

cosProperty

Copyright © 2000-2018 Ericsson AB. All Rights Reserved.
cosProperty 1.2.3
August 17, 2018

Copyright © 2000-2018 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..

August 17, 2018

1 cosProperty User's Guide

The cosProperty Application is an Erlang implementation of the OMG CORBA Property Service.

1.1 The cosProperty Application

1.1.1 Content Overview

The cosProperty documentation is divided into three sections:

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

1.1.2 Brief description of the User's Guide

The User's Guide contains the following parts:

- cosProperty overview
- cosProperty installation
- A tutorial example

1.2 Introduction to cosProperty

1.2.1 Overview

The cosProperty application is compliant with the **OMG** Service CosProperty Service.

Purpose and Dependencies

cosProperty 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, CORBA and the Orber application.

Recommended reading includes **CORBA, Fundamentals and Programming - Jon Siegel** and **Open Telecom Platform Documentation Set**. It is also helpful to have read **Concurrent Programming in Erlang**.

1.3 Installing cosProperty

1.3.1 Installation Process

This chapter describes how to install *cosProperty* in an Erlang Environment.

Preparation

Before starting the installation process for cosProperty, the application Orber must be running.

Configuration

First the cosProperty application must be installed by using `cosProperty:install()` and, if requested, `cosProperty:install_db()`, followed by `cosProperty:start()`. Now we can start the desired Factory type by using either `cosProperty:start_SetFactory()` or `cosProperty:start_SetDefFactory()`.

1.4 cosProperty Examples

1.4.1 A tutorial on how to create a simple service

Initiate the application

To use the cosProperty application Orber must be running.

How to run everything

Below is a short transcript on how to run cosProperty.

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

%% Install Property Service in the IFR.
cosProperty:install(),

%% Install Property Service in mnesia.
cosProperty:install_db(),

%% Now start the application.
cosProperty:start(),

%% To be able to create Property objects we must first a Factory
%% of our preferred type.
Fac = cosProperty:start_SetDefFactory(),

%% Now we can create a Property object.
'CosPropertyService_PropertySetDefFactory':
      create_propertysetdef(Fac),

%% Now we can create any allowed properties. There are many
%% options which are all described further in the documentation.
```

2 Reference Manual

The cosProperty Application is an Erlang implementation of the OMG CORBA Property Service.

cosProperty

Erlang module

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

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

Exports

`install() -> Return`

Types:

`Return = ok | {'EXIT', Reason}`

This operation installs the cosProperty application in the IFR.

`install_db() -> Return`

Types:

`Return = ok | {'EXIT', Reason}`

This operation installs data in mnesia necessary for running the cosProperty application.

`uninstall() -> Return`

Types:

`Return = ok | {'EXIT', Reason}`

This operation removes all data in the IFR related to the cosProperty application.

`uninstall_db() -> Return`

Types:

`Return = ok | {'EXIT', Reason}`

This operation removes all data from mnesia related to the cosProperty application.

`start() -> Return`

Types:

`Return = ok | {error, Reason}`

This operation starts the cosProperty application.

`start_SetDefFactory() -> Return`

Types:

`Return = Factory | {'EXCEPTION', E}`
`Factory = CosPropertyService::PropertySetDefFactory reference.`

This operation starts a PropertySetDef Factory.

`start_SetFactory() -> Return`

Types:

```
   Return = Factory | {'EXCEPTION', E}
   Factory = CosPropertyService::PropertySetDefFactory reference.
```

This operation starts a PropertySet Factory.

stop_SetDefFactory(Factory) -> Return

Types:

```
   Factory = CosPropertyService::PropertySetDefFactory reference.
   Return = ok | {'EXCEPTION', E}
```

This operation stops the supplied PropertySetDef Factory.

stop_SetFactory(Factory) -> Return

Types:

```
   Factory = CosPropertyService::PropertySetFactory reference.
   Return = ok | {'EXCEPTION', E}
```

This operation stops the supplied PropertySet Factory.

stop() -> Return

Types:

```
   Return = ok | {error, Reason}
```

This operation stops the cosProperty application.

CosPropertyService_PropertySetFactory

Erlang module

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

Exports

`create_propertyset(Factory) -> PropertySet`

Types:

`Factory = PropertySet = #objref`

This operation creates a new `PropertySet` with no predefined properties.

`create_constrained_propertyset(Factory, PropertyTypes, Properties) -> Reply`

Types:

`Factory = #objref`

`PropertyTypes = [CORBA::TypeCode]`

`Properties = [#'CosPropertyService_Property'{property_name = Name, property_value = Value}]`

`Name = string()`

`Value = #any`

`Reply = {'EXCEPTION', #CosPropertyService_ConstraintNotSupported{}} | PropertySet`

`PropertySet = #objref`

This operation creates a new `PropertySet` with specific constraints. PropertyTypes states allowed TypeCode's and Properties valid `CosPropertyService::Property` data.

`create_initial_propertyset(Factory, Properties) -> Reply`

Types:

`Factory = #objref`

`Properties = [#'CosPropertyService_Property'{property_name = Name, property_value = Value}]`

`Name = string()`

`Value = #any`

`Reply = {'EXCEPTION', #CosPropertyService_MultipleExceptions{exceptions = Excs}} | PropertySet`

`Excs = [#'CosPropertyService_PropertyException{reason = Reason, failing_property_name = Name}]`

`Reason = invalid_property_name | conflicting_property | property_not_found | unsupported_type_code | unsupported_property | unsupported_mode | fixed_property | read_only_property`

`PropertySet = #objref`

This operation creates a new `PropertySet` with specific initial properties.

CosPropertyService_PropertySetDefFactory

Erlang module

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

Exports

`create_propertysetdef(Factory) ->`

Types:

`Factory = PropertySetDef = #objref`

This operation creates a new `PropertySetDef` with no predefined settings.

`create_constrained_propertysetdef(Factory, PropertyTypes, PropertyDefs) -> Reply`

Types:

`Factory = PropertySetDef = #objref`

`PropertyTypes = [CORBA::TypeCode]`

`PropertyDefs = [#'CosPropertyService_PropertyDef'{property_name = Name, property_value = Value, property_mode = Mode}]`

`Name = string()`

`Value = #any`

`Mode = normal | read_only | fixed_normal | fixed_readonly | undefined`

`Reply = {'EXCEPTION', #CosPropertyService_ConstraintNotSupported{}} | PropertySetDef`

`PropertySetDef = #objref`

This operation creates a new `PropertySetDef` with specific constraints. PropertyTypes states allowed TypeCode's and PropertyDefs valid `CosPropertyService::PropertyDef` data.

`create_initial_propertysetdef(Factory, PropertyDefs) -> Reply`

Types:

`Factory = PropertySetDef = #objref`

`PropertyDefs = [#'CosPropertyService_PropertyDef'{property_name = Name, property_value = Value, property_mode = Mode}]`

`Name = string()`

`Value = #any`

`Mode = normal | read_only | fixed_normal | fixed_readonly | undefined`

`Reply = {'EXCEPTION', #CosPropertyService_MultipleExceptions{exceptions = Excs}} | PropertySetDef`

`Excs = [#'CosPropertyService_PropertyException{reason = Reason, failing_property_name = Name}]`

`Reason = invalid_property_name | conflicting_property | property_not_found | unsupported_type_code | unsupported_property | unsupported_mode | fixed_property | read_only_property`

`PropertySetDef = #objref`

This operation creates a new `PropertySetDef` with specific initial properties.

CosPropertyService_PropertySet

Erlang module

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

Exports

`define_property(PropertySet, Name, Value) -> Reply`

Types:

```
PropertySet = #objref
Name = non-empty string()
Value = #any
Reply = ok | {'EXCEPTION', #CosPropertyService_InvalidPropertyName{}}
| {'EXCEPTION', #CosPropertyService_ConflictingProperty{}} |
{'EXCEPTION', #CosPropertyService_UnsupportedTypeCode{}} | {'EXCEPTION',
#CosPropertyService_UnsupportedProperty{}} | {'EXCEPTION',
#CosPropertyService_ReadOnlyProperty{}}
```

This operation adds a new property to the given object. Depending on which initial arguments was supplied when starting the object several exceptions may be raised.

`define_properties(PropertySet, Properties) -> Reply`

Types:

```
PropertySet = #objref
Properties = [#'CosPropertyService_Property'{property_name = Name,
property_value = Value}]
Name = string()
Value = #any
Reply = ok | {'EXCEPTION',
#CosPropertyService_MultipleExceptions{exceptions = Excs}}
Excs = [#'CosPropertyService_PropertyException{reason = Reason,
failing_property_name = Name}]
Reason = invalid_property_name | conflicting_property | property_not_found
| unsupported_type_code | unsupported_property | unsupported_mode |
fixed_property | read_only_property
```

This operation adds several new properties to the given object. Depending on which initial arguments was supplied when starting the object an exceptions may be raised listing the properties failing.

`get_number_of_properties(PropertySet) -> ulong()`

Types:

```
PropertySet = #objref
```

This operation returns the number of properties associated with the target object.

get_all_property_names(PropertySet, Max) -> Reply

Types:

```
PropertySet = NamesIterator = #objref
Max = ulong()
Reply = {ok, Names, NamesIterator}
Names = [string()]
```

This operation returns up to `Max` property names. If the target object have additional associated properties they will be put in the returned Iterator, otherwise the Iterator will be a `NIL` object.

get_property_value(PropertySet, Name) -> Reply

Types:

```
PropertySet = #objref
Name = string()
Reply = #any | {'EXCEPTION', #CosPropertyService_PropertyNotFound{}} |
{'EXCEPTION', #CosPropertyService_InvalidPropertyName{}}
```

This operation returns the property value associated with given name. If no such property exists or the given name is an empty string an exception will be raised.

get_properties(PropertySet, Names) -> Reply

Types:

```
PropertySet = #objref
Names = [string()]
Reply = {boolean(), Properties}
Properties = [#'CosPropertyService_Property'{property_name = Name,
property_value = Value}]
```

This operation returns all properties associated with given names. If the boolean flag is true all properties where retrieved correctly, otherwise, all properties with the type `tk_void` was not found.

get_all_properties(PropertySet, Max) -> Reply

Types:

```
PropertySet = PropertiesIterator = #objref
Reply = {ok, Properties, PropertiesIterator}
Properties = [#'CosPropertyService_Property'{property_name = Name,
property_value = Value}]
```

This operation return a list `Max` properties or less. If more properties are associated with the target object they will be put in the `PropertiesIterator`. If the object had less than `Max` associated properties the Iterator will be a `NIL` object.

delete_property(PropertySet, Name) -> Reply

Types:

```
PropertySet = #objref
Name = string()
Reply = ok | {'EXCEPTION', #CosPropertyService_FixedProperty{}} |
{'EXCEPTION', #CosPropertyService_PropertyNotFound{}} | {'EXCEPTION',
#CosPropertyService_InvalidPropertyName{}}
```

This operation tries to delete the property with given Name. An exception which indicates why it failed is raised if so needed.

delete_properties(PropertySet, Names) -> Reply

Types:

```
PropertySet = #objref
Names = [string()]
Reply = ok | {'EXCEPTION',
#CosPropertyService_MultipleExceptions{exceptions = Excs}}
Excs = [#'CosPropertyService_PropertyException{reason = Reason,
failing_property_name = Name}]
Reason = invalid_property_name | conflicting_property | property_not_found
| unsupported_type_code | unsupported_property | unsupported_mode |
fixed_property | read_only_property
```

This operation tries to delete all given Properties. If one or more removal fails an exception is raised which describe why.

delete_all_properties(PropertySet) -> boolean()

Types:

```
PropertySet = #objref
```

This operation deletes all properties. The boolean flag, if set to false, indicates that it was not possible to remove one or more properties, e.g., may be read only.

is_property_defined(PropertySet, Name) -> Reply

Types:

```
PropertySet = #objref
Name = non-empty string()
Reply = boolean() | {'EXCEPTION',
#CosPropertyService_InvalidPropertyName{}}
```

This operation returns true if the target have an associated property with given name.

CosPropertyService_PropertySetDef

Erlang module

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

This module also exports the functions described in

CosPropertyService_PropertySet

Exports

`get_allowed_property_types(PropertySetDef) -> Reply`

Types:

```
PropertySetDef = #objref
Reply = {ok, PropertyTypes}
PropertyTypes = [CORBA::TypeCode]
```

This operation return the TypeCodes which we are allowed to use when adding new properties.

`get_allowed_properties(PropertySetDef) -> Reply`

Types:

```
PropertySetDef = #objref
Reply = {ok, PropertyDefs}
PropertyDefs = [#'CosPropertyService_PropertyDef'{property_name = Name,
property_value = Value, property_mode = Mode}]
Name = string()
Value = #any
Mode = normal | read_only | fixed_normal | fixed_readonly | undefined
```

This operation a sequence of the allowed properties we may alter; depends on which mode associated with a certain property.

`define_property_with_mode(PropertySetDef, Name, Value, Mode) -> Reply`

Types:

```
PropertySetDef = #objref
Name = non-empty string()
Value = #any
Mode = normal | read_only | fixed_normal | fixed_readonly | undefined
Reply = ok | {'EXCEPTION', #CosPropertyService_InvalidPropertyName{}}
| {'EXCEPTION', #CosPropertyService_ConflictingProperty{}}
| {'EXCEPTION', #CosPropertyService_UnsupportedTypeCode{}}
| {'EXCEPTION', #CosPropertyService_UnsupportedProperty{}} |
{'EXCEPTION', #CosPropertyService_UnsupportedMode{}} | {'EXCEPTION',
#CosPropertyService_ReadOnlyProperty{}}
```

This operation attempts to associate a new property with the target object. If we fail to do so the appropriate exception is raised.

define_properties_with_modes(PropertySetDef, PropertyDefs) -> Reply

Types:

```
PropertySetDef = #objref
PropertyDefs = [#'CosPropertyService_PropertyDef'{property_name = Name,
property_value = Value, property_mode = Mode}]
Name = string()
Value = #any
Mode = normal | read_only | fixed_normal | fixed_readonly | undefined
Reply = ok | {'EXCEPTION',
#CosPropertyService_MultipleExceptions{exceptions = Excs}}
Excs = [#'CosPropertyService_PropertyException{reason = Reason,
failing_property_name = Name}]
Reason = invalid_property_name | conflicting_property | property_not_found
| unsupported_type_code | unsupported_property | unsupported_mode |
fixed_property | read_only_property
```

This operation attempts to associate the given Property Definitions with the target object. If one or more attempts fail an exception is raised describing which properties we where not able to create.

get_property_mode(PropertySetDef, Name) -> Reply

Types:

```
PropertySetDef = #objref
Name = string()
Reply = Mode | {'EXCEPTION', #CosPropertyService_InvalidPropertyName{}} |
{'EXCEPTION', #CosPropertyService_PropertyNotFound{}}
Mode = normal | read_only | fixed_normal | fixed_readonly | undefined
```

This operation returns the type of the given property.

get_property_modes(PropertySetDef, Names) -> Reply

Types:

```
PropertySetDef = #objref
Names = [string()]
Reply = {boolean(), PropertyModes}
PropertyModes = [#'CosPropertyService_PropertyMode'{property_name = Name,
property_mode = Mode}]
Name = string()
Mode = normal | read_only | fixed_normal | fixed_readonly | undefined
```

This operation returns the modes of the listed properties. If the boolean flag is false, all properties with mode `undefined` this operation failed to comply.

set_property_mode(PropertySetDef, Name, Mode) -> Reply

Types:

```
PropertySetDef = #objref
Name = string()
Mode = normal | read_only | fixed_normal | fixed_readonly | undefined
```

```
Reply = ok | {'EXCEPTION', #CosPropertyService_InvalidPropertyName{}}
| {'EXCEPTION', #CosPropertyService_UnsupportedMode{}} | {'EXCEPTION',
#CosPropertyService_PropertyNotFound{}}
```

This operation changes the given property's mode. Return the appropriate exception if not able to fulfill the request.

```
set_property_modes(PropertySetDef, PropertyModes) -> Reply
```

Types:

```
PropertySetDef = #objref
PropertyModes = [#'CosPropertyService_PropertyMode'{property_name = Name,
property_mode = Mode}]
Name = string()
Mode = normal | read_only | fixed_normal | fixed_readonly | undefined
Reply = ok | {'EXCEPTION',
#CosPropertyService_MultipleExceptions{exceptions = Excs}}
Excs = [#'CosPropertyService_PropertyException{reason = Reason,
failing_property_name = Name}]
Reason = invalid_property_name | conflicting_property | property_not_found
| unsupported_type_code | unsupported_property | unsupported_mode |
fixed_property | read_only_property
```

This operation attempts to update the listed properties mode's. Raises an exception which describe which and why an operation failed.

CosPropertyService_PropertiesIterator

Erlang module

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

Exports

`reset(Iterator) -> ok`

Types:

`Iterator = #objref`

This operation resets the position to the first property.

`next_one(Iterator) -> Reply`

Types:

`Iterator = #objref`

`Reply = {boolean(), #'CosPropertyService_Property'{property_name = Name, property_value = Value}}`

`Name = string()`

`Value = #any`

This operation returns true . If false is returned the out parameter is a non-valid Property.

`next_n(Iterator, HowMany) -> Reply`

Types:

`Iterator = #objref`

`HowMany = long()`

`Reply = {boolean(), Properties}`

`Properties = [#'CosPropertyService_Property'{property_name = Name, property_value = Value}]`

`Name = string()`

`Value = #any`

This operation returns true if the requested number of properties can be delivered and there are additional properties. If false is returned and a sequence of max `HowMany` properties will be returned and no more properties can be delivered.

`destroy(Iterator) -> ok`

Types:

`Iterator = #objref`

This operation will terminate the Iterator and all subsequent calls will fail.

CosPropertyService_PropertyNamesIterator

Erlang module

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

Exports

`reset(Iterator) -> ok`

Types:

`Iterator = #objref`

This operation resets the position to the first property name.

`next_one(Iterator) -> Reply`

Types:

`Iterator = #objref`
`Reply = {boolean(), Name}`
`Name = string()`

This operation returns true if a Property Name exists at the current position and the out parameter is a valid Property Name. If false is returned the out parameter is a non-valid Property Name.

`next_n(Iterator, HowMany) -> Reply`

Types:

`Iterator = #objref`
`HowMany = long()`
`Reply = {boolean(), [Name]}`
`Name = string()`

This operation returns true if the requested number of Property Names can be delivered and there are additional property names. If false is returned a sequence of max `HowMany` property names will be returned and no more Property Names can be delivered.

`destroy(Iterator) -> ok`

Types:

`Iterator = #objref`

This operation will terminate the Iterator and all subsequent calls will fail.