Custom Content Access Module

Plug-in Configuration and Development Guide
Version 2.1.1

MasterObjects

Legal Notices

MasterObjects, Inc.
1156 Clement Street
San Francisco, CA 94118

info@masterobjects.com
http://www.masterobjects.com

Table of Contents

[1] Introduction

1.1 About this Document

This document contains developer documentation for the Custom Content Access Module (Custom CAM) with Custom Content Queries. The Custom CAM is part of the QuestFields software development kit (SDK).

Intended Audience

This document is intended for use by software engineers with a basic understanding of the Java programming language. It describes the open API provided as part of the QuestFields SDK, making it easy to interface the QuestFields Server with any third-party content source.

Related Documents

For information about the QuestFields Server, please refer to the *QuestFields Server Administration Guide*.

For information about the QuestFields Client, please refer to the *QuestFields Client Administration Guide*.

1.2 Custom Components Overview

Custom Content Access Module

A *Content Access Module* (CAM) provides a standardized mechanism to connect the QuestFields system with a content source. A CAM is the “middleware” between the QuestFields Server and the content sources it works with. Different CAMs are needed to communicate with various content source types, such as JDBC-compliant SQL databases (through the JDBC CAM), LDAP-compliant directory servers (through the LDAP CAM), or flat files (through the QuestFields Indexer option).

The *Custom Content Access Module* (Custom CAM) provides the means to create a connection between the QuestFields Server and *any* protocol or file. A Custom CAM is created in the Java programming language. The Custom CAM *API* uses simple predefined Java classes to connect to your proprietary protocol, web service, or files.

Server Plug-in

The resulting Custom CAM is compiled as a "jar file" and is placed into the plug-ins directory of the QuestFields Server. A Custom CAM implementation is therefore considered a "server plug-in". Using this guide and the files provided with the SDK, MasterObjects partners and customers can easily develop and maintain their own QuestFields Server plug-ins.

⇒ *Please note: Deployment of custom CAMs requires an addition to QuestFields Server license.*

Custom Content Query

A Custom Content Query holds the configuration for a predefined type of query to be executed by the Custom CAM. When you develop a CAM, you can make it work for any number of query definitions.

1.3 Version History

This chapter lists the changes for every release of this document.

1.0.0
- Covers the first release of the Java Content Access Module.

1.0.0L
- Changed into U.S. Letter page layout.

1.0.1
- Added collator-additional-rule configuration item.

2.0.0
- Renamed to "Custom Content Access Module";
- Updated the document for QuestFields Server release 2.0.

2.1.0
- Updated the document for QuestFields Server release 2.1.

2.1.1
- Various textual improvements.

[2] CAM Instance Configuration

Factory

The Custom CAM uses the "factory" design pattern to create connections to the proprietary protocol or files. This means that the QuestFields Server asks a custom factory object for an instance of a connection, which is subsequently used to execute QuestFields queries and return results to the QuestFields Server in a simple generic format.

A Custom CAM is configured using XML files in the "QuestFields Home" directory, just like the JDBC and LDAP CAMs that are bundled with the QuestFields Server. So you can make your Custom CAM configurable so it works with multiple sets of data or tables (as long as they can be queried using the protocol for which you develop the Custom CAM).

A Custom CAM instance configuration contains the location of the plug-in directory, custom factory-specific configuration parameters to create and use for creating the connections, and a Boolean value that indicates whether the custom results are pre-sorted or not (determining whether or not the QuestFields Server will need to perform another sort as configured in the QuestFields content channel).

The CAM instance configuration is used to configure the connection to the proprietary protocol or files. The actual query is defined in the content query configuration (see section [3]).

2.1 Settings Reference

The Custom Content Access Module instance is configured using the configuration file named `{CAM-Id}.xml`, located in the QuestFields CAM configuration directory, `QO_HOME/conf/cams`. The configuration file is in XML format and is encoded in the UTF-8 character encoding. The settings file can contain comments in standard XML format.

The Id of a Content Access Module instance, used internally in the QuestFields system to identify the CAM instance, is the name of the CAM's configuration file, without the `.xml` suffix.

The file starts with `<customCam id="{CAM-Id}">` and ends with `</customCam>`. The elements contained in the configuration file are described below.

2.1.1 General Custom CAM settings

pluginDirectory

Use: An optional path to the Custom CAM's directory. If not specified, *QO_HOME*/plugins is assumed. You may either specify the path relative to the default plugins directory or the full path.

All jar files in the Custom CAM's directory will end up on the class path of the Custom CAM.

Value type: String.

2.1.2 Custom CAM factory settings

factoryClassName

Use: Name of the factory class (including the package name), e.g. com.your_domain.questfields.cam.specialsearch.Factory

Value type: String.

factoryProperties

Use: An optional list of factory properties.

Value type: List of <entry> elements.

entry

Use: One factory property.

Value type: Contains <key> and <value> elements.

key

Use: Name of factory property.

Value type: String.

value

Use: Value of factory property.

Value type: String.

[3] Content Query Configuration

A Custom CAM needs at least one Custom Content Query configuration, which is used to execute queries once the Custom CAM instance is loaded by the QuestFields Server. The Custom Content Query configuration holds predefined information about how to query the proprietary protocol or files.

Each Custom Content Query configuration holds CAM-specific "name value pair" properties. These properties together form a predefined query (a method of querying) that is called (executed) by the QuestFields Server through the Custom CAM.

3.1 Settings Reference

A Custom Content Query is configured using the configuration file named *{custom_content_query_id}*.xml, located in the QuestFields Content Query configuration directory, *QO_HOME*/conf/content-queries. The configuration file is in XML format and is encoded in the UTF-8 character encoding. The settings file can contain comments in standard XML format.

The id of a channel, used internally in the QuestFields system to identify the channel, is the filename of the channel's configuration file without the .xml suffix.

The file starts with <customContentQuery id="*{custom_content_query_id}*"> and ends with </customContentQuery>. The elements contained in the configuration file are described below.

3.1.1 General Custom Content Query settings

camId

Use: Id of the Custom Content Access Module instance used by the Content Query (see section [2]).

Value type: String.

3.1.2 Custom Content Query property settings

contentQueryProperties

Use: An optional list of custom factory properties.

Value type: List of <entry> elements.

entry	*Use:*	For each custom CAM factory property.
	Value type:	Contains `<key>` and `<value>` elements.
key	*Use:*	Name of a custom property.
	Value type:	String.
value	*Use:*	Value of a custom property.
	Value type:	String.

[4] Developing a Custom CAM Plug-In

A Custom CAM plug-in is a QuestFields Server Content Access Module (CAM) that is customized to connect to a proprietary protocol or file. A Custom CAM plug-in is programmed in the Java programming language.

The design of the plug-in conforms to the Factory pattern as described in "Design Patterns" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (also called the "Gang of Four" book, ISBN 0-201-63361-2).

To create the plug-in, two Java interfaces must be implemented. One interface defines the *query executor factory* class and the other interface defines the *query executor* class. Various additional *helper classes* are available. For example, one is used to hold an array of *Results* and another is used for *Exceptions* that might occur in the plug-in.

4.1 Designing the Custom CAM Plug-In

The first step is to separate the query from the protocol.

- The *protocol* is defined by the combination of syntax and semantics describing a certain type of queries (e.g. `SELECT` or `INSERT` statements in SQL) and the way those queries are transported to the backend system (e.g. the address of the server to be queried and the third-party libraries to be linked to).

  In the context of a Custom Plug-in, a *connection* is regarded to be an instance of the protocol. Protocol configuration data can be defined in the CAM configuration file as name-value pairs (see chapter [2]).

- The *query* is the actual "question" (for example, `SELECT * FROM MYTABLE1`). Query configuration data can be defined in the Content Query configuration file as name-value pairs (see chapter [3]).

The query itself should be divided into a *static* and a *dynamic* part:

- The *static* part is the non changing part of the queries that need to be executed (for example `SELECT ? FROM MYTABLE,` where `?` is to be filled in later). This part is configured in the Content Query configuration file.

- The *dynamic* part is different for each query (above, it is the `?` that is filled in for each query). This is what the user *types* into the QuestField or what is passed to the QuestFields Server as an additional *qualifier*.

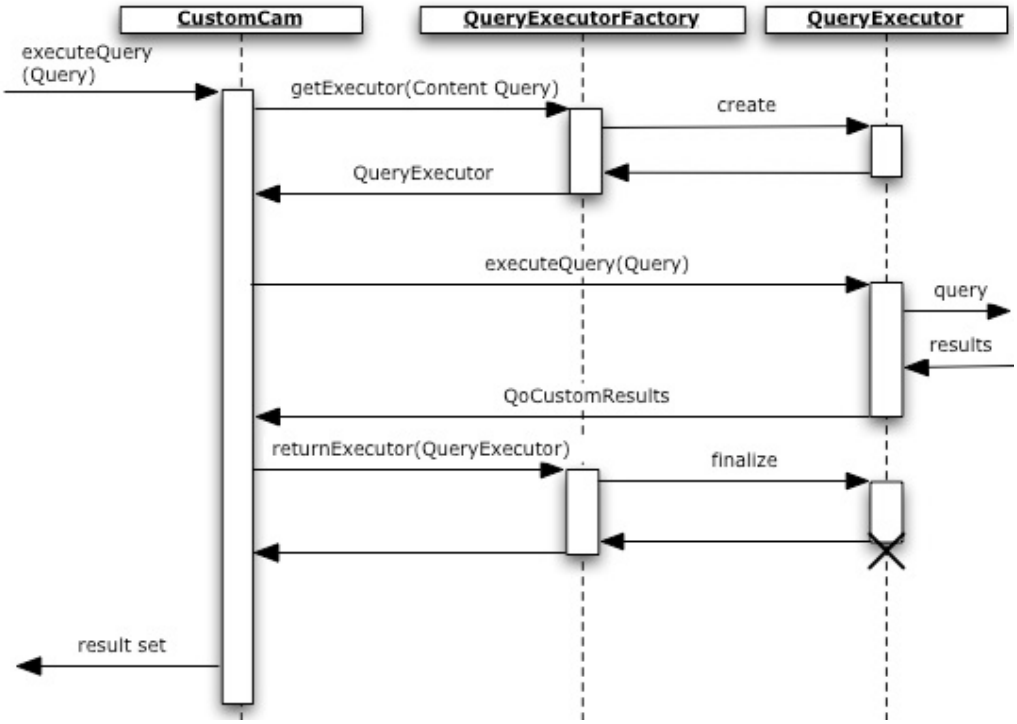
If the static part of the query is too complex for it to be defined by name-value pairs, another option is to create different methods or even classes and to use a name-value pair as a switch to determine which method (or which class) should be used.

So how do these 3 parts relate to the 2 interfaces that need to be implemented?

The `QueryExecutorFactory` configures the protocol and provides `QueryExecutor` instances by request. Before creating a `QueryExecutor`, the factory ensures that the `QueryExecutor` has a *connection* and a valid *static* part of the query.

The `QueryExecutor` instance receives the *dynamic* part of the query and executes the query by feeding it to the connection. After the `QueryExecutor` has handed the results over to the QuestFields Server, it is returned to the `QueryExecutorFactory` so the factory can handle the destruction of the `QueryExecutor`.

⇒ *Note: instead of creating and destroying `QueryExecutor` instances, it is possible to maintain a pool of them. Explanation on how to create and maintain a "connection pool" is beyond the scope of this document. For assistance, please contact MasterObjects.*



Custom CAM query execution overview

4.2 Developer Setup

To develop the Custom CAM plug-in you need the following jar files available during compilation:

log4J-1.2.8.jar,
questobjects-base.jar,
questobjects-service.jar.

These jar files are included in the lib directory of QuestFields Software Development Kit (SDK) on the QuestFields CD.

⇒ *Do not package these jar files into your plug-in because they are available to the QuestFields Server already.*

The com.masterobjects.qo.external.customplugin package, which contains the interfaces and helper classes of the Custom CAM, is contained in questobjects-service.jar. The other jar files mentioned above are needed by classes in questobjects-service.jar, but they are not called directly by the custom plug-in.

After compilation, package the plug-in classes into a new jar file. Then, place the jar file and its dependencies into the configured plug-in directory and restart the QuestFields Server (see chapter 2.1.1).

4.3 Implementation

Classes and interfaces that are needed to create the plug-in are available in the com.masterobjects.qo.external.customplugin package, and are described by the javadocs included with the SDK. The logging is done through the Logger class and is a part of Log4J.

4.3.1 Helper classes

Helper class names start with QoCustom, with the exception of the Logger.

Logger

Use: The Logger is a *Log4J* logging class and should be used to log messages. For more information on the Logger class please review the JavaDoc of Log4J.

QoCustomUtils

Use: This class provides utility methods that can be used by the Custom CAM plug-in. For descriptions, see the JavaDocs provided with the SDK.

QoCustomException

Use: This class must be used to wrap all *Exceptions* that can occur in the factory and in the QueryExecutor.

QoCustomResults

Use: This class must be used to build the result set, which will be used by the server to show the results to the user. Instantiate this class and use the addResult method to add each result.

QuestFields Results

A *result* returned by a Custom Access Module can consists of 4 parts:

- An optional *key*, which can be used to contain a compound value that needs to be returned by the HTML form into which the QuestField is placed (for example "JOHN_SMITH_DEP5_CA"). This property may be null.

- A *value*, which is the result that is displayed to the user and that is used to correct the input of the user (for example "`John Smith`"). This property is mandatory.

- Optional *metadata*, which is an array of Strings. This is typically shown in the pop-up list of the QuestField as extra information (for example `{"DEP5";"CA"}`). This property may be `null`.

- An optional *type*, as defined in class `QoCustomResultType`. This property defaults to `0` for "normal" results.

4.3.2 QueryExecutorFactory Interface

The `QueryExecutorFactory` interface defines the methods that must be implemented in a factory class so it can be used by the Custom CAM.

The Custom CAM uses the factory to get `QueryExecutor` instances, which it uses to execute queries. This is an implementation of the Factory pattern in Design Patterns (the GoF book). The factory must have a default constructor (a constructor without arguments). Initialization based on configuration information in the CAM configuration file must be done in the `init` method.

The order of calls is:

At startup:
1) `Constructor` 2) `init` 3) `getKeys` 4) `testConnection`

When querying (which can happen by multiple threads at the same time!):
1) `getExecutor` 2) `returnExecutor`

At shutdown:
1) `shutdown`

At any time after calling the constructor:
- `getBuildNumber, getName, getManufacturer`

init

The init method is called to initialize the Factory.

Passed parameters:

- `factoryProperties` contains properties that were read from the CAM configuration file, plus standard properties defined by the keys in class `QoStandardContentAccessModulePropertyKeys`. These standard properties include the full path of the actual plug-in directory used (derived from the –possibly relative or undefined-- `pluginDirectory` property of the CAM configuration).

- `Logger` is a *Log4J Logger instance* that can be used to log messages to the log files.

Return value: `void`

Exceptions thrown:

- `QoCustomException` should wrap any exception that occurs within the `init` method.

getKeys

The `testConnection` method tests whether the connection is available. If this is not the case the Java CAM will not accept queries.

Passed parameters:

- `contentQueryProperties` are properties that are defined in the Content Query configuration file, plus standard properties defined by the keys in class `QoStandardContentQueryPropertyKeys`. These standard properties include the Id of the content query.

Return value: An array of String, containing the keys that need to be filled in by the Channel when executing a Custom Content Query, and which are the keys available in the query `HashMap` in `QueryExecuter.executeQuery` for the `QueryExecuter` that was retrieved with the same `contentQueryProperties` in the `getExecutor`.

Exceptions thrown: None

testConnection

The `testConnection` method tests whether the connection is available. If this is not the case the Custom CAM will not accept queries.

Passed parameters: None

Return value: `void`

Exceptions thrown:

- `QoCustomException` should wrap any exception that occurs within the `testConnection` method.

getExecutor

The `getExecutor` method returns a class that implements the `QueryExecutor` interface. It can use the supplied `contentQueryProperties` to initialize the `QueryExecutor` with the right properties.

⇒ *This method may be called by multiple threads at the same time. Make sure that all shared resources (like class/object attributes) are either locked or accessed in a synchronized manner to avoid deadlocking.*

Passed parameters:

- `contentQueryProperties` are properties that are defined in the Content Query configuration file.

- `maxResults` is the maximum number of results that should be returned. This is set in the channel configuration.

- `queryTimeOut` is the maximum time it should take to get a result. This is set in the channel configuration.

Return value: `QueryExecutor` is returned for the use by the Custom CAM.

Exceptions thrown:

- `QoCustomException` should wrap any exception that occurs within the `getExecutor` method.

returnExecutor

The `returnExecutor` method is called after a `QueryExecutor` is used. This enables the factory to pool or to safely dispose of the `QueryExecutor`.

⇒ *This method may be called by multiple threads at the same time. Make sure that all shared resources (like class/object attributes) are either locked or accessed in a synchronized manner to ensure that deadlocking won't occur.*

Passed parameters:

- `QueryExecutor` returned by the Custom CAM after use.

Return value: `void`

Exceptions thrown: None

shutdown

The `shutdown` method is called by the Custom CAM at shutdown (when the server is shut down, the Custom CAM is called so that it can clean up). This is used to safely close and shut down systems used by the factory.

Passed parameters: None

Return value: `void`

Exceptions thrown: None

getBuildNumber

The `getBuildNumber` method is called by the Custom CAM. This is used by the QuestFields Server to retrieve the build number of the Java plugin.

Passed parameters: None

Return value: A String containing the build number.

Exceptions thrown: None

getName

The `getName` method is called by the Custom CAM. This is used by the QuestFields Server to retrieve the name of the Java plugin.

Passed parameters: None

Return value: A String containing the name.

Exceptions thrown: None

getManufacturer

The `getManufacturer` method is called by the Custom CAM. This is used by the QuestFields Server to retrieve the name of the manufacturer of the Java plugin.

Passed parameters: None

Return value: A String containing the name of the manufacturer.

Exceptions thrown: None

4.3.3 QueryExecutor Interface

The `QueryExecutor` interface should be implemented by the class that actually executes the query.

executeQuery

The `executeQuery` method is called by the Java CAM to execute a query. This method should be implemented by the query executer instance that was returned by the `QueryExecutorFactory`.

Passed parameters:

- `queryItems` is a Map containing key value pairs. The keys are the keys as defined by the `QueryExecutorFactory.getKeys`.

Return value: `QoCustomResults` contains the results of the query. The `executeQuery` should also set the `setComplete` on the results if the results aren't complete (if there are more results than `maxResults`) and `setSorted` if the results are sorted.

⇒ *Note that if the results are sorted that they are sorted in the same way as defined in the channel configuration. If not, make sure that only one Content Query is called for every query that enters the Channel. Else the result might not be in the order that is expected. This will be fixed in a future version of the server.*

Exceptions thrown:

- `QoCustomException` should wrap any exception that occurs within the `executeQuery` method.

[5] Glossary

CAM — A Content Access Module (CAM) provides a standardized mechanism to link the QuestFields system to a Content Engine. A CAM is the "middleware" between the QuestFields system and the content sources it works with. Different Content Access Modules are needed to communicate with various content sources, such as JDBC-compliant SQL databases (through the JDBC CAM), LDAP-compliant directory servers (through the LDAP CAM), or flat files (through the QuestFields Indexer CAM).

Content Channel — One of the services configured on the QuestFields Server, linking a QuestField to one or more Content Queries and returning appropriate results from those Content Queries.

Content Query — A pre-configured query for a specific CAM. To be used by that CAM to return results (answers) to the QuestFields system.

JDBC — Java Database Connectivity (JDBC) is a standard application program interface specification for connecting programs to the data in popular databases.

LDAP — Lightweight Directory Access Protocol (LDAP) is a standard application program interface for connecting programs to the data in corporate directories.

QuestField — A user interface element that sends queries to, and receives results from the QuestFields Server.