Platform SDK Developer's Guide

Using the Configuration Object Model Application Block

4/15/2025

Using the Configuration Object Model Application Block

> **Important**
> This application block is a reusable production-quality component. It has been designed using industry best practices and provided with source code so it can be used "as is," extended, or tailored if you need to. Please see the License Agreement for details.

The Configuration Object Model Application Block provides developers with a consistent and intuitive object model for working with Configuration Server objects.

Java

Architecture and Design

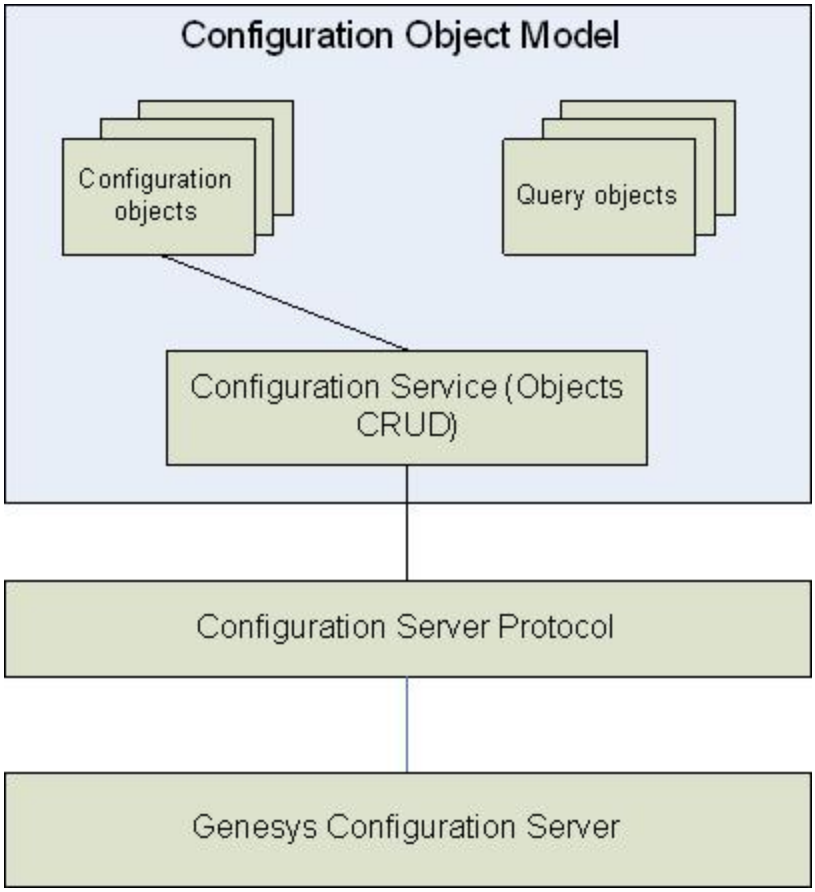
The Configuration Platform SDK allows you to work with objects in the Genesys Configuration Layer by using the interface provided by Configuration Server. Unfortunately, this interface can be difficult to work with. For example, in order to update or create Configuration Layer objects, you have to use special “delta” objects that are distinct from the objects used to retrieve information about Configuration Layer objects.

The Configuration Object Model Application Block provides a consistent and intuitive object model that hides many of the complexities involved in working with Configuration Layer objects. This object model is implemented by way of an event subscription/delivery model, which hides key-value details of the current protocol, and is integrated with the rest of the object model.

The architecture of the Configuration Object Model Application Block consists of three functional components:

- Configuration Objects
- Configuration Service
- Query Objects
- Cache Objects

These components are shown below.



Configuration Objects

Classes and Structures

The Configuration Object Model Application Block supports two types of configuration objects:

- Classes, which can be retrieved directly from Configuration Server using queries.
- Structures, which only exist as properties of classes, and cannot be retrieved directly from Configuration Server.

Classes and structures are different in many ways, but in order to determine whether a given object is a class or a structure, all you need to do is check to see whether the object has a “DBID” property. Classes have this property, while structures do not.

Classes and structures are also different in the following ways:

- Each structure is a property of another class or structure, and therefore must have a “parent” class.
- Classes can be changed and saved to the Configuration Server and structures can only be saved through their “parent” classes.
- Clients can subscribe to events on changes in a class, but not in a structure. To retrieve events on

changes in a structure, clients have to subscribe to changes in its parent class.

Property Types

Both classes and structures have properties. Each property has its own getter and setter methods, and each property is an instance of one of the following types:

- *Simple* — A property that is represented by a value type. Configuration Server supports two types of simple properties - `string` and `integer`. For example, the `CfgPerson` object has `FirstName` and `LastName` properties, both of the `string` type.
- *KV-list* — Tree-like properties that are represented by the `KeyValueCollection` class in the Configuration Object Model. Examples of this property include `userProperties` and `CfgPerson`.
- *Structure* — A complex property that includes one or more properties. In the Configuration Object Model, structures are represented by instances of classes that are similar to configuration objects, but cannot be created directly. For example, in the `CfgPerson` class, its `AgentInfo` property contains *simple*, *kv-list* and other property types.
- *List of structures* — A property that represents more than one structure. In Configuration Object Model, lists of structures are represented by a generic type `IList<structure_type>`, so that the collection is typed, and clients can easily iterate through the collection.
- *Links to a single object* — In Configuration Server, these properties are stored as DBIDs of external objects. The Configuration Object Model automatically resolves these DBIDs into the real objects, which can be manipulated in the same way as the objects directly retrieved from Configuration Server. Links are initialized at the time of the initial request to one of its properties.

> **Tip**
>
> For each link, there are two ways to set the new value of a link. There is a setter method of the property, which uses an object reference to set a new value of a link. There is also a `Set...DBID` method, which uses an integer DBID value.

- *Links to multiple objects* — A property that contains more than one link. In the Configuration Object Model, lists of structures are represented by a generic type `IList<class_type>`, so that the collection is typed, and clients can easily iterate through the collection.

Creating Instances

One way to create an instance of an object in the Configuration Object Model is to invoke a `Retrieve...` method of a `ConfService` class. This set of methods returns instances of objects that already exist in Configuration Server.

To create a new object in Configuration Server, a client must create a new instance of a COM or "*detached*" object. The detached object does not correspond to any objects in Configuration Server until it is saved. The detached object is created using the regular Object-Oriented language object instantiation. For example, a new detached `CfgPerson` object is created using the following construction:

```
[Java]

CfgPerson person = new CfgPerson(confService);
```

An object instance can also be created by using links to external objects. The Component Object Model creates a new object instance whenever the link is called, or any of the properties of a linked object are called. For example, you can write:

```
[Java]

// person has already been retrieved from Configuration Server.
CfgTenant tenant = person.getTenant(); // this is a link to an external object. It is
initialized internally right now
CfgAddress address = tenant.getAddress();
```

Common Methods

Each configuration class contains the following methods:

- Generic `GetProperty(string propertyName)` — Retrieves the property value by its name.
- Generic `SetProperty(string propertyName)` — Sets the new value of the property by its name.
- `Save()` — Commits all changes previously made to the object to Configuration Server. If the object was created detached from Configuration Server and has never been saved, a new object is created in Configuration Server using the `RequestCreateObject` method. If the object has been saved or has been retrieved from Configuration Server, a delta-object, which contains all changes to the object, is formed and sent to Configuration Server by means of the `RequestUpdateObject` method.
- `Delete()` — Deletes the object from the Configuration Server Database.
- `Refresh()` — Retrieves the latest version of the object and refreshes the value of all its properties.

> **Tip**
>
> In this release, all configuration objects are “static,” which means that if the object changes in the Configuration Server, the instance of a class is not automatically changed in the Configuration Object Model. Clients must subscribe to the corresponding event and manually refresh the COM object in order for these changes to take effect.

Configuration Service

> **Tip**
>
> The `IConfService` interface was added to COM in release 8.0. All applications should now use this interface to work with the configuration service instead of the old `ConfService` class. This change is an example of how all COM types in the interface are now referred to by interface; for instance, if a method previously returned `CfgObject` it now returns `ICfgObject`. This is not compatible with existing code, but upgrading should not be difficult as the new interfaces support the same methods as the implementing types.

The Configuration Service (`IConfService`) interface provides services such as retrieval of objects and

subscription to events from Configuration Server. Each connection to a Configuration Server (represented by a `ConfServerProtocol` class of Platform SDK) requires its own instance of the `IConfService` interface.

The protocol class should be created and initialized in the client code prior to `IConfService` initialization.

The `ConfServiceFactory` class is used to create the `IConfService`. This class uses the following syntax:

```
[Java]

IConfService service = ConfServiceFactory.CreateConfService(protocol);
```

Retrieving Objects

Objects can be retrieved from Configuration Service by using one of the following methods:

- `RetrieveObject` — Accepts a query that returns one object. If multiple objects are returned, an exception is thrown.
- `RetrieveMultipleObjects` — Accepts a query that returns one or more objects. A collection of objects is returned.

Each of the `Retrieve...` methods can be either specific (by using generic criteria entries, an object of a specified type is returned) or general (a general object is returned).

Handling Events

The following methods must be called before receiving events from Configuration Server:

1. Register - The application must register its callback by calling the `Register` method from the Configuration Service. This method supplies the client's filter, which enables the client to receive only requested events.
2. Subscribe - The application must subscribe to events from Configuration Server by calling the `Subscribe` method from the Configuration Service. This method provides a notification query object as a parameter.

   The `NotificationQuery` object determines whether the object (or set of objects) to which the client wants to subscribe has changed. The `NotificationQuery` object contains such parameters as `object type, object DBID` and `tenant DBID`.

   After calling the `Subscribe` method, Configuration Server starts sending events to the client. These events are objects, which contain information such as:

   - which object (ID and type) is affected
   - the type of event sent to the client
   - any additional information

   There are three types of events that the client might receive:

   - `ObjectCreated` — A new object has been added to Configuration Server.
   - `ObjectChanged` — Some of the object properties have been modified in Configuration Server.
   - `ObjectDeleted` — The object has been removed from Configuration Server.

Releasing a Configuration Service

Whenever a `ConfService` instance is no longer needed, the `ReleaseConfService` method can be used to remove it from the internal list.

```
[Java]

ConfServiceFactory.ReleaseConfService(service);
```

Query Objects

A query object is an instance of a class that contains information required for a successful query to a Configuration Server. This information includes an object type and its attributes (such as *name* and *tenant*), which are used in the search process.

The inheritance structure of configuration server queries is designed to allow for future expansion. The `CfgQuery` object is the base class for all query objects. Other classes extend `CfgQuery` to provide more specific functionality for different types of queries - for example, all filter-based queries use the `CfgFilterBasedQuery` class. This allows room for future query types (such as XPath) to implemented in this Application Block.

A list of currently available query types is provided below:

- CfgFilterBasedQuery — Contains mapped attribute name-value pairs, as well as the object type.

A special query class is supplied for each configuration object type, in order to facilitate the process of making queries to Configuration Server. For each searchable attribute, the query class has a property that can be set. All of these classes inherit attributes from the `CfgQuery` object, and can be supplied as parameters to the Retrieve... methods which are used to perform searches in Configuration Server.

Cache Objects

The cache functionality is intended to enhance the Configuration Object Model by allowing configuration objects to be stored locally, thereby minimizing requests to configuration server, as well as enhancing ease of use by providing automatic synchronization between locally stored objects and their server-side counterparts.

The cache functionality was designed with the following principles in mind:

- The cache functionality is designed to be extendable with custom implementations of provided interfaces and not via inheritance.
- The cache component is not designed to replicate the Configuration Server query engine or other Configuration Server functionality on the client side.
- Caching must be an optional feature. Work with Configuration Server should not be affected if caching is not used.

Use Cases

Analysis of use cases provides insight into the requirements for applications likely to require configuration cache functionality. The use cases described in the following table were selected for

analysis in order to highlight different functional requirements. There are several possible actors which are referenced in the use cases. The actors are as follows:

- Application - Any application which uses the Configuration Object Model application block
- User - Human (or software) user who may perform actions upon objects in the configuration which are separate from the Application

Use Case	Description	Actor	Steps
PLACE OBJECT INTO CACHE	Place a configuration object into the configuration cache (note the object must have been saved — ie must have a DBID in order to exist in the cache).	Application	1. Application adds object to the cache
PLACE OBJECT INTO CACHE ON SAVE	Place a newly created configuration object into the configuration cache when it is saved.	Application	1. Application creates object 2. Application saves object 3. Configuration Object Model Application Block adds object to the cache
PLACE OBJECT INTO CACHE ON RETRIEVE	Allow for automatic insertion of configuration objects into the cache upon retrieval from configuration server.	Application	1. Application retrieves configuration object 2. Configuration Object Model Application Block retrieves the configuration object from the server 3. Configuration Object Model Application Block places the configuration object into the cache 4. Configuration Object Model Application Block returns the object to the application
OBJECT REMOVED IN CONFIGURATION SERVER	When configuration objects are deleted in the configuration server, the cache can delete the local representation of the object as well.	User	1. User deletes object in the Configuration Server 2. Cache removes

Use Case	Description	Actor	Steps
			corresponding local object upon receiving delete notification 3. Cache sends notification of object deletion to Application
SYNCHRONIZE OBJECT PROPERTIES WITH CONFIGURATION SERVER	When an object stored in the cache is updated in the Configuration Server the object must be updated locally as well.	User	1. User updates a configuration object 2. Cache receives notification about object update 3. Cache updates the object based on the received delta 4. Cache fires event informing any subscribers of object change
FIND OBJECT IN CACHE	The cache must support the ability to find a specific configuration object in the cache using object DBID and type as the criteria for the search.	Application	1. Application retrieves object from cache. 2. If object is in the cache, the cache returns the object. Otherwise the application is notified that the requested object is not in the cache.
ACCESS CACHED OBJECTS	The cache must provide its full object collection to the application.	Application	1. Application requests a complete list of objects from the cache. 2. The cache returns a collection of all cached objects.
RETRIEVE LINKED OBJECT FROM CACHE	If caching is turned on, object links which the Configuration Object Model currently resolves through lazy	Application	1. Application accesses a property which requires link resolution

Use Case	Description	Actor	Steps
	initialization (i.e. if a property linking to another object is accessed, we retrieve the referred-to object from configuration server) must be resolvable through cache access.		2. Configuration Object Model Application Block retrieves the linked object from configuration server and stores it in the cache before returning to the application 3. Application again accesses the property and this time the Configuration Object Model Application Block retrieves the object from the cache
PROVIDE CACHE TRANSPARENCY ON RETRIEVE	A cache search should be performed on attempt to retrieve an object from Configuration Server. If the requested object is found in the cache then the Configuration Object Model should return the cached object rather than accessing Configuration Server.		1. Application creates query to retrieve configuration object 2. Application executes query using the Configuration Object Model 3. Configuration Object Model Application Block searches the cache • If object present, return the object • If object not present, query configuration server for the object
CACHE SERIALIZATION	The cache should support serialization.	Application	1. Application provides a stream to the cache 2. The cache serializes itself into the stream in an XML format 3. Application restarts 4. Application provides

Use Case	Description	Actor	Steps
			the cache a stream of cache data in the same XML format as in step 2 5. Cache restores itself 6. Cache subscribes for updates on the restored objects

Implementation Overview

Two new interfaces for cache management have been added to the Configuration Object Model: the `IConfCache` interface and a default cache implementation (`DefaultConfCache`). Note that the `ConfCache` also implements the Subscriber interface from `MessageBroker` so that the user can subscribe to notifications from Configuration Server, as discussed in *Notification And Delta Handling*.

The `IConfCache` interface provides methods for basic functionality such as adding, updating, retrieving, and removing objects in the cache. It also includes a `Policy` property that defines cache behavior and affects method implementation. (For more details about policies, see *Cache Policy*).

The `DefaultConfCache` component provides a default implementation of the `IConfCache` interface. It serializes and deserializes cache objects using the XML format described in the *XML Format* section, below.

To enable and configure caching functionality, and to specify `ConfService` policy, there are three `CreateConfService` methods available from `ConfServiceFactory`. The original `CreateConfService` method (not shown here) creates a `ConfService` instance that uses the default policy and does not use caching.

```
[Java]

public static IConfService createConfService(Protocol protocol, boolean enableCaching)
```

This method creates an instance of a Configuration Service based on the specified protocol. If caching is enabled, the default caching policy will be used. If `enableCaching` is set to true, caching functionality will be turned on. If caching is disabled, all policy flags related to caching will be false.

```
[Java]

public static IConfService createConfService(Protocol protocol,
        IConfServicePolicy confServicePolicy, IConfCache cache)
```

This method creates a configuration service with the specified policy information. The created service will have caching enabled if a cache object (implementing the `IConfCache` interface) is passed as a parameter.

```
[Java]

public static IConfService createConfService(Protocol protocol,
        IConfServicePolicy confServicePolicy, IConfCachePolicy confCachePolicy)
```

This method creates a configuration service with the specified policy information. The created service

will have caching enabled by default with the cache using the specified cache policy.

XML Format

The "Cache" node will be the root of the configuration cache XML, while "ConfData" is a child of the "Cache" node. The ConfData node contains a collection of XML representations for each configuration object in the cache. The XML format of each object is identical to that which is returned by the `ToXml` method supported by each the Configuration Object Model configuration object.

The "CacheConfiguration" element is a child of the "Cache" node. There can only be one instance of this node and it contains all cache configuration parameters, as follows:

- CONFIGURATIONSERVER NODE — There can be 1..n instances of this element. Each one will represent a configuration server for which the cache is applicable (a cache can be applicable to multiple configuration servers if they are working with the same database as in the case of a primary and backup configuration server pair). Each ConfigurationServer element will have a `URI` attribute specifying the unique URI identifying the Configuration Server, as well as a `Name` attribute specifying the name associated with the endpoint.

The example provided below shows a cache that is applicable for the configuration server at "server:2020" with some policy details specified. There are two objects in the cache for this example: a `CfgDN` and a `CfgService` object.

```
[XML]

<Cache>
  <CacheConfiguration>
    <ConfigurationServer name="serverName" uri="tcp://server:2020"/>
  </CacheConfiguration>
  <ConfData>
    <CfgDN>
      <DBID value="267" />
      <switchDBID value="111" />
      <tenantDBID value="1" />
      <type value="3" />
      <number value="1111" />
      <loginFlag value="1" />
      <registerAll value="2" />
      <groupDBID value="0" />
      <trunks value="0" />
      <routeType value="1" />
      <state value="1" />
      <name value="DNAlias" />
      <useOverride value="2" />
      <switchSpecificType value="1" />
      <siteDBID value="0" />
      <contractDBID value="0" />
      <accessNumbers />
      <userProperties />
    </CfgDN>

    <CfgService>
      <DBID value="102" />
      <name value="Solution1" />
      <type value="2" />
      <state value="1" />
      <solutionType value="1" />
      <components>
        <CfgSolutionComponent>
```

```
          <startupPriority value="3" />
          <isOptional value="2" />
          <appDBID value="153" />
        </CfgSolutionComponent>
      </components>
      <SCSDBID value="102" />
      <assignedTenantDBID value="101" />
      <version value="7.6.000.00" />
      <startupType value="2" />
      <userProperties />
      <componentDefinitions />
      <resources />
    </CfgService>
  </ConfData>
</Cache>
```

Cache Policy

The configuration cache can be assigned a policy represented by a `Policy` interface. A default implementation of the interface will be provided in the `DefaultConfCachePolicy` class.

The `IConfCache` interface interprets the policy as follows:

1. `CacheOnCreate` — When an object is created in the configuration server, the policy will be checked with the created object as the parameter. If the method returns true, the object will be added to the cache, if it is false, the object will not be added. Default implementation will always return false.
2. `RemoveOnDelete` — When an object is deleted in the configuration server, the policy will be checked with the deleted object as the parameter. If the method returns true, the object will be deleted in the cache, if it is false, the notification will be ignored. Default implementation will always return true.
3. `TrackUpdates` — When an object is updated in the configuration server, the policy will be checked with the current version of the object as the parameter. If the method returns true, the object will be updated with the received delta, if it is false, the notification will be ignored. Default implementation will always return true.
4. `ReturnCopies` — Determines whether the cache should return copies of objects when they are retrieved from the cache, or the original, cached versions. False by default.

IConfServicePolicy Interface

The `IConfServicePolicy` interface can be used to define the policy settings for the `ConfService`. Two default implementations are available:

1. `DefaultConfServicePolicy` contains the settings for a non-caching configuration service. That is, all of the cache-related policy flags will always return false.
2. `CachingConfServicePolicy` defines the default behavior for a configuration service with caching enabled. (Note that when referring to the "default" value below, we will be referring to this implementation.)

The policy interface settings are interpreted as follows:

- AttemptLinkResolutionThroughCache — Whenever a link resolution attempt is made, this policy will be checked for the type of object the link refers to. If this method returns true, the link resolution attempt will first be made through the cache. If the method returns false, or if the object has not been found in the cache, the server will queried. Default value is always true.

- `CacheOnRetrieve` — This method will be called for each object retrieved from the configuration. If the return value is "true" the object will be added to the cache. Default value is always true.
- `CacheOnSave` — This method will be called for each object that is being saved. If the return value is true, the object will be added to the cache. If the object is already in the cache, it will not be overwritten. Default value is always true.
- `ValidateBeforeSave` — This is a property from the `ConfService` which will be moved to the policy interface and is not related to caching. It is used to indicate whether property values are checked for valid values against the schema before a save attempt is made. Default value is true.
- `QueryCacheOnRetrieve` — This method will be called every time a retrieve operation is performed using a query. The `ConfService` will first check the cache for the existence of the requested configuration object. If the object exists, it will be returned and no configuration server request will be made. If there are no values returned, the `ConfService` will query the configuration server (see *Query Engine*). Default value is always false.
- `QueryCacheOnRetrieveMultiple` — This method will be called every time a retrieve multiple operation is performed. The `ConfService` will first execute the query against cache. If the returned object count is greater than 0 the found object collection will be returned and no configuration server request will be made. If there are no values returned, the `ConfService` will query the configuration server (see *Query Engine*). Default value is always false.

Note that the `RetrieveMultiple` operation is NOT implemented in the default query engine, so providing a policy where this method returns true will require a new query engine implementation.

Cache Extendability

Consistent with the design principles outlined above, the configuration cache is extendable via custom implementations of provided interfaces. The two areas of the cache which can be extended are the cache storage and the cache query engine.

Cache Storage

The storage interface defines the method by which objects are stored in the cache. When an instance of an implementing object is provided to the cache, the cache will store all cached objects in the storage component.

The default storage implementation stores cached objects using the object type and DBID as keys. Note that this means that objects in the cache are assumed to be from one configuration database. The default implementation is also thread safe using a reader/writer lock which allows for multiple concurrent readers and one writer. The storage methods are as follows:

- Add — Adds a new object to the storage. If object already exists in the storage, the default implementation thrown an exception.
- Update — Overwrites an existing object in the storage. If the object is not found in the storage, the default implementation creates a new version of the object.
- Remove — Removes an object from the storage.
- Retrieve — Retrieves an enumerable list of all objects in the storage (filtered by type), and possibly influenced by an optional helper parameter. Note that the helper parameter is not meant to provide querying logic — that should be done in the query engine. Because the query engine is to some degree dependent on the storage implementation, the helper parameter allows for some flexibility in the way stored objects are enumerated for the query engine. The default implementation can take a

`CfgObjectType` as a helper parameter.

- Clear — Removes all objects in the storage.

Query Engine

The query engine provides the ability to define the method by which objects are located in the cache.

Depending on the `IConfService` policy, `Retrieve` requests as well as link resolution can first be attempted through the cache. If the requested object is found in the cache, then that cached object is returned instead of sending a request to Configuration Server. If the object is not present in the cache, a request to Configuration Server is made.

A user-definable query engine module exists inside the cache to achieve this functionality. A query engine must implement the `IConfCacheQueryEngine` interface, which provides methods to retrieve objects (either individually, or as a list) and to test a query and determine if it can be executed.

If enabled by the policy, `IConfService` will attempt a query to its cache using the cache's query engine interface. If a result is returned, the `IConfService` will not query the Configuration Server. By following this contract, the Configuration Object Model user is then able to create a custom implementation of the `IConfCacheQueryEngine` with any extended search capabilities which may be missing from the simple default implementation.

Two implementations of the `IConfCacheQueryEngine` interface are provided in the Configuration Object Model, as described below:

- DEFAULTCONFCACHEQUERYENGINE CLASS - The `DefaultConfCacheQueryEngine` class is a default implementation of the `IConfCacheQueryEngine` interface.
- COMPOSITECONFCACHEQUERYENGINE CLASS - This class is a more advanced implementation of the query engine which allows child query engine modules to be registered in order to interpret different types of queries. It does not have a default query engine implementation, only the mechanism for working with multiple child query engines.

Notification and Delta Handling

The default configuration cache will implement the `Subscriber<ConfEvent>` interface which will allow the cache to be subscribed to receive configuration events. When a cache instance is associated with a Configuration Service, it will automatically be subscribed for configuration events from that service (note that if a custom cache implementation also implements this interface it will be subscribed for events as well). The way the cache is updated based on these notifications is determined by the cache policy.

In addition, a new filter class will be added in order to allow the subscriber to filter the cache events. The `ConfCacheFilter` will implement the MessageBroker's `Predicate` interface, allowing for the filter to be passed during registration for events via `SubscriptionService`. The `ConfCacheFilter`'s properties will specify the parameters by which the events will be filtered. Initially, the supported parameters will be object type, object DBID, and update type, allowing the user to filter events by one or a combination of these parameters assuming an AND relationship between the parameters specified.

Using the Application Block

Installing the Configuration Object Model Application Block

Before you install the Configuration Object Model Application Block, it is important to review the software requirements established in the Genesys Supported Operating Environment Reference Manual.

Building the Configuration Object Model Application Block

> **Tip**
>
> Starting with release 8.5.0, the common interfaces for COM Application Block and Message Broker code have been moved to an individual `commonsappblock.jar` file.

To build the Configuration Object Model Application Block:

1. Open the `<Platform SDK Folder>\applicationblocks\com` folder.
2. Run either `build.bat` or `build.sh`, depending on your platform.

This will create the `commonsappblock.jar` file, located within the `<Platform SDK Folder>\applicationblocks\com\dist\lib` directory.

Using the QuickStart Application

The easiest way to start using the Configuration Object Model Application Block is to use the bundled QuickStart application. This application ships in the same folder as the application block.

Configuring the QuickStart Application

In order to use the QuickStart application, you will need to change some lines of code in the `quickstart.properties` file, located in the `<Platform SDK Folder>\applicationblocks\com\quickstart` directory. Change the following lines to point to your Configuration Server, and then save the updated file:

```
ConfServerUri = tcp://:

ConfServerUser =
ConfServerPassword =

ConfServerClientName = default
ConfServerClientType = CFGSCE
```

Building the QuickStart Application

1. Open the `<Platform SDK Folder>\applicationblocks\com\quickstart` folder.
2. Run either `build.bat` or `build.sh`, depending on your platform.

Running the QuickStart Application

1. Open a Command Prompt or Terminal window.
2. Navigate to the `<Platform SDK Folder>\applicationblocks\com\quickstart` directory.
3. Run either `quickstart.bat` or `quickstart.sh`, depending on your platform.

How to Properly Initialize the ConfService Instance

To work with Configuration Server, the `ConfService` instance needs `ConfServerProtocol`.

Platform SDK protocol connections allow users to manage connections, setup custom asynchronus `MessageHandler` objects, substitute message receivers, and subscribe for protocol messages and channel events. So, to maintain Platform SDK flexibility, the Configuration Object Model Application Block does not manage a `ConfServerProtocol` connection inside of the `ConfService` - this must be done by the user. Instead users may create a simple instance and initialize it with `WarmStandbyService`.

It is important to note that asycnhronous protocol events may be configured for delivery to a single destination, with only one `MessageHandler` or `MessageReceiver` for one protocol instance. Starting from Platform SDK release 8.1.1, `ConfService` may be initialized without use of legacy Message Broker Application Block. Starting from version 8.5, this is the only way to create `ConfService`.

If your application needs to receive asynchronous protocol messages from Configuration Server on the protocol instance where `ConfService` is initialized, that can be done using `ConfService.setUserMessageHandler(messageHandler)`.

Protocol Initialization

A `ConfServerProtocol` instance is required for the creation of `ConfService`. It should be initialized with an Endpoint and handshake properties, but without setting either `confServerProtocol.setMessageHandler()` or `confServerProtocol.setReceiver()`.

```
// Initialize ConfService:
PropertyConfiguration       config;
ConfServerProtocol          confServerProtocol;
IConfService                confService;

config = new PropertyConfiguration();
config.setUseAddp(true);
config.setAddpClientTimeout(15);

confServerProtocol = new ConfServerProtocol(new Endpoint("ConfigServer", csHost, csPort,
config));
confServerProtocol.setUserName(userName);
confServerProtocol.setUserPassword(password);
confServerProtocol.setClientName(clientName);
confServerProtocol.setClientApplicationType(clientType.ordinal());
```

> **Important**
>
> Do *not* open the protocol before `ConfService` is created. `ConfService` sets its own internal `MessageHandler`, and this operation can only be done on a closed channel.

ConfService Initialization

```
confService = ConfServiceFactory.createConfService(confServerProtocol);
confServerProtocol.open();
```

ConfService Shutdown

```
confServerProtocol.close();
ConfServiceFactory.releaseConfService(confService);
confService = null;
```

Application Components Usage Notes

Older releases of `ProtocolManagementService` do not support using `ConfService` without the Message Broker service - an exception raised when users try to create the `ConfService` object on a protocol instance initialized by the Protocol Manager Application Block. To migrate away from Protocol Manager Application Block usage, we recommend creating and configuring `ConfServerProtocol` without Protocol Manager Application Block usage, as shown above.

`MessageHandler` is not compatible with the deprecated `MessageReceiver`; it is only possible to use one of these components on a protocol instance. Specific to Platform SDK for Java is the limitation that one protocol instance may have only one instance of `MessageHandler`. So, if an application uses a custom `MessageHandler` on a protocol used for `ConfService`, then only one handler will be able to receive asynchronous protocol events.

If application overwrites the `ConfService` object after creation, then that service will be unable to receive Configuration Server notifications or to perform multiple objects reading operations - a timeout exception will occur. If there is a need to get those protocol messages separately from `ConfService` logic, it is possible to initialize custom `MessageHandler` with `confService.setUserMessageHandler(messageHandler)`.

Notes for Previous Releases of Platform SDK

'''[+] Platform SDK 8.1.0 Specific Notes'''

Platform SDK 8.1.0 included some improvements to the Message Broker Application Block.

There was a new `EventReceivingBrokerService` class that implements the receiver interface, which can be used as an external receiver for Platform SDK protocols. When this class is in use, protocol messages will be handled a little bit faster (compared to the older Message Broker service) with no redundant intermediate queue, and there is no additional thread sleeping/waiting.

```
EventReceivingBrokerService broker = new EventReceivingBrokerService();
broker.setInvoker(new SingleThreadInvoker("COMBrokerService-" + cfgsrvEndpointName));
```

```
ConfServerProtocol protocol = new ConfServerProtocol(endpoint);
protocol.setReceiver(broker);
protocol.setUserName(...);
protocol.set...();
protocol.open();

IConfService confService = ConfServiceFactory.createConfService(protocol, broker);
```

To shutdown the Configuration Object Model Application Block, you can use the following code:

```
protocol.close();
ConfServiceFactory.releaseConfService(confService);
```

'''[+] Platform SDK 8.0, 7.6 Specific Notes'''

In earlier releases of Platform SDK, the initialization logic could look like this:

```
ConfServerProtocol protocol = new ConfServerProtocol(endpoint);
protocol.setUserName(...);
protocol.set...();
protocol.open();

EventBrokerService broker = BrokerServiceFactory.CreateEventBroker(protocol);
IConfService confService = ConfServiceFactory.createConfService(protocol, broker);
```

If the protocol has an external receiver initialized (for example, with Protocol Manager usage), then the `EventBrokerService` should be initialized on that receiver instead of the protocol itself:

```
EventBrokerService broker =
BrokerServiceFactory.CreateEventBroker(protocolManager.getReceiver());
```

To shutdown the Configuration Object Model Application Block, you can use the following code:

```
protocol.close();
broker.dispose();
ConfServiceFactory.releaseConfService(confService);
```

> **Important**
>
> Legacy `EventBrokerService` objects need to be disposed on shutdown because they include an internal reading thread which should be stopped.

Editing Capacity Rules

The Configuration Object Model Application Block includes the `CapacityRuleHelper` class (introduced in release 8.1.4) which allows you to edit and update Capacity Rules. This helper class presents an XML representation for `CfgScript` objects of type `CfgScriptType.CFGCapacityRule`, which can be updated and saved to edit existing Capacity Rules.

An example of how to edit capacity rules is provided below.

```
IConfService service = (IConfService)ConfServiceFactory.createConfService(protocol);
service.getProtocol().open();
CfgScriptQuery query = new CfgScriptQuery(service);
CfgScript script = (CfgScript)service.retrieveObject(query);
```

An instance of the `CapacityRuleHelper` class can now be created with static method `create`. This method validates the input script object and can throw an exception: `ConfigException` if the script has an invalid format, or `IllegalArgumentException` if the script is null or the script type is not valid. Once the instance is created, `getXMLPresentation()` allows you access to Capacity Rules.

```
CapacityRuleHelper helper = CapacityRuleHelper.create(script);
Document doc = helper.getXMLPresentation();
// edit xml document here
```

The `setXMLPresentation()` method allows you to save changes to the XML into the `CapacityRuleHelper` class instance. Once changes have been made to the XML document, apply your changes using the following code:

```
helper.setXMLPresentation(doc);
helper.getCfgScript().save();
service.getProtocol().close();
ConfServiceFactory.releaseConfService(service);
```

.NET

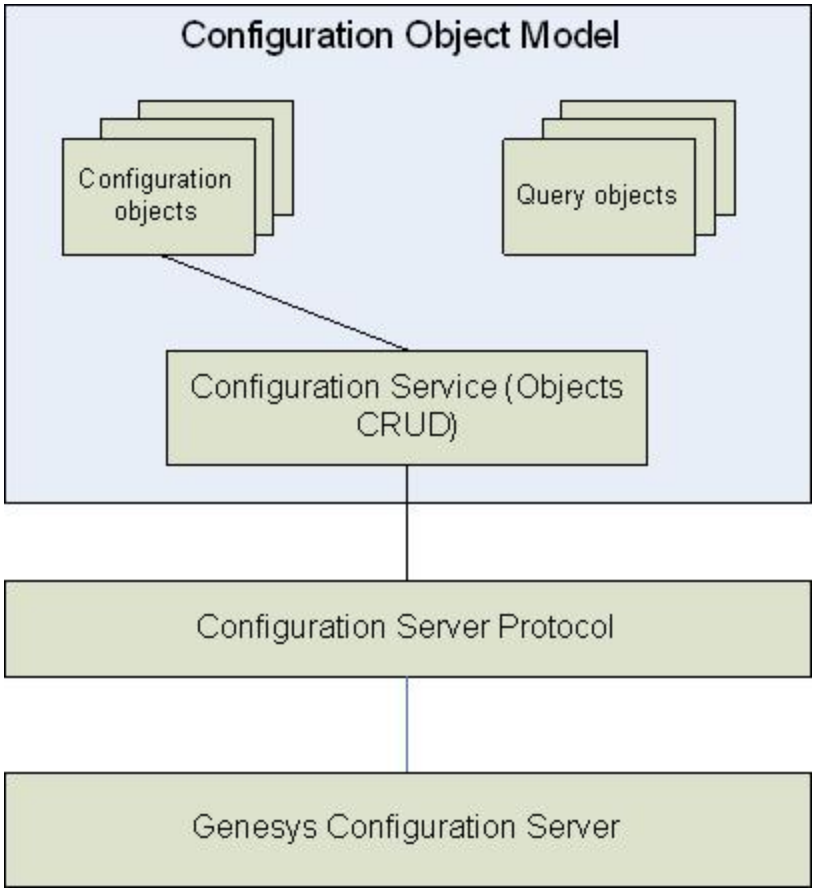
Architecture and Design

The Configuration Object Model Application Block provides a consistent and intuitive object model for working with Configuration Server objects, as well as a straightforward object model for queries with different filters. This Application Block hides the complexities of object creation and changing by means of "delta" objects. It also creates an event subscription/delivery model, which hides key-value details of the current protocol, and is integrated with the rest of the object model.

The architecture of the Configuration Object Model Application Block consists of three functional components:

- Configuration Objects
- Configuration Service
- Query Objects
- Cache Objects

These components are shown in the figure below.



Configuration Objects

Classes and Structures

There are two types of configuration objects are supported by the Configuration Object Model Application Block:

- Classes, which can be retrieved directly from Configuration Server using queries.
- Structures, which only exist as properties of classes, and cannot be retrieved directly from Configuration Server.

The main differences between classes and structures are as follows:

1. Each structure is a property of another class or structure, and therefore must have a "parent" class.
2. Classes can be changed and saved to the Configuration Server and structures can only be saved through their "parent" classes.
3. Clients can subscribe to events on changes in a class, but not in a structure. To retrieve events on changes in a structure, clients have to subscribe to changes in a parent class.

Property Types

Both classes and structures have properties. For each property, the object has getter and setter methods which retrieve the value of the property and set a new value correspondingly. However, some properties are read-only and therefore will only have a getter method. For each object, its properties can be one of the following types:

- *Simple* — A property that is represented by a value type. Configuration Server supports two types of simple properties - `string` and `integer`. For example, the `CfgPerson` object has `FirstName` and `LastName` properties, both of the string type.
- *KV-list* — Tree-like properties that are represented by the `KeyValueCollection` class in the Configuration Object Model. Examples of this property include `userProperties` of `CfgPerson`.
- *Structure* — A complex property that includes one or more properties. In the Configuration Object Model, structures are represented by instances of classes that are similar to configuration objects, but cannot be created directly. For example, in the `CfgPerson` class, its `AgentInfo` property contains simple, kv-list and other property types.
- *List of structures* — A property that represents more than one structure. In Configuration Object Model, lists of structures are represented by a generic type `IList<structure_type>`, so that the collection is typed, and clients can easily iterate through the collection.
- *Links to a single object* — In Configuration Server, these properties are stored as DBIDs of external objects. The Configuration Object Model automatically resolves these DBIDs into the real objects, which can be manipulated in the same way as the objects directly retrieved from Configuration Server. Links are initialized at the time of the initial request to one of its properties.

> **Tip**
>
> For each link, there are two ways to set the new value of a link. There is a setter method of the property, which uses an object reference to set a new value of a link. There is also a Set DBID method, which uses an integer DBID value.

- *Links to multiple objects* — A property that contains more than one link. In the Configuration Object Model, lists of structures are represented by a generic type `IList<class_type>`, so that the collection is typed, and clients can easily iterate through the collection.

Creating Instances

One way to create an instance of an object in the Configuration Object Model is to invoke one of the Retrieve methods of the `ConfService` class. This set of methods returns instances of objects that already exist in Configuration Server.

To create a new object in Configuration Server, a client must create a new instance of a COM or "detached" object. The detached object does not correspond to any objects in Configuration Server until it is saved. The detached object is created using the regular object-oriented language object instantiation. For example, a new detached `CfgPerson` object is created using the following construction:

```
[C#]

CfgPerson person = new CfgPerson(confService);
```

An object instance can also be created by using links to external objects. The Component Object Model creates a new object instance whenever the link is called, or any of the properties of a linked object are called. For example, you can write:

```
[C#]

// Person has already been retrieved from Configuration Server.
CfgTenant tenant = person.Tenant;
// This is a link to an external object. It is initialized internally right now...
CfgAddress address = tenant.Address;
```

Common Methods

Each configuration class contains the following methods:

- `Generic GetProperty(string propertyName)` — Retrieves the property value by its name.
- `Generic SetProperty(string propertyName)` — Sets the new value of the property by its name.
- `Save()` — Commits all changes previously made to the object to Configuration Server. If the object was created detached from Configuration Server and has never been saved, a new object is created in Configuration Server using the `RequestCreateObject` method. If the object has been saved or has been retrieved from Configuration Server, a delta-object, which contains all changes to the object, is formed and sent to Configuration Server by means of the `RequestUpdateObject` method.
- `Delete()` — Deletes the object from the Configuration Server Database.
- `Refresh()` — Retrieves the latest version of the object and refreshes the value of all its properties.

> **Tip**
>
> In this release, all configuration objects are "static," which means that if the object changes in the Configuration Server, the instance of a class is not automatically changed in the Configuration Object Model. Clients must subscribe to the corresponding event and manually refresh the COM object in order for these changes to take effect.

Configuration Service

> **Important**
>
> The `IConfService` interface was added to COM in release 8.0. All applications should now use this interface to work with the configuration service instead of the old `ConfService` class. This change is an example of how all COM types in the interface are now referred to by interface; for instance, if a method previously returned `CfgObject` it now returns `ICfgObject`. This is not compatible with existing code, but upgrading should not be difficult as the new interfaces support the same methods as the implementing types.

The Configuration Service (`IConfService`) interface provides services such as retrieval of objects and

subscription to events from Configuration Server. Each connection to a Configuration Server (represented by a `ConfServerProtocol` class of Platform SDK) requires its own instance of the `IConfService` interface.

The protocol class should be created and initialized in the client code prior to `IConfService` initialization.

The `ConfServiceFactory` class is used to create the `IConfService`. This class uses the following syntax:

```
[C#]

IConfService service = ConfServiceFactory.CreateConfService(protocol);
```

Retrieving Objects

Objects can be retrieved from Configuration Service by using one of the following methods:

- `RetrieveObject` — Accepts a query that returns one object. If multiple objects are returned, an exception is thrown.
- `RetrieveMultipleObjects` — Accepts a query that returns one or more objects. A collection of objects is returned.

Each of the `Retrieve` methods can be can be strongly typed (with use of generics, an object of a specified type is returned) or general (a general object is returned).

Handling Events

The following methods must be called before receiving events from Configuration Server:

1. Register

The application must register its callback by calling the `Register` method from the Configuration Service. This method supplies the client's filter, which enables the client to receive only requested events.

2. Subscribe

The application must subscribe to events from Configuration Server by calling the `Subscribe` method from the Configuration Service. This method provides a notification query object as a parameter.

The `NotificationQuery` object determines whether the object (or set of objects) to which the client wants to subscribe has changed. The `NotificationQuery` object contains such parameters as object type, object DBID and tenant DBID.

After calling the `Subscribe` method, Configuration Server starts sending events to the client. These events are objects, which contain information such as:

- which object (ID and type) is affected
- the type of event sent to the client
- any additional information

There are three types of events that the client might receive:

- `ObjectCreated` — A new object has been added to Configuration Server.
- `ObjectChanged` — Some of the object properties have been modified in Configuration Server.
- `ObjectDeleted` — The object has been removed from Configuration Server.

Logging Messages

Configuration Object Model Application Block supports logging through the standard Platform SDK logging interfaces. The `IConfService` interface inherits the `EnableLogging` method that provides the ability to log messages through the provided `ILogger` interface.

Releasing a Configuration Service

Whenever a `ConfService` instance is no longer needed, the `ReleaseConfService` method can be used to remove it from the internal list.

```
[C#]

ConfServiceFactory.ReleaseConfService(service);
```

Query Objects

A query object is an instance of a class that contains information required for a successful query to a Configuration Server. This information includes an object type and its attributes (such as `name` and `tenant`), which are used in the search process.

The inheritance structure of configuration server queries is designed to allow for future expansion. The `CfgQuery` object is the base class for all query objects. Other classes extend `CfgQuery` to provide more specific functionality for different types of queries - for example, all filter-based queries use the `CfgFilterBasedQuery` class. This allows room for future query types (such as XPath) to be implemented in this Application Block.

A list of currently available query types is provided below:

- `CfgFilterBasedQuery` — Contains mapped attribute name-value pairs, as well as the object type.

A special query class is supplied for each configuration object type, in order to facilitate the process of making queries to Configuration Server. For each searchable attribute, the query class has a property that can be set. All of these classes inherit attributes from the `CfgQuery` object, and can be supplied as parameters to the `Retrieve` methods which are used to perform searches in Configuration Server.

Cache Objects

The cache functionality is intended to enhance the Configuration Object Model by allowing configuration objects to be stored locally, thereby minimizing requests to configuration server, as well as enhancing ease of use by providing automatic synchronization between locally stored objects and their server-side counterparts.

The cache functionality was designed with the following principles in mind:

- The cache functionality is designed to be extendable with custom implementations of provided interfaces and not via inheritance.
- The cache component is not designed to replicate the Configuration Server query engine or other Configuration Server functionality on the client side.
- Caching must be an optional feature. Work with Configuration Server should not be affected if caching is not used.

Use Cases

Analysis of use cases provides insight into the requirements for applications likely to require configuration cache functionality. The use cases described in the following table were selected for analysis in order to highlight different functional requirements. There are several possible actors which are referenced in the use cases. The actors are as follows:

- Application - Any application which uses the Configuration Object Model application block
- User - Human (or software) user who may perform actions upon objects in the configuration which are separate from the Application

Use Case	Description	Actor	Steps
PLACE OBJECT INTO CACHE	Place a configuration object into the configuration cache (note the object must have been saved — ie must have a DBID in order to exist in the cache).	Application	1. Application adds object to the cache
PLACE OBJECT INTO CACHE ON SAVE	Place a newly created configuration object into the configuration cache when it is saved.	Application	1. Application creates object 2. Application saves object 3. Configuration Object Model Application Block adds object to the cache
PLACE OBJECT INTO CACHE ON RETRIEVE	Allow for automatic insertion of configuration objects into the cache upon retrieval from configuration server.	Application	1. Application retrieves configuration object 2. Configuration Object Model Application Block retrieves the configuration object from the server 3. Configuration Object Model Application Block places the configuration object

Use Case	Description	Actor	Steps
			into the cache 4. Configuration Object Model Application Block returns the object to the application
OBJECT REMOVED IN CONFIGURATION SERVER	When configuration objects are deleted in the configuration server, the cache can delete the local representation of the object as well.	User	1. User deletes object in the Configuration Server 2. Cache removes corresponding local object upon receiving delete notification 3. Cache sends notification of object deletion to Application
SYNCHRONIZE OBJECT PROPERTIES WITH CONFIGURATION SERVER	When an object stored in the cache is updated in the Configuration Server the object must be updated locally as well.	User	1. User updates a configuration object 2. Cache receives notification about object update 3. Cache updates the object based on the received delta 4. Cache fires event informing any subscribers of object change
FIND OBJECT IN CACHE	The cache must support the ability to find a specific configuration object in the cache using object DBID and type as the criteria for the search.	Application	1. Application retrieves object from cache. 2. If object is in the cache, the cache returns the object. Otherwise the application is notified that the requested object is not in the cache.

Use Case	Description	Actor	Steps
ACCESS CACHED OBJECTS	The cache must provide its full object collection to the application.	Application	1. Application requests a complete list of objects from the cache. 2. The cache returns a collection of all cached objects.
RETRIEVE LINKED OBJECT FROM CACHE	If caching is turned on, object links which the Configuration Object Model currently resolves through lazy initialization (i.e. if a property linking to another object is accessed, we retrieve the referred-to object from configuration server) must be resolvable through cache access.	Application	1. Application accesses a property which requires link resolution 2. Configuration Object Model Application Block retrieves the linked object from configuration server and stores it in the cache before returning to the application 3. Application again accesses the property and this time the Configuration Object Model Application Block retrieves the object from the cache
PROVIDE CACHE TRANSPARENCY ON RETRIEVE	A cache search should be performed on attempt to retrieve an object from Configuration Server. If the requested object is found in the cache then the Configuration Object Model should return the cached object rather than accessing Configuration Server.		1. Application creates query to retrieve configuration object 2. Application executes query using the Configuration Object Model 3. Configuration Object Model Application Block searches the cache • If object present, return the object • If object not present, query configuration

Use Case	Description	Actor	Steps
			server for the object
CACHE SERIALIZATION	The cache should support serialization.	Application	1. Application provides a stream to the cache 2. The cache serializes itself into the stream in an XML format 3. Application restarts 4. Application provides the cache a stream of cache data in the same XML format as in step 2 5. Cache restores itself 6. Cache subscribes for updates on the restored objects

Implementation Overview

Two new interfaces for cache management have been added to the Configuration Object Model: the `IConfCache` interface and a default cache implementation (`DefaultConfCache`). Note that the `ConfCache` also implements the `ISubscriber` interface from `MessageBroker`. The cache implements `ISubscriber` in order to allow the user to subscribe to notifications from Configuration Server, as discussed in *Notification And Delta Handling*.

The `IConfCache` interface provides methods for basic functionality such as adding, updating, retrieving, and removing objects in the cache. It also includes a `Policy` property that defines cache behavior and affects method implementation. (For more details about policies, see *Cache Policy*).

The `DefaultConfCache` component provides a default implementation of the `IConfCache` interface. It serializes and deserializes cache objects using the XML format described in the *XML Format* section, below.

To enable and configure caching functionality, and to specify `ConfService` policy, there are three `CreateConfService` methods available from `ConfServiceFactory`. The original `CreateConfService` method (not shown here) creates a `ConfService` instance that uses the default policy and does not use caching.

```
[C#]

public static IConfService CreateConfService(IProtocol protocol, bool enableCaching)
```

This method creates an instance of a Configuration Service based on the specified protocol. If caching is enabled, the default caching policy will be used. If `enableCaching` is set to true, caching

functionality will be turned on. If caching is disabled, all policy flags related to caching will be false.

```
[C#]

public static IConfService CreateConfService(IProtocol protocol,
        IConfServicePolicy confServicePolicy, IConfCache cache)
```

This method creates a configuration service with the specified policy information. The created service will have caching enabled if a cache object (implementing the `IConfCache` interface) is passed as a parameter.

```
[C#]

public static IConfService CreateConfService(IProtocol protocol,
        IConfServicePolicy confServicePolicy, IConfCachePolicy confCachePolicy)
```

This method creates a configuration service with the specified policy information. The created service will have caching enabled by default with the cache using the specified cache policy.

XML Format

The "Cache" node will be the root of the configuration cache XML, while "ConfData" is a child of the "Cache" node. The ConfData node contains a collection of XML representations for each configuration object in the cache. The XML format of each object is identical to that which is returned by the `ToXml` method supported by each the Configuration Object Model configuration object.

The "CacheConfiguration" element is a child of the "Cache" node. There can only be one instance of this node and it contains all cache configuration parameters, as follows:

- CONFIGURATIONSERVER NODE – There can be 1..n instances of this element. Each one will represent a configuration server for which the cache is applicable (a cache can be applicable to multiple configuration servers if they are working with the same database as in the case of a primary and backup configuration server pair). Each `ConfigurationServer` element will have a `URI` attribute specifying the unique URI identifying the Configuration Server, as well as a `Name` attribute specifying the name associated with the endpoint.

The example provided below shows a cache that is applicable for the configuration server at "server:2020" with some policy details specified. There are two objects in the cache for this example: a `CfgDN` and a `CfgService` object.

```
[XML]

<Cache>
        <CacheConfiguration>
                <ConfigurationServer name="serverName" uri="tcp://server:2020"/>
        </CacheConfiguration>

        <ConfData>
                <CfgDN>
                        <DBID value="267" />
                        <switchDBID value="111" />
                        <tenantDBID value="1" />
                        <type value="3" />
                        <number value="1111" />
                        <loginFlag value="1" />
                        <registerAll value="2" />
                        <groupDBID value="0" />
                        <trunks value="0" />
```

```
                        <routeType value="1" />
                        <state value="1" />
                        <name value="DNAlias" />
                        <useOverride value="2" />
                        <switchSpecificType value="1" />
                        <siteDBID value="0" />
                        <contractDBID value="0" />
                        <accessNumbers />
                        <userProperties />
                </CfgDN>

                <CfgService>
                        <DBID value="102" />
                        <name value="Solution1" />
                        <type value="2" />
                        <state value="1" />
                        <solutionType value="1" />
                        <components>
                                <CfgSolutionComponent>
                                        <startupPriority value="3" />
                                        <isOptional value="2" />
                                        <appDBID value="153" />
                                </CfgSolutionComponent>
                        </components>
                        <SCSDBID value="102" />
                        <assignedTenantDBID value="101" />
                        <version value="7.6.000.00" />
                        <startupType value="2" />
                        <userProperties />
                        <componentDefinitions />
                        <resources />
                </CfgService>
        </ConfData>

</Cache>
```

Cache Policy

The configuration cache can be assigned a policy represented by a Policy interface. A default implementation of the interface will be provided in the `DefaultConfCachePolicy` class.

The `IConfCache` interface will interpret the policy as follows:

1. `CacheOnCreate` – When an object is created in the configuration server, the policy will be checked with the created object as the parameter. If the method returns true, the object will be added to the cache, if it is false, the object will not be added. Default implementation will always return false.
2. `RemoveOnDelete` – When an object is deleted in the configuration server, the policy will be checked with the deleted object as the parameter. If the method returns true, the object will be deleted in the cache, if it is false, the notification will be ignored. Default implementation will always return true.
3. `TrackUpdates` – When an object is updated in the configuration server, the policy will be checked with the current version of the object as the parameter. If the method returns true, the object will be updated with the received delta, if it is false, the notification will be ignored. Default implementation will always return true.
4. `ReturnCopies` – Determines whether the cache should return copies of objects when they are retrieved from the cache, or the original, cached versions. False by default.

IConfServicePolicy Interface

The `IConfServicePolicy` interface can be used to define the policy settings for the `ConfService`. Two default implementations are available:

1. `DefaultConfServicePolicy` contains the settings for a non-caching configuration service. That is, all of the cache-related policy flags will always return false.
2. `CachingConfServicePolicy` defines the default behavior for a configuration service with caching enabled. (Note that when referring to the "default" value below, we will be referring to this implementation.)

The policy interface settings are interpreted as follows:

- `AttemptLinkResolutionThroughCache` – Whenever a link resolution attempt is made, this policy will be checked for the type of object the link refers to. If this method returns true, the link resolution attempt will first be made through the cache. If the method returns false, or if the object has not been found in the cache, the server will queried. Default value is always true.
- `CacheOnRetrieve` – This method will be called for each object retrieved from the configuration. If the return value is "true" the object will be added to the cache. Default value is always true.
- `CacheOnSave` – This method will be called for each object that is being saved. If the return value is true, the object will be added to the cache. If the object is already in the cache, it will not be overwritten. Default value is always true.
- `ValidateBeforeSave` – This is a property from the `ConfService` which will be moved to the policy interface and is not related to caching. It is used to indicate whether property values are checked for valid values against the schema before a save attempt is made. Default value is true.
- `QueryCacheOnRetrieve` – This method will be called every time a retrieve operation is performed using a query. The `ConfService` will first check the cache for the existence of the requested configuration object. If the object exists, it will be returned and no configuration server request will be made. If there are no values returned, the `ConfService` will query the configuration server (see *Query Engine*). Default value is always false.
- `QueryCacheOnRetrieveMultiple` – This method will be called every time a retrieve multiple operation is performed. The ConfService will first execute the query against cache. If the returned object count is greater than 0 the found object collection will be returned and no configuration server request will be made. If there are no values returned, the `ConfService` will query the configuration server (see *Query Engine*). Default value is always false.

Note that the `RetrieveMultiple` operation is NOT implemented in the default query engine, so providing a policy where this method returns true will require a new query engine implementation.

Cache Extendability

Consistent with the design principles outlined above, the configuration cache is extendable via custom implementations of provided interfaces. The two areas of the cache which can be extended are the cache storage and the cache query engine.

Cache Storage

The storage interface defines the method by which objects are stored in the cache. When an instance of an implementing object is provided to the cache, the cache will store all cached objects in the storage component.

The default storage implementation stores cached objects using the object type and DBID as keys. Note that this means that objects in the cache are assumed to be from one configuration database. The default implementation is also thread safe using a reader/writer lock which allows for multiple concurrent readers and one writer. The storage methods are as follows:

- Add – Adds a new object to the storage. If object already exists in the storage, the default implementation thrown an exception.
- Update – Overwrites an existing object in the storage. If the object is not found in the storage, the default implementation creates a new version of the object.
- Remove – Removes an object from the storage.
- Retrieve – Retrieves an enumerable list of all objects in the storage (filtered by type), and possibly influenced by an optional helper parameter. Note that the helper parameter is not meant to provide querying logic – that should be done in the query engine. Because the query engine is to some degree dependent on the storage implementation, the helper parameter allows for some flexibility in the way stored objects are enumerated for the query engine. The default implementation can take a `CfgObjectType` as a helper parameter.
- Clear–Removes all objects in the storage.

Query Engine

The query engine provides the ability to define the method by which objects are located in the cache.

Depending on the `IConfService` policy, `Retrieve` requests as well as link resolution can first be attempted through the cache. If the requested object is found in the cache, then that cached object is returned instead of sending a request to Configuration Server. If the object is not present in the cache, a request to Configuration Server is made.

A user-definable query engine module exists inside the cache to achieve this functionality. A query engine must implement the `IConfCacheQueryEngine` interface, which provides methods to retrieve objects (either individually, or as a list) and to test a query and determine if it can be executed.

If enabled by the policy, `IConfService` will attempt a query to its cache using the cache's query engine interface. If a result is returned, the `IConfService` will not query the Configuration Server. By following this contract, the Configuration Object Model user is then able to create a custom implementation of the `IConfCacheQueryEngine` with any extended search capabilities which may be missing from the simple default implementation.

Two implementations of the `IConfCacheQueryEngine` interface are provided in the Configuration Object Model, as described below:

- DEFAULTCONFCACHEQUERYENGINE CLASS - The `DefaultConfCacheQueryEngine` class is a default implementation of the `IConfCacheQueryEngine` interface.
- COMPOSITECONFCACHEQUERYENGINE CLASS - This class is a more advanced implementation of the query engine which allows child query engine modules to be registered in order to interpret different types of queries. It does not have a default query engine implementation, only the mechanism for working with multiple child query engines.

Notification and Delta Handling

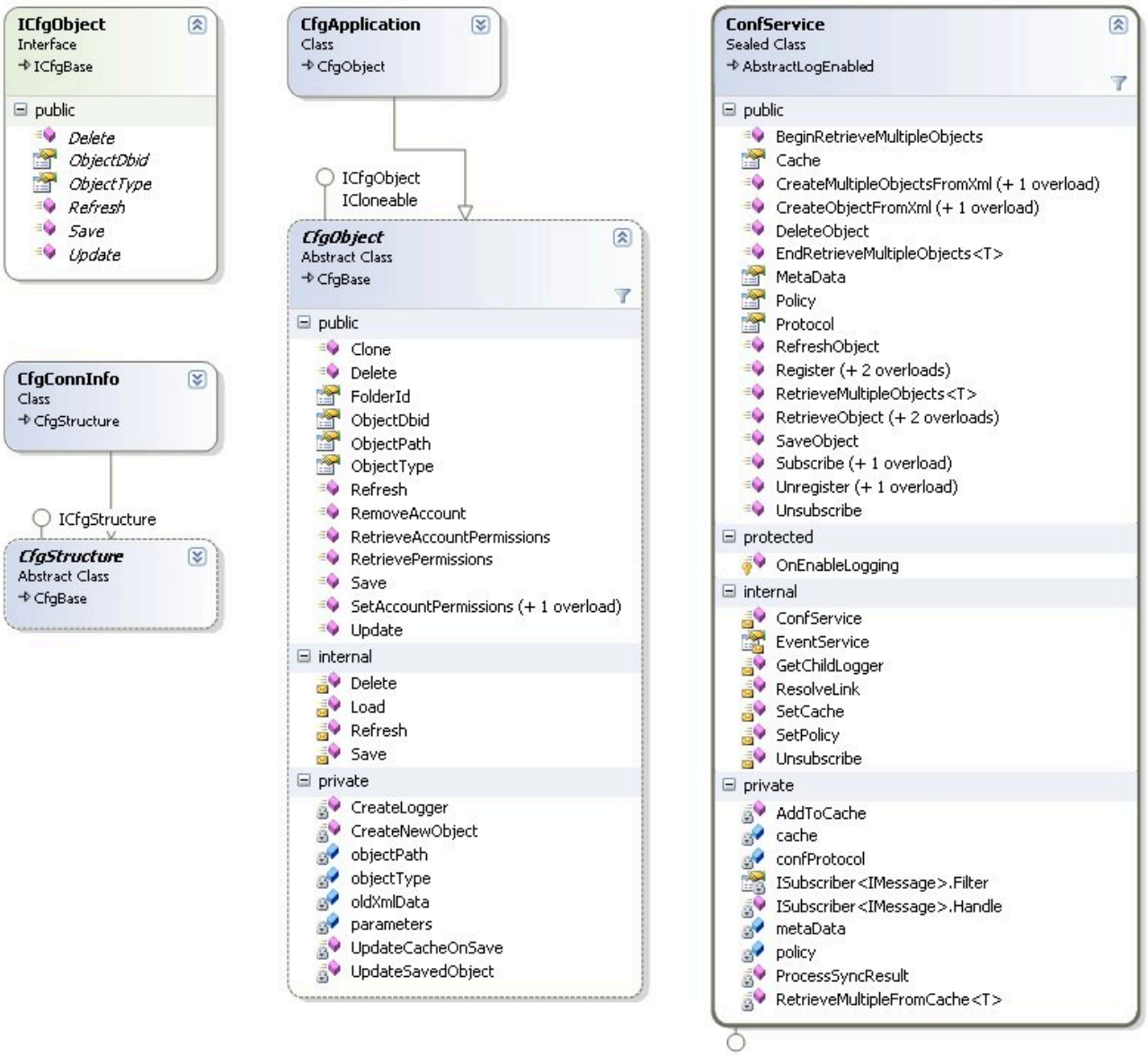
The default configuration cache will implement the `ISubscriber<ConfEvent>` interface which will allow the cache to be subscribed to receive configuration events. When a cache instance is

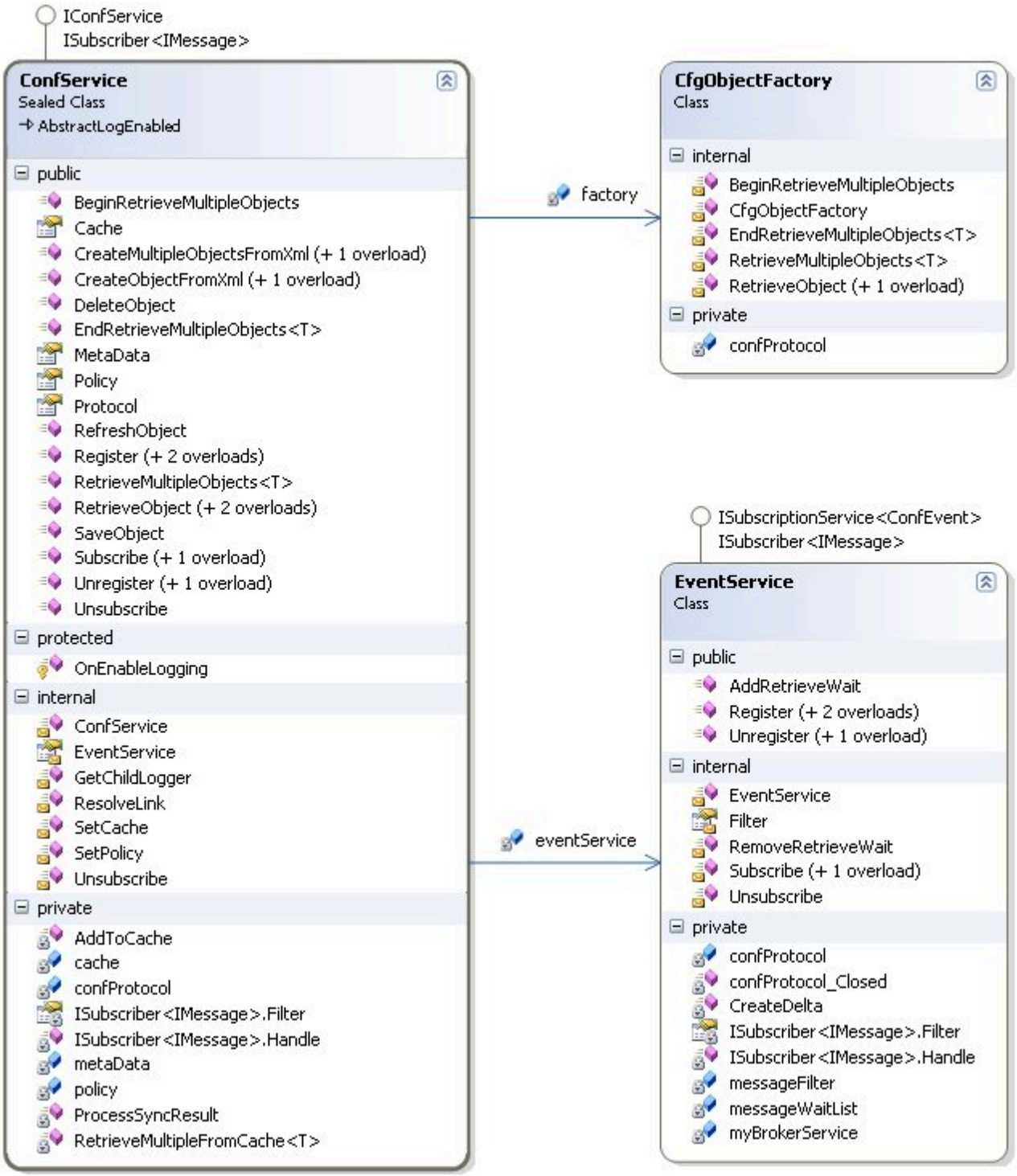
associated with a Configuration Service, it will automatically be subscribed for configuration events from that service (note that if a custom cache implementation also implements this interface it will be subscribed for events as well). The way the cache is updated based on these notifications is determined by the cache policy.

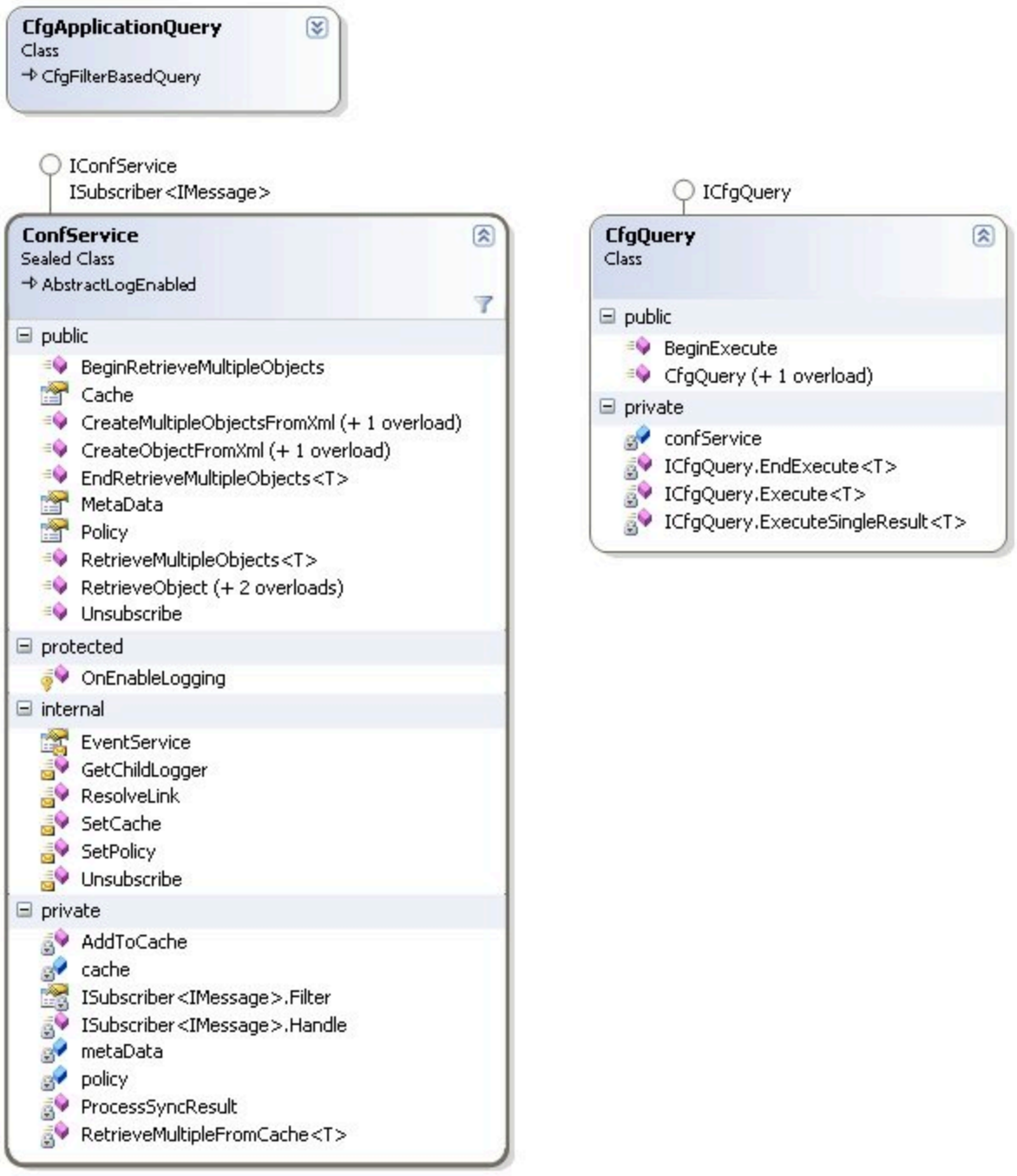
In addition, a new filter class will be added in order to allow the subscriber to filter the cache events. The `ConfCacheFilter` will implement the `MessageBroker`'s `IPredicate` interface, allowing for the filter to be passed during registration for events via `ISubscriptionService`. The `ConfCacheFilter`'s properties will specify the parameters by which the events will be filtered. Initially, the supported parameters will be object type, object DBID, and update type, allowing the user to filter events by one or a combination of these parameters assuming an AND relationship between the parameters specified.

The Configuration Object Model Application Block Interface

The following figures show the relationships among many of the classes that make up this application block.


ICfgObject
Interface
ICfgBase
public
Delete
ObjectDbid
ObjectType
Refresh
Save
Update
CfgConnInfo
Class
CfgStructure
ICfgStructure
CfgStructure
Abstract Class
CfgBase
CfgApplication
Class
CfgObject
ICfgObject
ICloneable
CfgObject
Abstract Class
CfgBase
public
Clone
Delete
FolderId
ObjectDbid
ObjectPath
ObjectType
Refresh
RemoveAccount
RetrieveAccountPermissions
RetrievePermissions
Save
SetAccountPermissions (+ 1 overload)
Update
internal
Delete
Load
Refresh
Save
private
CreateLogger
CreateNewObject
objectPath
objectType
oldXmlData
parameters
UpdateCacheOnSave
UpdateSavedObject
ConfService
Sealed Class
AbstractLogEnabled
public
BeginRetrieveMultipleObjects
Cache
CreateMultipleObjectsFromXml (+ 1 overload)
CreateObjectFromXml (+ 1 overload)
DeleteObject
EndRetrieveMultipleObjects<T>
MetaData
Policy
Protocol
RefreshObject
Register (+ 2 overloads)
RetrieveMultipleObjects<T>
RetrieveObject (+ 2 overloads)
SaveObject
Subscribe (+ 1 overload)
Unregister (+ 1 overload)
Unsubscribe
protected
OnEnableLogging
internal
ConfService
EventService
GetChildLogger
ResolveLink
SetCache
SetPolicy
Unsubscribe
private
AddToCache
cache
confProtocol
ISubscriber<IMessage>.Filter
ISubscriber<IMessage>.Handle
metaData
policy
ProcessSyncResult
RetrieveMultipleFromCache<T>


IConfService
ISubscriber<IMessage>
ConfService
Sealed Class
AbstractLogEnabled
public
BeginRetrieveMultipleObjects
Cache
CreateMultipleObjectsFromXml (+ 1 overload)
CreateObjectFromXml (+ 1 overload)
DeleteObject
EndRetrieveMultipleObjects<T>
MetaData
Policy
Protocol
RefreshObject
Register (+ 2 overloads)
RetrieveMultipleObjects<T>
RetrieveObject (+ 2 overloads)
SaveObject
Subscribe (+ 1 overload)
Unregister (+ 1 overload)
Unsubscribe
protected
OnEnableLogging
internal
ConfService
EventService
GetChildLogger
ResolveLink
SetCache
SetPolicy
Unsubscribe
private
AddToCache
cache
confProtocol
ISubscriber<IMessage>.Filter
ISubscriber<IMessage>.Handle
metaData
policy
ProcessSyncResult
RetrieveMultipleFromCache<T>
factory
CfgObjectFactory
Class
internal
BeginRetrieveMultipleObjects
CfgObjectFactory
EndRetrieveMultipleObjects<T>
RetrieveMultipleObjects<T>
RetrieveObject (+ 1 overload)
private
confProtocol
eventService
ISubscriptionService<ConfEvent>
ISubscriber<IMessage>
EventService
Class
public
AddRetrieveWait
Register (+ 2 overloads)
Unregister (+ 1 overload)
internal
EventService
Filter
RemoveRetrieveWait
Subscribe (+ 1 overload)
Unsubscribe
private
confProtocol
confProtocol_Closed
CreateDelta
ISubscriber<IMessage>.Filter
ISubscriber<IMessage>.Handle
messageFilter
messageWaitList
myBrokerService


CfgApplicationQuery
Class
CfgFilterBasedQuery
IConfService
ISubscriber<IMessage>
ConfService
Sealed Class
AbstractLogEnabled
public
BeginRetrieveMultipleObjects
Cache
CreateMultipleObjectsFromXml (+ 1 overload)
CreateObjectFromXml (+ 1 overload)
EndRetrieveMultipleObjects<T>
MetaData
Policy
RetrieveMultipleObjects<T>
RetrieveObject (+ 2 overloads)
Unsubscribe
protected
OnEnableLogging
internal
EventService
GetChildLogger
ResolveLink
SetCache
SetPolicy
Unsubscribe
private
AddToCache
cache
ISubscriber<IMessage>.Filter
ISubscriber<IMessage>.Handle
metaData
policy
ProcessSyncResult
RetrieveMultipleFromCache<T>
ICfgQuery
CfgQuery
Class
public
BeginExecute
CfgQuery (+ 1 overload)
private
confService
ICfgQuery.EndExecute<T>
ICfgQuery.Execute<T>
ICfgQuery.ExecuteSingleResult<T>

Using the Application Block

Installing the Configuration Object Model Application Block

Before you install the Configuration Object Model Application Block, it is important to review the software requirements established in the Genesys Supported Operating Environment Reference Manual.

Configuring the Configuration Object Model Application Block

In order to use the QuickStart application, you will need to set up the XML configuration file that comes with the application block. This file is located at `Quickstart\app.config`. This is what the contents look like:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Uri" value="tcp://yourhost:yourport"/>

    <add key="ClientName" value="StarterApp"/>

    <add key="ClientType" value="CFGAgentDesktop"/>

    <add key="UserName" value="default"/>

    <add key="Password" value="password"/>

  </appSettings>
</configuration>
```

Follow the instructions in the comments and save the file.

Building the Configuration Object Model Application Block

> **Tip**
>
> Starting with release 8.5.0, the common interfaces for COM Application Block and Message Broker have been moved to an individual `Genesyslab.Platform.ApplicationBlocks.Commons.dll` file.

The Platform SDK distribution includes a `Genesyslab.Platform.ApplicationBlocks.Commons.dll` file that you can use as is. This file is located in the `bin` directory at the root level of the Platform SDK directory. To build your own copy of this application block, follow the instructions below:

To build the Configuration Object Model Application Block:

1. Open the <Platform SDK Folder>\ApplicationBlocks\Com folder.
2. Double-click Com.sln.
3. Build the solution.

Using the QuickStart Application

The easiest way to start using the Configuration Object Model Application Block is to use the bundled QuickStart application. This application ships in the same folder as the application block.

To run the QuickStart application:

1. Open the <Platform SDK Folder>\ApplicationBlocks\Com folder.
2. Double-click ComQuickStart.sln.
3. Build the solution.
4. Find the executable for the QuickStart application, which will be at <Platform SDK Folder>\ApplicationBlocks\Com\QuickStart\bin\Debug\ComQuickStart.exe.
5. Double-click ComQuickStart.exe.

Editing Capacity Rules

The Configuration Object Model Application Block includes the CapacityRuleHelper class (introduced in release 8.1.4) which allows you to edit and update Capacity Rules. This helper class presents an XML representation for CfgScript objects of type CapacityRule, which can be updated and saved to edit existing Capacity Rules.

An example of how to edit capacity rules is provided below.

```
IConfService service = ConfServiceFactory.CreateConfService(protocol);
service.Protocol.Open();
CfgScriptQuery query = new CfgScriptQuery(service);
CfgScript script = service.RetrieveObject(query);
```

An instance of the CapacityRuleHelper class can now be created with static method Create. This method validates the input script object and can throw an exception (CapacityRuleException) if the object is not valid. Once the instance is created, the XMLPresentation property allows you access to Capacity Rules.

```
CapacityRuleHelper helper = CapacityRuleHelper.Create(script);
Document doc = helper.XMLPresentation;
// edit xml document here
```

The XMLPresentation property is able to be saved into the CapacityRuleHelper class instance. Once changes have been made to the XML document, apply your changes using the following code:

```
helper.XMLPresentation = doc;
helper.Script.Save();
service.Protocol.Close();
ConfServiceFactory.ReleaseConfService(service);
```