defined in the transform:string-validate-with-xml-schema function. The "Done" button comes from the Application Services mlogic tag library. :)

```
declare function transform:schema-validate-with-xml-schema(
  $model as element(plugin:plugin-model)?,
  $lang as xs:string,
  $submit-here as xs:string
) as element(plugin:config-view)
{
  let $m := ($model, transform:model-validate-with-xml-schema())[1]
  let $is-strict := $m/plugin:data/mode eq "strict"
  return
    <config-view xmlns="http://marklogic.com/extension/plugin">
      <html xmlns="http://www.w3.org/1999/xhtml">
        <link href="/infostudio/static/css/transform-plugin.css"
          type="text/css" rel="stylesheet"/>
        <body>
          <h3>Schema Validation Transform</h3>
          <form action="{ $submit-here }" method="post">
            <div class="name">
              <label for="name">
                Transform name:
                <input name="name" id="name"
                  value="{ $m/plugin:data/*:name }"/>
              </label>
            </div>
            <hr/>
            <p>Validation mode:
              <select name="mode">
                <option>
                  {if ($is-strict)
                    then attribute selected {"selected"}
                    else ()}strict
                </option>
                <option>{
                  if ($is-strict)
                    then ()
                    else attribute selected {"selected"}}lax
                </option>
              </select>
            </p>
            <div id="submit">
              <ml:submit label="Done"/>
            </div>
```

```

        <div class="tips guide">
          <hr/>
          <p>
            <label>Validation mode:</label><br/>
            Validation mode specifies the initial validation mode.
          </p>
          <p>
            <label>Note:</label><br/>
            Validation is performed using schemas from the 'Schemas'
database.
          </p>
        </div>
      </form>
    </body>
  </html>
</config-view>
};

```

(::~ Implement a compile function to validate the document against the schema using the validation level specified in the data model. ::)

```

declare function transform:compile-step-validate(
  $model as element()
) as element(plugin:compile)
{
  let $xproc := <xproc:validate-with-xml-schema
    validation="{ $model/plugin:data/mode }"/>
  return
    <plugin:compile>
      <plugin:xslt>{pipe:compile-step($xproc)}</plugin:xslt>
    </plugin:compile>
};

```

(::~ Implement a function to define all of the labels displayed by the transformer. This plugin uses the labels defined in the Application Services label library. ::)

```

declare function transform:string-validate-with-xml-schema(
  $key as xs:string,
  $model as element(plugin:plugin-model)?,
  $lang as xs:string
) as xs:string?
{
  let $labels :=
    <lbl:labels xmlns:lbl="http://marklogic.com/xqutils/labels">
      <lbl:label key="name">
        <lbl:value xml:lang="en">Schema Validation</lbl:value>
      </lbl:label>
      <lbl:label key="description">
        <lbl:value xml:lang="en">{
          if($model)
          then $model/plugin:data/*:name/string()
          else "Validate a document against existing schemas." }
        </lbl:value>
      </lbl:label>
    </lbl:labels>

```

```
</lbl:label>
<lbl:label key="validation">
  <lbl:value xml:lang="en">Validation Type</lbl:value>
</lbl:label>
<lbl:label key="validation-strict">
  <lbl:value xml:lang="en">Strict</lbl:value>
</lbl:label>
<lbl:label key="validation-lax">
  <lbl:value xml:lang="en">Lax</lbl:value>
</lbl:label>
</lbl:labels>

return $labels/lbl:label[@key eq $key]/lbl:value[@xml:lang eq
$lang]/string()
};

(:~ Register the plugin with the capabilities map and identify the
plugin as "transform-validate-with-xml-schema.xqy". :)

plugin:register(
  transform:capabilities-validate-with-xml-schema(),
  "transform-validate-with-xml-schema.xqy")
```



## 7.0 Technical Support

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

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

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

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

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

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