

cosEventDomain

Copyright © 2001-2025 Ericsson AB. All Rights Reserved.
cosEventDomain 1.2.2
July 16, 2025

Copyright © 2001-2025 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

July 16, 2025

1 cosEventDomain User's Guide

The **cosEventDomain** application is an Erlang implementation of a CORBA Service CosEventDomainAdmin.

1.1 The cosEventDomain Application

1.1.1 Content Overview

The cosEventDomain documentation is divided into three sections:

- PART ONE - The User's Guide
  Description of the cosEventDomain Application including services and a small tutorial demonstrating the development of a simple service.
- PART TWO - Release Notes
  A concise history of cosEventDomain.
- PART THREE - The Reference Manual
  A quick reference guide, including a brief description, to all the functions available in cosEventDomain.

1.1.2 Brief Description of the User's Guide

The User's Guide contains the following parts:

- CosEventDomain overview
- CosEventDomain installation and examples

1.2 Introduction to cosEventDomain

1.2.1 Overview

The cosEventDomain application is a Event Domain Service compliant with the **OMG** Service CosEventDomainAdmin.

Purpose and Dependencies

CosEventDomain is dependent on **Orber**, which provides CORBA functionality in an Erlang environment.

Prerequisites

To fully understand the concepts presented in the documentation, it is recommended that the user is familiar with distributed programming and CORBA.

1.3 Quality Of Service and Admin Properties

1.3.1 Quality Of Service and Admin Properties

This chapter explains the allowed properties it is possible to set for this application.

Quality Of Service

The cosEventDomain application supports the following QoS settings:

QoS	Range	Default
CycleDetection	AuthorizeCycles/ForbidCycles	ForbidCycles
DiamondDetection	AuthorizeDiamonds/ ForbidDiamonds	ForbidDiamonds

Table 3.1: Supported QoS settings

Comments on the table 'Supported QoS Settings':

CycleDetection
If a cycle is created, the user **must** be aware of the fact that unless they set timeout on events, events that are not filtered will loop endlessly through the topology.

DiamondDetection
A Diamond in this context, means that the same event may reach a point in the graph by more than one route (i.e. transitive). Hence, it is possible that multiple copies are delivered.

Setting Quality Of Service

Assume we have a Consumer Admin object which we want to change the current Quality of Service. Typical usage:

```
QoS =
   [#'CosNotification_Property'
    {name='CosEventDomainAdmin':'DiamondDetection'(),
     value=any:create(orber_tc:short(),
     'CosEventDomainAdmin':'AuthorizeDiamonds'())},
    #'CosNotification_Property'
    {name='CosEventDomainAdmin':'CycleDetection'(),
     value=any:create(orber_tc:short(),
     'CosEventDomainAdmin':'ForbidCycles'())}],
'CosEventDomainAdmin_EventDomain':set_qos(ED, QoS),
```

If it is not possible to set the requested QoS the `UnsupportedQoS` exception is raised, which includes a sequence of `PropertyError`'s describing which QoS, possible range and why is not allowed. The error codes are:

- UNSUPPORTED_PROPERTY - QoS not supported for this type of target object.
- UNAVAILABLE_PROPERTY - due to current QoS settings the given property is not allowed.
- UNSUPPORTED_VALUE - property value out of range; valid range is returned.
- UNAVAILABLE_VALUE - due to current QoS settings the given value is not allowed; valid range is returned.
- BAD_PROPERTY - unrecognized property.
- BAD_TYPE - type of supplied property is incorrect.
- BAD_VALUE - illegal value.

The CosEventDomainAdmin_EventDomain interface also supports an operation called `validate_qos/2`. The purpose of this operations is to check if a QoS setting is supported by the target object and if so, the operation returns additional properties which could be optionally added as well.

Admin Properties

The OMG specification do not contain any definitions of Admin Properties. Hence, the cosEventDomain application currently does not support any Admin Properties.

1.4 Event Domain Service

1.4.1 Overview of the CosEventDomain Service

The Event Domain service allows programmers to manage a cluster of information channels.

Event Domain Service Components

There are two components in the OMG CosEventDomainAdmin service architecture:

- **EventDomainFactory:** a factory for creating EventDomains.
- **EventDomain:** supplies a tool, which makes it easy to create topologies of interconnected channels (i.e. a directed graph).

A Tutorial on How to Create a Simple Service

To be able to use the cosEventDomain application, the cosNotification and, possibly, the cosTime application must be installed.

How to Run Everything

Below is a short transcript on how to run cosEventDomain.

```
%% Start Mnesia and Orber
mnesia:delete_schema([node()]),
mnesia:create_schema([node()]),
orber:install([node()]),
mnesia:start(),
orber:start(),

%% Install and start cosNotification.
cosNotificationApp:install(),
cosNotificationApp:start(),

%% Install and start cosEventDomain.
cosEventDomainApp:install(),
cosEventDomainApp:start(),

%% Start a CosEventDomainAdmin factory.
AdminFac = cosEventDomainApp:start_factory(),

%% Define the desired QoS settings:
QoS =
   [#'CosNotification_Property'
    {name='CosEventDomainAdmin':'DiamondDetection'(),
     value=any:create(orber_tc:short(),
     'CosEventDomainAdmin':'AuthorizeDiamonds'())},
    #'CosNotification_Property'
    {name='CosEventDomainAdmin':'CycleDetection'(),
     value=any:create(orber_tc:short(),
     'CosEventDomainAdmin':'ForbidCycles'())}],

%% Create a new EventDomain:
{ED, EDId} = 'CosEventDomainAdmin_EventDomainFactory':
                        create_event_domain(Fac, QoS, []),

%% Now we can add Notification Channels to the Domain. How this
%% is done, see the cosNotification documentation. Let us assume
%% that we have gained access to two Channel Objects; add them to the
%% domain:
ID1 = 'CosEventDomainAdmin_EventDomain':add_channel(ED, Ch1),
ID2 = 'CosEventDomainAdmin_EventDomain':add_channel(ED, Ch2),

%% To connect them, we must first define a connection struct:
C1 = #'CosEventDomainAdmin_Connection'{supplier_id=ID1,
                                       consumer_id=ID2,
                                       ctype='STRUCTURED_EVENT',
                                       notification_style='Pull'},

%% Connect them:
'CosEventDomainAdmin_EventDomain':add_connection(ED, C1),
```

2 Reference Manual

The **cosEventDomain** application is an Erlang implementation of a CORBA Service CosEventDomainAdmin.

cosEventDomainApp

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosEventDomain/include/*.hrl").`

This module contains the functions for starting and stopping the application.

Exports

`install() -> Return`

Types:

`Return = ok | {'EXCEPTION', E} | {'EXIT', R}`

This operation installs the cosEventDomain application.

`uninstall() -> Return`

Types:

`Return = ok | {'EXCEPTION', E} | {'EXIT', R}`

This operation uninstalls the cosEventDomain application.

`start() -> Return`

Types:

`Return = ok | {error, Reason}`

This operation starts the cosEventDomain application.

`stop() -> Return`

Types:

`Return = ok | {error, Reason}`

This operation stops the cosEventDomain application.

`start_factory() -> Factory`

Types:

`Factory = #objref`

This operation creates a new instance of a *Event Domain Factory* using the default settings.

`start_factory(Options) -> Factory`

Types:

`Options = [Option]`
`Option = currently no options defined.`
`Factory = #objref`

This operation creates a new instance of a *Event Domain Factory*

`start_factory_link() -> Factory`

Types:

`Factory = #objref`

This operation creates a new instance of a *Event Domain Factory*, which is linked to the invoking process, using the default settings.

`start_factory_link(Options) -> Factory`

Types:

`Options = [Option]`
`Option = currently no options defined.`
`Factory = #objref`

This operation creates a new instance of a *Event Domain Factory*, which is linked to the invoking process, with settings defined by the given options. Allowed options are the same as for `cosEventDomainApp:start_factory/1`.

`stop_factory(Factory) -> Reply`

Types:

`Factory = #objref`
`Reply = ok | {'EXCEPTION', E}`

This operation stop the target factory.

CosEventDomainAdmin

Erlang module

To get access to all definitions include necessary `hrl` files by using:
`-include_lib("cosEventDomain/include/*.hrl").`

Exports

`'CycleDetection'() -> string()`

This function returns the CycleDetection identifier; required when defining QoS Properties.

`'AuthorizeCycles'() -> short()`

This function returns the AuthorizeCycles value; required when defining QoS Properties.

`'ForbidCycles'() -> short()`

This function returns the ForbidCycles value; required when defining QoS Properties.

`'DiamondDetection'() -> string()`

This function returns the DiamondDetection identifier; required when defining QoS Properties.

`'AuthorizeDiamonds'() -> short()`

This function returns the AuthorizeDiamonds value; required when defining QoS Properties.

`'ForbidDiamonds'() -> short()`

This function returns the ForbidDiamonds value; required when defining QoS Properties.

CosEventDomainAdmin_EventDomainFactory

Erlang module

To get access to all definitions include necessary `hrl` files by using:
`-include_lib("cosEventDomain/include/*.hrl").`

Exports

`create_event_domain(Factory, QoS, Admin) -> Reply`

Types:

```
Factory = #objref
QoS = CosNotification::QoSProperties
Admin = CosNotification::AdminProperties
Reply = {EventDomain, DomainID} | {'EXCEPTION',
#'CosNotification_UnsupportedQoS'{}} | {'EXCEPTION',
#'CosNotification_UnsupportedAdmin'{}}
EventDomain = #objref
```

To create a new EventDomain this operation is used. If it is not possible to support the given `QoSProperties` or `AdminProperties` an exception is raised, which list the properties not supported. For more information see the `cosNotification` user's guide.

`get_all_domains(Factory) -> DomainIDSeq`

Types:

```
Factory = #objref
DomainIDSeq = [long()]
```

This function returns a DomainID sequence of all domains associated with the target object.

`get_event_domain(Factory, DomainID) -> Reply`

Types:

```
Factory = #objref
DomainID = long()
Reply = EventDomain | {'EXCEPTION',
#'CosEventDomainAdmin_DomainNotFound'{}}
EventDomain = #objref
```

This operation returns the EventDomain object associated with the given DomainID. If no such binding exists an exception is raised.

CosEventDomainAdmin_EventDomain

Erlang module

To get access to all definitions include necessary `hrl` files by using:
`-include_lib("cosEventDomain/include/*.hrl").`

This module also exports the functions described in:

- **CosNotification_QoSAdmin**
- **CosNotification_AdminPropertiesAdmin**

Exports

`add_channel(EventDomain, Channel) -> MemberID`

Types:

`EventDomain = Channel = #objref`
`MemberID = long()`

Adds the given channel to the target domain. The channel must be a `CosNotifyChannelAdmin::EventChannel`.

`get_all_channels(EventDomain) -> MemberIDSeq`

Types:

`EventDomain = #objref`
`MemberIDSeq = [long()]`

Returns a a sequence of all channels associated with the target object.

`get_channel(EventDomain, MemberID) -> Reply`

Types:

`EventDomain = #objref`
`MemberID = long()`
`Reply = Channel | {'EXCEPTION',`
`#'CosNotifyChannelAdmin_ChannelNotFound'{}}`
`Channel = #objref`

If the target domain have a `CosNotifyChannelAdmin::EventChannel` represented by the given id this channel is returned. Otherwise, an exception is raised.

`remove_channel(EventDomain, MemberID) -> Reply`

Types:

`EventDomain = #objref`
`MemberID = long()`
`Reply = ok | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}`

If a `CosNotifyChannelAdmin::EventChannel` with the `MemberID` exists it will removed and all its `Connections` terminated. Otherwise an exception is raised.

`add_connection(EventDomain, Connection) -> Reply`

Types:

```
EventDomain = #objref
Connection = 'CosEventDomainAdmin_Connection'{supplier_id=MemberID,
consumer_id=MemberID, ctype=Type, notification_style=Style}
MemberID = long()
Type = 'ANY_EVENT' | 'STRUCTURED_EVENT' | 'SEQUENCE_EVENT'
Style = 'Pull' | 'Push'
Reply = ConnectionID | {'EXCEPTION', Exc}
ConnectionID = long()
Exc = #'CosNotifyChannelAdmin_ChannelNotFound'{} |
#'CosNotifyChannelAdmin_TypeError'{} |
#'CosEventDomainAdmin_AlreadyExists'{} |
#'CosEventDomainAdmin_DiamondCreationForbidden'{diam=RouteSeq} |
#'CosEventDomainAdmin_CycleCreationForbidden'{cyc=MemberIDSeq}
RouteSeq = [MemberIDSeq]
MemberIDSeq = [long()]
```

The Connection parameter must contain valid data to enable the target domain to setup a connection between two channels. The struct members `supplier_id` and `consumer_id` determines which channel should produce and consume events. which type of events and if the supplier should push or the consumer pull events is determined by `ctype` and `notification_style` respectively.

If the target domain is not able to setup the connection the appropriate exception is raised.

`get_all_connections(EventDomain) -> ConnectionIDSeq`

Types:

```
EventDomain = #objref
ConnectionIDSeq = [long()]
```

This operation returns a sequence of all connections within the target domain.

`get_connection(EventDomain, ConnectionID) -> Reply`

Types:

```
EventDomain = #objref
ConnectionID = long()
Reply = Connection | {'EXCEPTION',
#'CosEventDomainAdmin_ConnectionNotFound'{}}
Connection = 'CosEventDomainAdmin_Connection'{supplier_id=MemberID,
consumer_id=MemberID, ctype=Type, notification_style=Style}
MemberID = long()
Type = 'ANY_EVENT' | 'STRUCTURED_EVENT' | 'SEQUENCE_EVENT'
Style = 'Pull' | 'Push'
```

If a connection identified by the given id exists within the target domain, a `#'CosEventDomainAdmin_Connection'{}` which describe the connection is returned. Otherwise, an exception is raised.

`remove_connection(EventDomain, ConnectionID) -> Reply`

Types:

EventDomain = #objref
ConnectionID = long()
Reply = ok | {'EXCEPTION', #'CosEventDomainAdmin_ConnectionNotFound'{}}

If the supplied connection id exists, the connection the id represents is terminated. Otherwise, an exception is raised.

`get_offer_channels(EventDomain, MemberID) -> Reply`

Types:

EventDomain = #objref
MemberID = long()
Reply = MemberIDSeq | {'EXCEPTION',
#'CosNotifyChannelAdmin_ChannelNotFound'{}}

This operation returns a sequence, containing the member id's of all channels within the target domain which will supply events to the channel identified by the given id. But, if no such id exists in this domain, an exception is raised.

`get_subscription_channels(EventDomain, MemberID) -> Reply`

Types:

EventDomain = #objref
Reply = MemberIDSeq | {'EXCEPTION',
#'CosNotifyChannelAdmin_ChannelNotFound'{}}

This operations behaves like `get_subscription_channels`; the difference is that the id's returned identifies channels which will consume events supplied by the channel associated with the given id.

`destroy(EventDomain) -> ok`

Types:

EventDomain = #objref

Calling this operation will terminate all connections within the target domain. The domain will terminate but all channels will not be affected.

`get_cycles(EventDomain) -> RouteSeq`

Types:

EventDomain = #objref
RouteSeq = [MemberIDSeq]
MemberIDSeq = [long()]

Returns a list of all cycles within the target domain.

`get_diamonds(EventDomain) -> DiamondSeq`

Types:

EventDomain = #objref
DiamondSeq = [RouteSeq]
RouteSeq = [MemberIDSeq]
MemberIDSeq = [long()]

Returns a list of all diamonds within the target domain

`set_default_consumer_channel(EventDomain, MemberID) -> Reply`

Types:

EventDomain = #objref

Reply = MemberID | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

MemberID = long()

If the given id represents a channel within the target domain, this channel will be used when connection a supplier client without specifying a certain channel. If no such channel exists an exceptions is raised.

`set_default_supplier_channel(EventDomain, MemberID) -> Reply`

Types:

EventDomain = #objref

Reply = MemberID | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

MemberID = long()

If the given id represents a channel within the target domain, this channel will be used when connection a consumer client without specifying a certain channel. If no such channel exists an exceptions is raised.

`connect_push_consumer(EventDomain, Consumer) -> Reply`

Types:

EventDomain = #objref

Consumer = CosEventComm::PushConsumer

Reply = CosNotifyChannelAdmin::ProxyPushSupplier | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a default Channel have been set, this operation connects the given PushConsumer to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_pull_consumer(EventDomain, Consumer) -> Reply`

Types:

EventDomain = #objref

Consumer = CosEventComm::PullConsumer

Reply = CosNotifyChannelAdmin::ProxyPullSupplier | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a default Channel have been set, this operation connects the given PullConsumer to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_push_supplier(EventDomain, Supplier) -> Reply`

Types:

EventDomain = #objref

Supplier = CosEventComm::PushSupplier

Reply = CosNotifyChannelAdmin::ProxyPushConsumer | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a default Channel have been set, this operation connects the given PushSupplier to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_pull_supplier(EventDomain, Supplier) -> Reply`

Types:

`EventDomain = #objref`

`Supplier = CosEventComm::PullSupplier`

`Reply = CosNotifyChannelAdmin::ProxyPushConsumer | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}`

If a default Channel have been set, this operation connects the given PullSupplier to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_structured_push_consumer(EventDomain, Consumer) -> Reply`

Types:

`EventDomain = #objref`

`Consumer = CosNotifyComm::StructuredPushConsumer`

`Reply = CosNotifyChannelAdmin::StructuredProxyPushSupplier | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}`

If a default Channel have been set, this operation connects the given StructuredPushConsumer to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_structured_pull_consumer(EventDomain, Consumer) -> Reply`

Types:

`EventDomain = #objref`

`Consumer = CosNotifyComm::StructuredPullConsumer`

`Reply = CosNotifyChannelAdmin::StructuredProxyPullSupplier | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}`

If a default Channel have been set, this operation connects the given StructuredPullConsumer to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_structured_push_supplier(EventDomain, Supplier) -> Reply`

Types:

`EventDomain = #objref`

`Supplier = CosNotifyComm::StructuredPushSupplier`

`Reply = CosNotifyChannelAdmin::StructuredProxyPushConsumer | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}`

If a default Channel have been set, this operation connects the given StructuredPushSupplier to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_structured_pull_supplier(EventDomain, Supplier) -> Reply`

Types:

`EventDomain = #objref`

`Supplier = CosNotifyComm::StructuredPullSupplier`

`Reply = CosNotifyChannelAdmin::StructuredProxyPullConsume | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}`

If a default Channel have been set, this operation connects the given StructuredPullSupplier to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_sequence_push_consumer(EventDomain, Consumer) -> Reply`

Types:

EventDomain = #objref

Consumer = CosNotifyComm::SequencePushConsumer

Reply = CosNotifyChannelAdmin::SequenceProxyPushSupplier | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a default Channel have been set, this operation connects the given SequencePushConsumer to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_sequence_pull_consumer(EventDomain, Consumer) -> Reply`

Types:

EventDomain = #objref

Consumer = CosNotifyComm::SequencePullConsumer

Reply = CosNotifyChannelAdmin::SequenceProxyPullSupplier | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a default Channel have been set, this operation connects the given SequencePullConsumer to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_sequence_push_supplier(EventDomain, Supplier) -> Reply`

Types:

EventDomain = #objref

Supplier = CosNotifyComm::SequencePushSupplier

Reply = CosNotifyChannelAdmin::SequenceProxyPushConsumer | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a default Channel have been set, this operation connects the given SequencePushSupplier to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_sequence_pull_supplier(EventDomain, Supplier) -> Reply`

Types:

EventDomain = #objref

Supplier = CosNotifyComm::SequencePullSupplier

Reply = CosNotifyChannelAdmin::SequenceProxyPullConsumer | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a default Channel have been set, this operation connects the given SequencePullSupplier to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_push_consumer_with_id(EventDomain, Consumer, MemberID) -> Reply`

Types:

EventDomain = #objref

Consumer = CosEventComm::PushConsumer

MemberID = long()

Reply = CosNotifyChannelAdmin::ProxyPushSupplier | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given PushConsumer to it. Otherwise, the #'CosNotifyChannelAdmin_ChannelNotFound'{} exception is raised.

```
connect_pull_consumer_with_id(EventDomain, Consumer, MemberID) -> Reply
```

Types:

```
EventDomain = #objref
Consumer = CosEventComm::PullConsumer
MemberID = long()
Reply = CosNotifyChannelAdmin::ProxyPullSupplier | {'EXCEPTION',
#'CosNotifyChannelAdmin_ChannelNotFound'{}}
```

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given PullConsumer to it. Otherwise, the #'CosNotifyChannelAdmin_ChannelNotFound'{} exception is raised.

```
connect_push_supplier_with_id(EventDomain, Supplier, MemberID) -> Reply
```

Types:

```
EventDomain = #objref
Supplier = CosEventComm::PushSupplier
MemberID = long()
Reply = CosNotifyChannelAdmin::ProxyPushConsumer | {'EXCEPTION',
#'CosNotifyChannelAdmin_ChannelNotFound'{}}
```

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given PushSupplier to it. Otherwise, the #'CosNotifyChannelAdmin_ChannelNotFound'{} exception is raised.

```
connect_pull_supplier_with_id(EventDomain, Supplier, MemberID) -> Reply
```

Types:

```
EventDomain = #objref
Supplier = CosEventComm::PullSupplier
MemberID = long()
Reply = CosNotifyChannelAdmin::ProxyPushConsumer | {'EXCEPTION',
#'CosNotifyChannelAdmin_ChannelNotFound'{}}
```

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given PullSupplier to it. Otherwise, the #'CosNotifyChannelAdmin_ChannelNotFound'{} exception is raised.

```
connect_structured_push_consumer_with_id(EventDomain, Consumer, MemberID) ->
Reply
```

Types:

```
EventDomain = #objref
Consumer = CosNotifyComm::StructuredPushConsumer
MemberID = long()
Reply = CosNotifyChannelAdmin::StructuredProxyPushSupplier | {'EXCEPTION',
#'CosNotifyChannelAdmin_ChannelNotFound'{}}
```

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given StructuredPushConsumer to it. Otherwise, the #'CosNotifyChannelAdmin_ChannelNotFound'{} exception is raised.

`connect_structured_pull_consumer_with_id(EventDomain, Consumer, MemberID) -> Reply`

Types:

EventDomain = #objref

Consumer = CosNotifyComm::StructuredPullConsumer

MemberID = long()

Reply = CosNotifyChannelAdmin::StructuredProxyPullSupplier | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given StructuredPullConsumer to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_structured_push_supplier_with_id(EventDomain, Supplier, MemberID) -> Reply`

Types:

EventDomain = #objref

Supplier = CosNotifyComm::StructuredPushSupplier

MemberID = long()

Reply = CosNotifyChannelAdmin::StructuredProxyPushConsumer | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given StructuredPushSupplier to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_structured_pull_supplier_with_id(EventDomain, Supplier, MemberID) -> Reply`

Types:

EventDomain = #objref

Supplier = CosNotifyComm::StructuredPullSupplier

MemberID = long()

Reply = CosNotifyChannelAdmin::StructuredProxyPullConsumer | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given StructuredPullSupplier to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_sequence_push_consumer_with_id(EventDomain, Consumer, MemberID) -> Reply`

Types:

EventDomain = #objref

Consumer = CosNotifyComm::SequencePushConsumer

MemberID = long()

Reply = CosNotifyChannelAdmin::SequenceProxyPushSupplier | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given SequencePushConsumer to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_sequence_pull_consumer_with_id(EventDomain, Consumer, MemberID) -> Reply`

Types:

`EventDomain = #objref`

`Consumer = CosNotifyComm::SequencePullConsumer`

`MemberID = long()`

`Reply = CosNotifyChannelAdmin::SequenceProxyPullSupplier | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}`

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given SequencePullConsumer to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_sequence_push_supplier_with_id(EventDomain, Supplier, MemberID) -> Reply`

Types:

`EventDomain = #objref`

`Supplier = CosNotifyComm::SequencePushSupplier`

`MemberID = long()`

`Reply = CosNotifyChannelAdmin::SequenceProxyPushConsumer | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}`

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given SequencePushSupplier to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.

`connect_sequence_pull_supplier_with_id(EventDomain, Supplier, MemberID) -> Reply`

Types:

`EventDomain = #objref`

`Supplier = CosNotifyComm::SequencePullSupplier`

`MemberID = long()`

`Reply = CosNotifyChannelAdmin::SequenceProxyPullConsumer | {'EXCEPTION', #'CosNotifyChannelAdmin_ChannelNotFound'{}}`

If a Channel associated with the given MemberID exists within the target Domain, this operation connects the given SequencePullSupplier to it. Otherwise, the `#'CosNotifyChannelAdmin_ChannelNotFound'{}` exception is raised.