YANG Development Kit Documentation

Release 0.2.0

Cisco Systems, Inc

June 19, 2017

Contents

CHAPTER 1

Contents:

About YDK

A Software Development Kit that provides API's that are modeled in YANG.

License

Version

Version : version-id

This ydk-cpp is generated using ydk-gen.

To check out the version of ydk-gen used to generate this ydk-cpp, use the below commands:

```
$ git clone repo-url
$ git checkout commit-id
```

Changelog

- Changelog

Getting Started

Table of Contents

Overview

The YANG Development Kit (YDK) is a Software Development Kit that provides API's that are modeled in YANG. The main goal of YDK is to reduce the learning curve of YANG data models by expressing the model semantics in an API and abstracting protocol/encoding details. YDK is composed of a core package that defines services and providers, plus one or more module bundles that are based on YANG models.

How to Install

You can install YDK-Cpp on macOS or Linux. It is not currently supported on Windows.

Quick Install

macOS

You can install the latest model packages using homebrew. This utility will manage the dependencies between YDK packages and all other sytem dependencies. First, add the third-party repository (homebrew tap) for YDK:

```
$ brew tap CiscoDevNet/ydk
```

You get a fully operational YDK environment by installing the `cisco-ios-xr` bundle which automatically installs all other YDK-related packages (`ydk`, `cisco-ios-xr`, `openconfig` and `ietf` packages):

```
$ brew install ydk-cisco-ios-xr
```

Alternatively, you can perform a partial installation. If you only want to install the `openconfig` bundle and its dependencies (`ydk` and `ietf` packages), execute:

```
$ brew install ydk-openconfig
```

If you only want to install the `ietf` bundle and its dependencies (`ydk` package), execute:

```
$ brew install ydk-ietf
```

Linux

Debian and RPM packages are coming soon. Currently, you have to install it from source (see below).

Installing from source

System Requirements

Linux

Ubuntu (Debian-based) - The following packages must be present in your system before installing YDK-Cpp:

```
$ sudo apt-get install libcurl4-openssl-dev libpcre3-dev libssh-dev libxml2-dev libxslt1-dev libtool-
```

Centos (Fedora-based) - The following packages must be present in your system before installing YDK-Cpp:

```
$ sudo yum install epel-release
$ sudo yum install libxml2-devel libxslt-devel libssh-devel libtool gcc-c++ pcre-devel cmake
```

Mac

It is recommended to install homebrew and Xcode command line tools on your system before installing YDK-Cpp:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
$ brew install curl libssh pcre xml2 cmake
$ xcode-select --install
```

Building YDK

YDK uses `cmake` as the build system of choice. To install the `core` package, execute:

```
$ ydk-cpp$ cd core/ydk
$ core$ mkdir build && cd build
$ build$ cmake .. && make
$ build$ sudo make install
```

Once you have installed the `core` package, you can install one or more model bundles. Note that some bundles have dependencies on other bundles. Those dependencies are captured in the bundle packages used for quick installation. To install the `ietf` bundle, execute:

```
$ core$ cd ../../ietf
$ ietf$ mkdir build && cd build
$ build$ cmake .. && make
$ build$ sudo make install
```

To install the `openconfig` bundle, execute:

```
$ ietf$ cd ../openconfig
$ openconfig$ mkdir build && cd build
$ build$ cmake .. && make
$ build$ sudo make install
```

To install the `cisco-ios-xr` bundle, execute:

```
$ openconfig$ cd ../cisco-ios-xr
$ cisco-ios-xr$ mkdir build && cd build
$ build$ cmake .. && make
$ build$ sudo make install
$ build$ cd ../..
```

Samples

To get started with using the YDK API, there are sample apps available in the YDK-Cpp repository. For example, to run the `bgp_create.cpp` sample, execute:

```
ydk-cpp$ cd core/samples
samples$ mkdir build && cd build
build$ cmake .. && make
build$ ./bgp_create ssh://<username>:<password>@<host-address>:<port> [-v]
```

Documentation and Support

- Numerous additional samples can be found in the YDK-Cpp samples repository
- Join the YDK community to connect with other users and with the makers of YDK

Developer Guide

Introduction

Table of Contents

YDK consists of two main components: core library, which consists of services and providers, and C++ model API, which are APIs generated based on YANG models and packaged as bundles.

Core library consists of the below:

- **Service:** Provides simple API interface to be used with the bindings and providers
- **ServiceProvider:** Provides concrete implementation that abstracts underlying protocol details (e.g. *`NetconfServiceProvider`*, which is based on the NETCONF protocol)

Applications can be written using the C++ model API in conjunction with a service and a provider.

Writing an app

In this example, we set some BGP configuration using the OpenConfig model, the CRUD (Create/Read/Update/Delete) service and the NETCONF service provider. The example in this document is a simplified version of the more complete sample that is available in `core/samples/bgp_create.cpp`. Assuming you have performed the core and bundle installations first, that more complete sample can be run with the below steps:

```
ydk-cpp$ cd core/samples
samples$ mkdir build && cd build
build$ cmake .. && make
build$ ./bgp_create ssh://<username>:<password>@<host-address>:<port> [-v]
```

What happens underneath

YDK performs the below actions when running this application:

1. Establish a session with the device and, optionally, fetch any data models which are advertised by the device
2. Encode C++ data objects to the protocol format (e.g. netconf XML payload)
3. Perform transport operation with the device and collect the response (e.g. netconf reply)
4. Decode response as C++ object and return the result to app
5. Raise C++ exceptions for any errors that occurred

Header includes

In our example YDK application, first, let us include the necessary header files

```
#include <iostream>
#include <spdlog/spdlog.h>

#include "ydk/crud_service.hpp"
#include "ydk/netconf_provider.hpp"

#include "ydk_openconfig/openconfig_bgp.hpp"

using namespace std;
using namespace ydk;
```

Service Providers

The first step in any application is to create a service provider instance. In this case, the NETCONF service provider (defined in `ydk/netconf_provider.hpp`) is responsible for mapping between the CRUD service API and the underlying manageability protocol (NETCONF RPCs).

We instantiate an instance of the service provider that creates a NETCONF session to the machine with address 10.0.0.1

```
NetconfServiceProvider provider{"10.0.0.1", "test", "test", 830};
```

Using the model APIs

After establishing the connection, we instantiate the entities and set some data. Now, create an `openconfig BGP` configuration object and set the attributes

```
// Create BGP object
auto bgp = make_unique<openconfig_bgp::Bgp>();

// Set the Global AS
bgp->global->config->as = 65001;
```

```
bgp->global->config->router_id = "1.2.3.4";

// Create a neighbor
auto neighbor = make_unique<openconfig_bgp::Bgp::Neighbors::Neighbor>();
neighbor->neighbor_address = "6.7.8.9";
neighbor->config->neighbor_address = "6.7.8.9";
neighbor->config->peer_as = 65001;
neighbor->config->local_as = 65001;
neighbor->config->peer_group = "IBGP";

// Set the parent container of the neighbor
neighbor->parent = bgp->neighbors.get();

// Add the neighbor config to the BGP neighbors list
bgp->neighbors->neighbor.push_back(move(neighbor));
```

Invoking the CRUD Service

The CRUD service provides methods to create, read, update and delete entities on a device making use of the session provided by a service provider (NETCONF in this case). In order to use the CRUD service, we need to instantiate the *CrudService* class

```
CrudService crud_service{};
```

Finally, we invoke the create method of the *CrudService* class passing in the service provider instance and our entity, `bgp`

```
try
{
  crud_service.create(provider, *bgp);
}
catch(YCPPError & e)
{
  cerr << "Error details: " << e.what() << endl;
}
```

Note if there were any errors the above API will raise an exception with the base type *YCPPError*

Logging

YDK uses the *spdlog* logging library. The logging can be enabled as follows by creating a logger called "ydk". For other options like logging the "ydk" log to a file, see the spdlog reference.

```
if(verbose)
{
  auto console = spdlog::stdout_color_mt("ydk");
}
```

Using OpenDaylight with YDK

Contents

YDK makes it easy to interact with OpenDaylight programmatically using the YANG model APIs.

Applications can be written using the C++ model API in conjunction with a service and a provider.

Writing the app

In this example, we set some BGP configuration using the Cisco IOS XR model, the *CRUD (Create/Read/Update/Delete) service* and the *OpenDaylight service provider*. The example in this document is a simplified version of the more complete sample that is available in `core/samples/bgp_xr_opendaylight.cpp`. Assuming you have performed the core and cisco-ios-xr bundle installations first, that more complete sample can be run with the below steps:

```
ydk-cpp$ cd core/samples
samples$ mkdir build && cd build
build$ cmake .. && make
build$ ./bgp_xr_opendaylight http://<username>:<password>@<host-address>:<port> [-v]
```

What happens underneath

YDK performs the below actions when running this application:

1. Establish a session with the OpenDaylight instance and fetch the details of the nodes mounted on the southbound
2. Encode C++ data objects to the protocol format (e.g. restconf JSON payload)
3. For a chosen node on the southbound, perform transport operation with the device and collect the response (e.g. restconf reply)
4. Decode response as C++ object and return the result to app
5. Raise C++ exceptions for any errors that occurred

Header includes

In our example YDK application, first, let us include the necessary header files

```
#include <iostream>
#include <spdlog/spdlog.h>

#include <ydk/crud_service.hpp>
#include <ydk/path_api.hpp>
#include <ydk/opendaylight_provider.hpp>
#include <ydk/types.hpp>

#include <ydk_cisco_ios_xr/Cisco_IOS_XR_ipv4_bgp_cfg.hpp>
```

```
#include <ydk_cisco_ios_xr/Cisco_IOS_XR_ipv4_bgp_datatypes.hpp>

 // indicate the namespaces being used (optional)
 using namespace std;
 using namespace ydk;
 using namespace Cisco_IOS_XR_ipv4_bgp_cfg;
 using namespace Cisco_IOS_XR_ipv4_bgp_datatypes;
```

OpenDaylight service provider

The first step in any application is to create a service provider instance. In this case, the OpenDaylight service provider is responsible for mapping between the CRUD service API and the underlying manageability protocol (Restconf).

We first instantiate a *Repository* using the location of the schema cache of the OpenDaylight instance. We instantiate an instance of the service provider that can communicate using Restconf with an OpenDaylight instance running at host address: `127.0.0.1` and port: `8181`

```
path::Repository repo{"/Users/home/distribution-karaf-0.5.2-Boron-SR2/cache/schema"}; // In this case
OpenDaylightServiceProvider odl_provider{repo, "127.0.0.1", "admin", "admin", 8181};
```

Using the model APIs

After establishing the connection, we instantiate the entities and set some data. Now, create an `Cisco IOS XR BGP` configuration object and set the attributes

```
// Create BGP object
auto bgp = make_unique<Bgp>();

// BGP instance
auto instance = make_unique<Bgp::Instance>();
instance->instance_name = "test";
auto instance_as = make_unique<Bgp::Instance::InstanceAs>();
instance_as->as = 65001;
auto four_byte_as = make_unique<Bgp::Instance::InstanceAs::FourByteAs>();
four_byte_as->as = 65001;
four_byte_as->bgp_running = Empty();

// global address family
auto global_af = make_unique<Bgp::Instance::InstanceAs::FourByteAs::DefaultVrf::Global::GlobalAfs::Gl
global_af->af_name = BgpAddressFamilyEnum::ipv4_unicast;
global_af->enable = Empty();
global_af->parent = four_byte_as->default_vrf->global->global_afs.get();
four_byte_as->default_vrf->global->global_afs->global_af.push_back(move(global_af));

// add the instance to the parent BGP object
four_byte_as->parent = instance_as.get();
instance_as->parent = instance.get();
instance->parent = bgp.get();
instance_as->four_byte_as.push_back(move(four_byte_as));
instance->instance_as.push_back(move(instance_as));
bgp->instance.push_back(move(instance));
```

Invoking the CRUD Service

The CRUD service provides methods to create, read, update and delete entities on a device making use of the session provided by a service provider. In order to use the CRUD service, we need to instantiate the *CrudService* class

```
CrudService crud_service{};
```

At this point we can explore the southbound device node-IDs using the function call: `provider.get_node_ids()`. Let us assume there is a XR device mounted with the node ID "xr". We can obtain the *ServiceProvider* instance corresponding to this node using the function call: `odl_provider.get_node_provider("xr")`.

Finally, we invoke the create method of the *CrudService* class passing in the service provider instance and our entity, `bgp`

```
try
{
  auto & provider = odl_provider.get_node_provider("xr");
  crud_service.create(provider, *bgp);
}
catch(YCPPError & e)
{
  cerr << "Error details: " << e.what() << endl;
}
```

Note if there were any errors the above API will raise an exception with the base type *YCPPError*

Logging

YDK uses the *spdlog* logging library. The logging can be enabled as follows by creating a logger called "ydk". For other options like logging the "ydk" log to a file, see the spdlog reference.

```
if(verbose)
{
  auto console = spdlog::stdout_color_mt("ydk");
}
```

Presence Classes

According to RFC 6020, YANG supports two styles of containers, one for organizing hierarchy, another for representing configuration data. The latter type of containers are called presence containers. For instance, the existence of a presence container `ssh` may be used to indicate whether ssh login is enabled or not.

Let us consider a class named `Conditions`, with two members `match_prefix_set` (which is a presence node) and `match_neighbor_set` (which is a non-presence node).

```
Conditions::Conditions()
  :     match_prefix_set(nullptr),
        match_neighbor_set(std::make_unique<openconfig_bgp::Bgp::Conditions::MatchNeighborSet>())
```

When instantiating the `Conditions` class, the child `match_prefix_set` will be initially assigned to a null pointer. So, in order to configure the match prefix set, the user has to initialize the `match_prefix_set` as shown below:

```
auto conditions = std::make_unique<openconfig_bgp::Bgp::Conditions>();
conditions->match_prefix_set = std::make_unique<openconfig_bgp::Bgp::Conditions::MatchPrefixSet>();
conditions->match_prefix_set = conditions.get(); //set the parent
```

Using the Path API

YDK C++ provides a new interface in the form of Path API, which can be used to write apps using a generic API, using xpath-like path expression to create and access YANG data nodes. Internally, the nodes created using the YDK model API are converted to Path API data nodes for validation and encoding to respective protocol payloads.

Encoding and Decoding

A given `DataNode` Tree can be encoded and decoded into a variety of formats using the `CodecService`.

DataNode Tree

How to create DataNode Trees

Memory management

Quirks

Accessing the Schema Tree

Traversing the hierarchy (iteration and find)

Exceptions and Error Handling

TODO

Logger

TODO

Memory

Node containment hierarchies

Best practices

Path Syntax

Full XPath notation is supported for find operations on `DataNode`(s). This XPath conforms to the YANG specification (RFC 6020 section 6.4). Some useful examples:

- Get all top-level nodes of the `module-name`

```
/module-name:*
```

- Get all the descendants of `container` (excluding `container`)

```
/module-name:container//*
```

- Get `list` instance with `key1` of value `1` and `key2` of value `2` (this can return more `list` instances if there are more keys than `key1` and `key2`)

```
/module-name:container/list[key1='1'][key2='2']
```

- Get `leaf-list` instance with the value `val`

```
/module-name:container/leaf-list[.='val']
```

- Get `aug-leaf`, which was added to `module-name` from an augment module `augment-module`

```
/module-name:container/container2/augment-module:aug-cont/aug-leaf
```

A very small subset of this full XPath is recognized by `DataNode::create`. Basically, only a relative or absolute path can be specified to identify a new data node. However, lists must be identified by all their keys and created with all of them, so for those cases predicates are allowed. Predicates must be ordered the way the keys are ordered and all the keys must be specified. Every predicate includes a single key with its value. Optionally, leaves and leaf-lists can have predicates specifying their value in the path itself. All these paths are valid XPath expressions. Example: (Relative to Root Data or `RootSchemaNode`)

```
ietf-yang-library:modules-state/module[name='ietf-yang-library'][revision=''']/conformance[.='impleme
```

Almost the same XPath is accepted by `SchemaNode` methods. The difference is that it is not used on data, but schema, which means there are no key values and only one node matches one path. In effect, lists do not have to have any predicates. If they do, they do not need to have all the keys specified and if values are included, they are ignored. Nevertheless, any such expression is still a valid XPath, but can return more nodes if executed on a data tree. Examples (all returning the same node):

```
ietf-yang-library:modules-state/module/submodules
ietf-yang-library:modules-state/module[name]/submodules
ietf-yang-library:modules-state/module[name][revision]/submodules
ietf-yang-library:modules-state/module[name='ietf-yang-library'][revision]/submodules
```

In all cases the node's prefix is specified as the name of the appropriate YANG schema. Any node can be prefixed by the module name. However, if the prefix is omitted, the module name is inherited from the previous (parent) node. It means, that the first node in the path is always supposed to have a prefix.

RPC

An `Rpc` represents an instance of the YANG RPC schema node. To invoke a rpc the user first creates an `Rpc` using the `RootSchemaNode::create_rpc()` call passing in a path with the name of the rpc. For example:

```
auto get_config = root_schema->create_rpc("ietf-netconf:get-config")
```

The input `DataNode` can be obtained using `Rpc::input()`. This can be used to populate/create the child nodes of input as per this rpc's schema. The `Rpc` is a callable that takes a single argument which is the `ServiceProvider`. To invoke the rpc do this

```
auto config = get_config(sp); /// sp is a service provider
```

The config variable above is the `DataNode` representing the output of the rpc.

ServiceProvider

A ServiceProvider extends the class `ServiceProvider` and provides an interface to obtain the root `SchemaNode` tree based on the set of `Capability`(s) supported by it.

ServiceProvider Errors

TODO

Capability

A capability is a tuple that contains:

- module-name
- revision
- set of enabled features
- set of deviations active on this module

Use the `Repository` class to instantiate a `SchemaNode` tree based on the `Capability`.

SchemaNode Tree

Talk about `SchemaNode` tree , `RootSchemaNode` tree.

Memory management of trees

DO NOT get rid of the `SchemaNode` tree if there are `DataNode` tree's referencing it.

Thread safety

Inspecting YANG meta data

Traversing the hierarchy (iterations and find)

Validation

`DataNode` tree can be validated using the `ValidationService`.

API Guide

YDK API

Service

class `Service`

Base class for YDK service

CrudService

class ydk::**CrudService** : **public** Service

CRUD Service class for supporting CRUD operations on entities.

CrudService()

bool **create**(path::ServiceProvider &*provider*, Entity &*entity*)
Create the entity.

Parameters

- **provider** – An instance of ServiceProvider
- **entity** – An instance of *Entity* class defined under a bundle

Returns true if successful, false if not

Raises YCPPError If an error has occurred

bool **update**(path::ServiceProvider &*provider*, Entity &*entity*)
Update the entity.

Parameters

- **provider** – An instance of ServiceProvider
- **entity** – An instance of *Entity* class defined under a bundle

Returns true if successful, false if not

Raises YCPPError If an error has occurred

bool **delete_**(path::ServiceProvider &*provider*, Entity &*entity*)
Delete the entity.

Parameters

- **provider** – An instance of ServiceProvider
- **entity** – An instance of *Entity* class defined under a bundle

Returns true if successful, false if not

Raises YCPPError If an error has occurred

std::shared_ptr<ydk::*Entity*> **read**(path::ServiceProvider &*provider*, Entity &*filter*)
Read the entity.

Parameters

- **provider** – An instance of ServiceProvider
- **filter** – An instance of *Entity* class defined under a bundle

Returns A pointer to an instance of *Entity* as identified by the filter if successful, nullptr if not

Raises YCPPError If an error has occurred

std::shared_ptr<ydk::*Entity*> **read_config**(path::ServiceProvider &*provider*, Entity &*filter*)
Read only config.

Parameters

- **provider** – An instance of ServiceProvider
- **filter** – An instance of *entity* class defined under a bundle

Returns A pointer to an instance of *Entity* as identified by the `filter` if successful, `nullptr` if not

Raises YCPPError If an error has occurred

NetconfService

enum class `Datastore`

Type of datastore to perform operation on

enumerator `candidate`

enumerator `running`

enumerator `startup`

enumerator `url`

class `NetconfService` : **public** Service

Netconf Service class for supporting encoding and decoding C++ model API objects of type `Entity`

`NetconfService`()
Constructs an instance of NetconfService

bool **`cancel_commit`**(NetconfServiceProvider &*provider*, std::string *persist_id* = "")
Cancels an ongoing confirmed commit. If the *persist_id* parameter is not given, the operation MUST be issued on the same session that issued the confirmed commit.

Parameters

- **`provider`** – An instance of *NetconfServiceProvider*
- **`persist_id`** – Cancels a persistent confirmed commit.

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

bool **`close_session`**(NetconfServiceProvider &*provider*)
Request graceful termination of a NETCONF session

Parameters **`provider`** – An instance of *NetconfServiceProvider*

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

bool **`commit`**(NetconfServiceProvider &*provider*, std::string *confirmed* = "", std::string *confirm_timeout* = "", std::string *persist* = "", std::string *persist_id* = "")
Instructs the device to implement the configuration data contained in the candidate configuration

Parameters

- **`provider`** – An instance of *NetconfServiceProvider*
- **`confirmed`** – An optional argument
- **`confirm_timeout`** – An optional argument
- **`persist`** – An optional argument
- **`persist_id`** – An optional argument

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

bool **copy_config**(NetconfServiceProvider &*provider*, DataStore *target*, DataStore *source*)
Create or replace an entire configuration datastore with the contents of another complete configuration datastore. If the target datastore exists, it is overwritten. Otherwise, a new one is created, if allowed.

Parameters

- **provider** – An instance of *NetconfServiceProvider*
- **target** – The configuration being used as the destination of type `Datastore`
- **source** – The configuration being used as the source of type `Datastore`

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

bool **copy_config**(NetconfServiceProvider &*provider*, DataStore *target*, Entity &*source*)
Create or replace an entire configuration datastore with the contents of another complete configuration datastore. If the target datastore exists, it is overwritten. Otherwise, a new one is created, if allowed.

Parameters

- **provider** – An instance of *NetconfServiceProvider*
- **target** – The configuration being used as the destination of type `Datastore`
- **source** – The configuration being used as the source of type *Entity*

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

bool **delete_config**(NetconfServiceProvider &*provider*, DataStore *target*, std::string *url* = "")
Delete a configuration datastore. The RUNNING configuration datastore cannot be deleted.

Parameters

- **provider** – An instance of *NetconfServiceProvider*
- **target** – The configuration of type `Datastore` to be deleted
- **url** – Required only when target is set to `url`

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

bool **discard_changes**(NetconfServiceProvider &*provider*)
Used to revert the candidate configuration to the current running configuration

Parameters **provider** – An instance of *NetconfServiceProvider*

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

bool **edit_config**(NetconfServiceProvider &*provider*, DataStore *target*, Entity &*config*, std::string *default_operation* = "", std::string *test_option* = "", std::string *error_option* = "")
Loads all or part of a specified configuration to the specified target configuration datastore. Allows the new configuration to be expressed using a local file, a remote file, or inline. If the target configuration datastore does not exist, it will be created.

Parameters

- **provider** – An instance of *NetconfServiceProvider*
- **target** – The configuration being edited of type `Datastore`
- **config** – An instance of *Entity* that is a hierarchy configuration of data as defined by one of the device's data models
- **default_operation** – Selects the default operation (merge, replace, or none). The default value for this parameter is "merge".
- **test_option** – Optionally set to "test-then-set", "set", or "test-only" if the device advertises the :validate:1.1 capability
- **error_option** – Optionally set to "stop-on-error", "continue-on-error", or "rollback-on-error"

Returns true if the operation succeeds, else false.

Raises YCPPError If an error has occurred

std::shared_ptr<Entity> **get_config**(NetconfServiceProvider &*provider*, DataStore *source*, Entity &*filter*)

Retrieve all or part of a specified configuration datastore

Parameters

- **provider** – An instance of *NetconfServiceProvider*
- **source** – The configuration being queried of type `Datastore`

Returns The requested data as *Entity*

Raises YCPPError If an error has occurred

std::shared_ptr<Entity> **get**(NetconfServiceProvider &*provider*, Entity &*filter*)

Retrieve running configuration and device state information

Parameters

- **provider** – An instance of *NetconfServiceProvider*
- **filter** – An instance of *Entity* that specifies the portion of the system configuration and state data to retrieve

Returns The requested data as *Entity*

Raises YCPPError If an error has occurred

bool **kill_session**(NetconfServiceProvider &*provider*, int *session_id*)

Force the termination of a NETCONF session

Parameters

- **provider** – An instance of *NetconfServiceProvider*
- **session_id** – An instance of int that is the session identifier of the NETCONF session to be terminated

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

bool **lock**(NetconfServiceProvider &*provider*, DataStore *target*)

Allows the client to lock the entire configuration datastore system of a device

Parameters

- **provider** – An instance of *NetconfServiceProvider*

- **target** – The configuration of type `Datastore` to lock

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

bool **unlock**(NetconfServiceProvider &*provider*, DataStore *target*)
Used to release a configuration lock, previously obtained with the LOCK operation

Parameters

- **provider** – An instance of *NetconfServiceProvider*
- **target** – The configuration of type `Datastore` to unlock

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

bool **validate**(NetconfServiceProvider &*provider*, DataStore *source*)
Checks a complete configuration for syntactical and semantic errors before applying the configuration to the device

Parameters

- **provider** – An instance of *NetconfServiceProvider*
- **source** – An instance of `Datastore`

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

bool **validate**(NetconfServiceProvider &*provider*, Entity &*source_config*)
Checks a complete configuration for syntactical and semantic errors before applying the configuration to the device

Parameters

- **provider** – An instance of *NetconfServiceProvider*
- **source** – An instance of *Entity*

Returns true if the operation succeeds, else false

Raises YCPPError If an error has occurred

CodecService

class CodecService : **public** Service

Codec Service class for supporting encoding and decoding C++ model API objects of type `Entity`

CodecService()
Constructs an instance of CodecService

std::string **encode**(CodecServiceProvider &*provider*, Entity &*entity*, bool *pretty* = false)
Perform encoding

Parameters

- **provider** – An instance of *CodecServiceProvider*
- **entity** – An instance of *Entity* class defined under a bundle
- **pretty** – Optionally produce formatted output

Returns Encoded payload in the form of `std::string`

Raises YCPPError If an error has occurred

std::unique_ptr<ydk::*Entity*> **decode**(CodecServiceProvider &*provider*, **const** std::string &*payload*)

Decode the payload to produce an instance of *Entity*

Parameters

- **provider** – An instance of *CodecServiceProvider*
- **payload** – Payload to be decoded

Returns A pointer to an instance of the decoded *Entity*

Raises YCPPError If an error has occurred

ValidationService

class ValidationService

Validation Service class for validating C++ model API objects of type `Entity`

void **validate**(**const** path::ServiceProvider &*provider*, Entity &*entity*, ValidationService::Option *option*)

Validates an entity based on the option.

Parameters

- **provider** – An instance of `ServiceProvider`
- **entity** – An instance of `Entity` class defined under a bundle
- **option** – An instance of type *Option*<*Option*>

Returns An instance of `EntityDiagnostic`

Raises YCPPError If validation errors were detected

enum Option

All validation is performed in the context of some operation. These options capture the context of use.

enumerator DATASTORE

Datastore validation.

enumerator GET_CONFIG

Get config validation. Checks to see if only config nodes are references.

enumerator GET

Get validation.

enumerator EDIT_CONFIG

Edit validation. Checks on the values of leafs etc.

virtual ~ValidationService()

path::CodecService

class `ydk::path::`**CodecService**

Codec Service, part of YDK path API, which deals with generic path-based YANG data nodes

virtual std::string **encode**(**const** std::unique_ptr<DataNode> *datanode*, EncodingFormat *format*, bool *pretty*)

Encode the given DataNode Tree

Parameters

- **datanode** – The `DataNode` to encode
- **format** – format to encode to, either `JSON` or `XML`
- **pretty** – The output is indented for human consumption if pretty is `true`

Returns The encoded string

Raises `YCPPInvalidArgumentError` if the arguments are invalid

virtual std::unique_ptr<DataNode> **decode**(**const** RootSchemaNode &*root_schema*, **const** std::string &*buffer*, Format *format*)

Decode the buffer to return a DataNode

Parameters

- **root_schema** – The root schema to use
- **buffer** – The string representation of the `DataNode`
- **format** – Decode format

Returns The `DataNode` instantiated or `nullptr` in case of error.

Raises `YCPPInvalidArgumentError` if the arguments are invalid.

virtual ~CodecService()

ServiceProvider

NetconfServiceProvider

class NetconfServiceProvider : **public** path::ServiceProvider

Implementation of `ServiceProvider` for the netconf protocol.

NetconfServiceProvider(std::string *address*, std::string *username*, std::string *password*, int *port* = 830)

Constructs an instance of the `NetconfServiceProvider` to connect to a server which **has** to support model download

Parameters

- **address** – IP address of the device supporting a netconf interface
- **username** – Username to log in to the device
- **password** – Password to log in to the device
- **port** – Device port used to access the netconf interface. Default value is 830

NetconfServiceProvider(**const** path::Repository &*repo*, std::string *address*, std::string *username*, std::string *password*, int *port* = 830)

Constructs an instance of the `NetconfServiceProvider` using the provided `repository`

Parameters

- **repository** – Reference to an instance of *path::Repository*
- **address** – IP address of the device supporting a netconf interface

- **username** – Username to log in to the device
- **password** – Password to log in to the device
- **port** – Device port used to access the netconf interface. Default value is 830

path::RootSchemaNode &**get_root_schema**() **const**
Returns the `RootSchemaNode` tree supported by this instance of the `ServiceProvider`.

Returns Pointer to the `RootSchemaNode` or `nullptr` if one could not be created.

std::shared_ptr<path::DataNode> **invoke**(path::Rpc &*rpc*) **const**

Invokes or executes the given rpc and returns a `DataNode` pointer if the Rpc has an output modelled in YANG.

Parameters **rpc** – Reference to the `Rpc` node.

Returns Shared pointer to the `DataNode` representing the output.

EncodingFormat **get_encoding**()
Returns the type of encoding supported by the service provider. In the case of netconf service provider, `EncodingFormat::XML` is returned.

~NetconfServiceProvider()

RestconfServiceProvider

class RestconfServiceProvider : **public** path::ServiceProvider

Implementation of `ServiceProvider` for the restconf protocol.

RestconfServiceProvider(path::Repository &*repo*, **const** std::string &*address*, **const** std::string &*username*, **const** std::string &*password*, int *port* = 80, EncodingFormat *encoding* = EncodingFormat::JSON, **const** std::string &*config_url_root* = "/data", **const** std::string &*state_url_root* = "/data")
Constructs an instance of the `RestconfServiceProvider` to connect to a restconf server

Parameters

- **repository** – Reference to an instance of *path::Repository*
- **address** – IP address of the device supporting a restconf interface
- **username** – Username to log in to the device
- **password** – Password to log in to the device
- **port** – Device port used to access the restconf interface. Default value is 80
- **encoding** – Type of encoding to be used for the payload. Default is `JSON`
- **config_url_root** – To provider backwards compatibility with older drafts of restconf RFC, this can be "/config" or "/data" (which is the default)
- **state_url_root** – To provider backwards compatibility with older drafts of restconf RFC, this can be "/operational" or "/data" (which is the default)

path::RootSchemaNode *__get_root_schema__() **const**
Returns the `RootSchemaNode` tree supported by this instance of the `ServiceProvider`.

Returns Pointer to the `RootSchemaNode` or `nullptr` if one could not be created.

path::DataNode *__invoke__(path::Rpc *_rpc_) **const**

Invokes or executes the given rpc and returns a `DataNode` pointer if the Rpc has an output modelled in YANG.

Parameters **rpc** – Pointer to the `Rpc` node.

Returns Pointer to the `DataNode` representing the output.

EncodingFormat **get_encoding**()
Returns the type of encoding supported by the service provider.

~RestconfServiceProvider()

CodecServiceProvider

class CodecServiceProvider

A provider to be used with :cpp:class‘CodecService<CodecService>‘ for performing encoding and decoding.

CodecServiceProvider(EncodingFormat *encoding*)
Constructs an instance of the `CodecServiceProvider`

Parameters **encoding** – Indicates type of encoding (currently, either `JSON` or `XML`)

CodecServiceProvider(**const** path::Repository &*repo*, EncodingFormat *encoding*)
Constructs an instance of the `CodecServiceProvider`

Parameters

- **repository** – Pointer to an instance of *path::Repository*
- **encoding** – Indicates type of encoding (currently, either `JSON` or `XML`)

~CodecServiceProvider()

OpenDaylightServiceProvider

class OpenDaylightServiceProvider

A service provider to be used to communicate with an OpenDaylight instance.

OpenDaylightServiceProvider(path::Repository &*repo*, **const** std::string &*address*, **const** std::string &*username*, **const** std::string &*password*, int *port* = 8181, EncodingFormat *encoding* = EncodingFormat::JSON, Protocol *protocol* = Protocol::restconf)
Constructs an instance of the `OpenDaylightServiceProvider` to connect to a OpenDaylight instance

Parameters

- **repository** – Reference to an instance of *path::Repository*
- **address** – IP address of the ODL instance
- **username** – Username to log in to the instance
- **password** – Password to log in to the instance
- **port** – Device port used to access the ODL instance. Default value is 8181

- **encoding** – Type of encoding to be used for the payload. Default is `JSON`
- **protocol** – Type of OpenDaylight northbound protocol. Currently, only restconf is supported and is the default value

path::ServiceProvider &**get_node_provider**(**const** std::string &*node_id*)
Returns the `ServiceProvider` instance corresponding to the device being controlled by the OpenDaylight instance, indicated by "node_id".

Parameters **node_id** – The name of the device being controlled by the OpenDaylight instance.

Returns Reference to the `ServiceProvider` or raises `YCPPServiceProviderError` if one could not be found.

const std::vector<std::string> &**get_node_ids**()

Returns a list of node ID's of the devices being controlled by this OpenDaylight instance.

Returns List of node ID's of the devices being controlled by this OpenDaylight instance.

~OpenDaylightServiceProvider()

class ServiceProvider

Interface for all ServiceProvider implementations.

Concrete instances of ServiceProviders are expected to extend this interface.

virtual RootSchemaNode &**get_root_schema**()
Returns the `SchemaNode` tree supported by this instance of the `ServiceProvider`.

Returns Pointer to the `RootSchemaNode` or `nullptr` if one could not be created.

virtual std::shared_ptr<DataNode> **invoke**(Rpc &*rpc*) **const**
Invokes or executes the given rpc and returns a `DataNode` pointer if the Rpc has an output modelled in YANG.

Parameters **rpc** – Reference to the `Rpc` node.

Returns Shared pointer to the `DataNode` representing the output.

EncodingFormat **get_encoding**()
Returns the type of encoding supported by the service provider.

virtual **~ServiceProvider**()

Types

Contents

Edit Operations

enum class `EditOperation`

Operations as defined under netconf edit-config operation attribute in RFC 6241 to be used with various `YDK services` and `entities`.

enumerator `merge`
The configuration data identified by the element containing this attribute is merged with the configuration at the corresponding level in the configuration datastore identified by the <target> parameter. This is the default behavior.

enumerator `create`
The configuration data identified by the element containing this attribute is added to the configuration if and only if the configuration data does not already exist in the configuration datastore. If the configuration data exists, an <rpc-error> element is returned with an <error-tag> value of "data-exists".

enumerator `remove`
The configuration data identified by the element containing this attribute is deleted from the configuration if the configuration data currently exists in the configuration datastore. If the configuration data does not exist, the "remove" operation is silently ignored by the server.

enumerator `delete_`
The configuration data identified by the element containing this attribute is deleted from the configuration if and only if the configuration data currently exists in the configuration datastore. If the configuration data does not exist, an <rpc-error> element is returned with an <error-tag> value of "data-missing".

enumerator `replace`
The configuration data identified by the element containing this attribute replaces any related configuration in the configuration datastore identified by the <target> parameter. If no such configuration data exists in the configuration datastore, it is created. Unlike a <copy-config> operation, which replaces the entire target configuration, only the configuration actually present in the <config> parameter is affected.

enumerator `not_set`
Default value to which all configuration data is initialized to, indicating no operation has been selected. If no operation is selected, `merge` is performed

Example usage An example of setting the operation for an `entity` (address family) under `openconfig BGP` is shown below

```
// Instantiate a bgp smart pointer object representing the bgp container from the openconfig-bgp YANG
auto bgp = std::make_unique<ydk::openconfig_bgp::Bgp>();

// Instantiate an af-safi object representing the af-safi list from the openconfig-bgp YANG model
auto afi_safi = make_unique<ydk::openconfig_bgp::Bgp::Global::AfiSafis::AfiSafi>();

// Set the operation to delete, which will delete this instance of the address family
afi_safi->operation = EditOperation::delete_;

// Set the key
afi_safi->afi_safi_name = L3VpnIpv4UnicastIdentity();
// Set afi-safis as the parent of the list instance
afi_safi->parent = bgp->global->afi_safis.get();
//Append the list instance to afi-safis's afi-safi field
bgp->global->afi_safis->afi_safi.push_back(std::move(afi_safi));
```

```

// Instantiate the CRUD service and Netconf provider to connect to a device with address 10.0.0.1
CrudService crud_service{};
NetconfServiceProvider provider{"10.0.0.1", "test", "test", 830};

// Invoke the CRUD Update method
crud_service.update(provider, *bgp);
```

The C++ types present in ydk namespace corresponding to YANG types. See below for example usage.

enum class `EncodingFormat`
Format of encoding to be used when creating the payload.

enumerator `XML`
XML

enumerator `JSON`
JSON

enum class `Protocol`
Type of protocol.

enumerator `netconf`
Netconf protocol

enumerator `restconf`
Restconf protocol

class `Entity`

Super class of all classes that represents containers in YANG. YANG lists are represented as `std::vector` of Entity objects, with support for hanging a parent

EditOperation **`operation`**
Optional attribute of the Entity class which can be set to perform various `operations`

YANG container and list

class `Entity`

Super class of all classes that represents containers in YANG. YANG lists are represented as `std::vector` of Entity objects, with support for hanging a parent

EditOperation **`operation`**
Optional attribute of the Entity class which can be set to perform various `operations`

YANG leaf and leaf-list

class `YLeaf`

Concrete class that represents a YANG leaf to which data can be assigned.

EditOperation **`operation`**
Optional attribute of the YLeaf class which can be set to perform various `operations`

class `YLeafList`

Concrete class that represents a YANG leaf-list to which multiple instances of data can be appended to.

EditOperation **`operation`**
Optional attribute of the YLeafList class which can be set to perform various `operations`

YANG type

class `Bits`

Concrete class representing a bits data type.

class `Decimal64`

Concrete class representing a decimal64 data type.

class `Empty`

Represents the empty type in YANG. The empty built-in type represents a leaf that does not have any value, it conveys information by its presence or absence.

class `Enum`

Super class of all classes that represents the enumeration type in YANG.

class `Identity`

Super class of all classes that represents the identity type in YANG. Instances of this type of classes are assigned as data to leafs of `identityref` type.

Example usage

Examples of instantiating and using objects of `Entity` type are shown below

```
// Instantiate a bgp smart pointer object representing the bgp container from the openconfig-bgp YANG
auto bgp = std::make_unique<ydk::openconfig_bgp::Bgp>();

// Instantiate an af-safi object representing the af-safi list from the openconfig-bgp YANG model
auto afi_safi = make_unique<ydk::openconfig_bgp::Bgp::Global::AfiSafis::AfiSafi>();
// Set afi-safis as the parent of the list instance
afi_safi->parent = bgp->global->afi_safis.get();
//Append the list instance to afi-safis's afi-safi field
bgp->global->afi_safis->afi_safi.push_back(std::move(afi_safi));
```

Examples of assigning values to leafs are shown below

```
// Assign values to leafs of various types

bgp->global->config->as = 65172; // uint32
bgp->global->config->router_id = "1.2.3.4"; //ip-address
afi_safi->afi_safi_name = L3VpnIpv4UnicastIdentity(); //identityref
afi_safi->config->enabled = false; //boolean
neighbor->config->peer_type = PeerTypeEnum::INTERNAL // enum
Decimal64 deci{"1.2"};
node->decimal_value = deci; //decimal64
node->empty_value = Empty(); // empty
node->bits_value["first-position"] = true // bits
node->bits_value["second-position"] = false // bits
```

Examples of appending values to leaf-lists are shown below

```
// Append values to leaf-lists of various types

config->as_list.append(65172); // uint32
config->router_id.append("1.2.3.4"); //ip-address
L3VpnIpv4UnicastIdentity id{}; //identityref
config->types_list.append(id); //identityref
```

```
config->enabled_list.append(false); //boolean
config->peer_types.append(PeerTypeEnum::INTERNAL) // enum
Decimal64 deci{"1.2"};
node->decimal_values.append(deci); //decimal64

Bits bits_value; // bits
bits_value["first-position"] = true; // bits
bits_value["first-position"] = false; // bits
node->bits_values.append(bits_value); // bits
```

Errors

class `YCPPError`

Base class for YDK Exceptions.

std::string **`err_msg`**

`YCPPError`(**const** std::string &*msg*)

YCPPCodecError

class `YCPPCodecError` : **public** YCPPCoreError

Exception that encapsualtes the validation errors for YDK `CodecService`.

`YCPPCodecError`(YCPPCodecError::Error *merror*)

enum `Error`

enumerator `SUCCESS`
No error.

enumerator `XML_MISS`
Missing XML object.

enumerator `XML_INVAL`
Invalid XML object.

enumerator `XML_INCHAR`
Invalid XML character.

enumerator `EOF_ERR`
Unexpected end of input data.

YCPPDataValidationError

class `YCPPDataValidationError` : **public** YCPPCoreError

Exception that encapsualtes the validation errors on a data tree.

std::vector<std::pair<DataNode *, Error>> **`errors`**
List of validation errors specific to this node.

`YCPPDataValidationError`()

enum `Error`
Data Validation Error Enum.

enumerator SUCCESS
No error.

enumerator TOOMANY
Too many instances of some object.

enumerator DUPLEAFLIST
Multiple instances of leaf-list.

enumerator DUPLIST
Multiple instances of list.

enumerator NOUNIQ
Unique leaves match on 2 list instances (data).

enumerator OBSDATA
Obsolete data instantiation (data).

enumerator NORESOLV
No resolvents found for an expression (data).

enumerator INELEM
Nvalid element (data).

enumerator MISSELEM
Missing required element (data).

enumerator INVAL
Invalid value of an element (data).

enumerator INVALATTR
Invalid attribute value (data).

enumerator INATTR
Invalid attribute in an element (data).

enumerator MISSATTR
Missing attribute in an element (data).

enumerator NOCONSTR
Value out of range/length/pattern (data).

enumerator INCHAR
Unexpected characters (data).

enumerator INPRED
Predicate resolution fail (data).

enumerator MCASEDATA
Data for more cases of a choice (data).

enumerator NOMUST
Unsatisfied must condition (data).

enumerator NOWHEN
Unsatisfied when condition (data).

enumerator INORDER
Invalid order of elements (data).

enumerator INWHEN
Irresolvable when condition (data).

enumerator NOMIN
Min-elements constraint not honored (data).

enumerator NOMAX
Max-elements constraint not honored (data).

enumerator NOREQINS
Required instance does not exits (data).

enumerator NOLEAFREF
Leaf pointed to by leafref does not exist (data).

enumerator NOMANDCHOICE
No mandatory choice case branch exists (data).

YCPPPathError

class YCPPPathError : **public** YCPPCoreError

Exception that encapsualtes the validation errors for YDK Path.

Error **err**

YCPPPathError(YCPPPathError::Error *error_code*)

YCPPSchemaValidationError

class YCPPSchemaValidationError : **public** YCPPCoreError

Exception that encapsualtes the validation errors for schema validation.

enum Error

enumerator SUCCESS
No error.

enumerator INSTMT
Invalid statement (schema).

enumerator INID
Nvalid identifier (schema).

enumerator INDATE
Invalid date format.

enumerator INARG
Invalid value of a statement argument (schema).

enumerator MISSSTMT
Missing required statement (schema).

enumerator MISSARG
Missing required statement argument (schema).

enumerator TOOMANY
Too many instances of some object.

enumerator DUPID
Duplicated identifier (schema).

enumerator DUPLEAFLIST
Multiple instances of leaf-list.

enumerator DUPLIST
Multiple instances of list.

enumerator NOUNIQ
Unique leaves match on 2 list instances (data).

enumerator ENUM_DUPVAL
Duplicated enum value (schema).

enumerator ENUM_DUPNAME
Duplicated enum name (schema).

enumerator ENUM_WS
Enum name with leading/trailing whitespaces (schema).

enumerator BITS_DUPVAL
Duplicated bits value (schema).

enumerator BITS_DUPNAME
Duplicated bits name (schema).

enumerator INMOD
Nvalid module name.

enumerator KEY_NLEAF
List key is not a leaf (schema).

enumerator KEY_TYPE
Invalid list key type (schema).

enumerator KEY_CONFIG
Key config value differs from the list config value.

enumerator KEY_MISS
List key not found (schema).

enumerator KEY_DUP
Duplicated key identifier (schema).

enumerator INREGEX
Nvalid regular expression (schema).

enumerator INRESOLV
No resolvents found (schema).

enumerator INSTATUS
Nvalid derivation because of status (schema).

enumerator CIRC_LEAFREFS
Circular chain of leafrefs detected (schema).

enumerator CIRC_IMPORTS
Circular chain of imports detected (schema).

enumerator CIRC_INCLUDES
Circular chain of includes detected (schema).

YCPPCoreError

class YCPPCoreError : **public** YCPPError

The subclasses give a specialized view of the error that has occurred.

YCPPCoreError()

YCPPCoreError(**const** std::string &*msg*)

YCPPIllegalStateError

class YCPPIllegalStateError : **public** YCPPError

Thrown when an operation/service is invoked on an object that is not in the right state. Use the `msg` for the error.

YCPPIllegalStateError(**const** std::string &*msg*)

YCPPInvalidArgumentError

class YCPPInvalidArgumentError : **public** YCPPError

Use the msg for the error.

YCPPInvalidArgumentError(**const** std::string &*msg*)

YCPPOperationNotSupportedError

Operation Not Supported Exception.

class YCPPOperationNotSupportedError : **public** YCPPError

Thrown when an operation is not supported.

YCPPOperationNotSupportedError(**const** std::string &*msg*)

YCPPServiceProviderError

class YCPPServiceProviderError : **public** YCPPError

Exception for service provider.

YCPPServiceProviderError(**const** std::string &*msg*)

YCPPModelError

class YCPPModelError : **public** YCPPError

This error is raised in case a model constraint is violated.

YCPPModelError(**const** std::string &*msg*)

Path API

YDK C++ provides a new interface in the form of Path API, which can be used to write apps using a generic API, using xpath-like path expression to create and access YANG data nodes. Internally, the nodes created using the YDK model API are converted to Path API data nodes for validation and encoding to respective protocol payloads.

Annotation

class `Annotation`

Class represents an annotation.

An annotation has a namespace and a name and an associated value. Annotations are not defined in the YANG model and hence just provide a means of hanging some useful data to `DataNode`. For example netconf edit-config rpc operation uses the annotation `nc:operation` (`nc` refers to the netconf namespace) on the data nodes to describe the kind of operation one needs to perform on the given `DataNode`.

std::string **`m_ns`**
: Annotation namespace.

std::string **`m_name`**
: Annotation name.

std::string **`m_val`**
: Annotation value.

void **`Annotation`**(**const** std::string &*ns*, **const** std::string &*name*, **const** std::string &*val*)

void **`Annotation`**(**const** Annotation &*an*)

void **`Annotation`**(Annotation &&*an*)

Annotation &**`operator=`**(**const** Annotation &*an*)

Annotation &**`operator=`**(Annotation &&*an*)

bool **`operator==`**(**const** Annotation &*an*) **const**

Capability

class `Capability`

Class represents the Capability. An instance of Capability is defined by the module name and revision along with the set of enabled features defined in this modules as well as the list of deviations which target nodes defined by this module.

std::string **`module`**
: The module.

std::string **`revision`**
: The revision.

std::vector<std::string> **`features`**
: List of features defined in this module that are enabled.

std::vector<std::string> **`deviations`**
: List of deviations that target nodes defined by this module.

`Capability`(**const** std::string &*mod*, **const** std::string &*rev*)

`Capability`(**const** std::string &*mod*, **const** std::string &*rev*, **const** std::vector<std::string> &*f*, **const** std::vector<std::string> &*d*)

`Capability`(**const** Capability &*cap*)

`Capability`(Capability &&*cap*)

Capability &**`operator=`**(**const** Capability &*cap*)

Capability &**`operator=`**(Capability &&*cap*)

bool **operator==**(**const** Capability &*cap*)

Repository

class Repository

Represents the Repository of YANG models.

A instance of the `Repository` will be used to create a `RootSchemaNode` given a set of `Capabilities`.

Behind the scenes the repository is responsible for loading and parsing the YANG modules and creating the `SchemaNode` tree. The `ServiceProvider` is expected to use the method `create_root_schema` to generate the `RootSchemaNode`.

Repository()
Constructs an instance of the `Repository` class. If the server supports model download, the repo will attempt to download all models from the server using the `ModelProvider` provided using the `add_model_provider` method.

Raises `YCPPInvalidArgumentError` if the search_dir is not a valid directory in the filesystem.

Repository(**const** std::string &*search_dir*)
Constructs an instance of the `Repository` class.

Parameters **search_dir** – The path in the filesystem where yang files can be found.

Raises `YCPPInvalidArgumentError` if the search_dir is not a valid directory in the filesystem.

std::unique_ptr<RootSchemaNode> **create_root_schema**(**const** std::vector<Capability> *capabilities*) **const**
Creates the root schema based on the `std::vector` of capabilities passed in.

This method verifies the said capabilities and can throw exceptions if a module is not found in the search directory or cannot be loaded.

Parameters **capabilities** – `std::vector` of `Capability`.

Returns Pointer to the `RootSchemaNode` or `nullptr` if one could not be created.

void **add_model_provider**(ModelProvider **model_provider*)
Adds a model provider to this Repository. If the repository does not find a model while trying to create a SchemaTree it calls on the model_provider to see if the said model can be downloaded by one of them. If that fails it tries the next.

Parameters **model_provider** – The model provider to add

void **remove_model_provider**(ModelProvider **model_provider*)
Removes the given model provider from this repository

Parameters **model_provider** – The model provider to remove

std::vector<ModelProvider *> **get_model_providers**() **const**
Gets all model providers registered with this repository.

Returns `std::vector` of model providers associated with this repository

std::string **path**
Location where YANG models are present and/or downloaded to

ModelProvider

NetconfModelProvider

class `NetconfModelProvider`

Implementation of `ModelProvider` for netconf

`NetconfModelProvider`(NetconfClient &*client*)
Constructs an instance of the `NetconfModelProvider`

Parameters **`client`** – Instance of an existing netconf session

std::string **`get_model`**(**const** std::string &*name*, **const** std::string &*version*, Format *format*)
Returns the model identified by the name and version

Parameters

- **`name`** – name of the model
- **`version`** – version of the model
- **`format`** – `format` of the model to download

Returns `std::string` containing the model downloaded. If empty then the model probably cannot be provided

std::string **`get_hostname_port`**()

class `ModelProvider`

Interface for a YANG model provider which can download YANG models from a server

enum `Format`
Format of model to be downloaded

enumerator `YANG`

enumerator `YIN`

`ModelProvider`()
Constructs an instance of the `ModelProvider`

virtual std::string **`get_model`**(**const** std::string &*name*, **const** std::string &*version*, Format *format*) = 0
Returns the model identified by the name and version

Parameters

- **`name`** – name of the model
- **`version`** – version of the model
- **`format`** – `format` of the model to download

Returns `std::string` containing the model downloaded. If empty then the model probably cannot be provided

virtual std::string **`get_hostname_port`**() = 0
Return the hostname and port of the connected server

Rpc

class Rpc

An instance of the YANG schmea rpc node.

Instances of this class represent a YANG rpc and are modelled as Callables. The input data node tree is used to populate the input parameters to the rpc if any. The Callable takes as a parameter the `ServiceProvider` that can execute this rpc as its parameter returning a pointer to a `DataNode` tree if output is available.

virtual ~Rpc()

virtual std::shared_ptr<DataNode> **operator()**(**const** ServiceProvider &*provider*)
Execute/Invoke the rpc through the given service provider.

Parameters **sp** – The Service provider.

Returns Pointer to the `DataNode` or `nullptr` if none exists.

virtual DataNode &**input**() **const**
Get the input data tree.

Returns Pointer to the input `DataNode` or `nullptr` if the rpc does not have an input element in the schema.

virtual SchemaNode &**schema**() **const**

Returns Pointer to the `SchemaNode` associated with this rpc.

Statement

class Statement

Represents the YANG `Statement`.

std::string **keyword**
YANG keyword corresponding to the `Statement`.

std::string **arg**
The arg if any.

Statement(): keyword{}, arg{}

Statement(**const** std::string &*mkeyword*, **const** std::string &*marg*)

Statement(**const** Statement &*stmt*)

Statement(Statement &&*stmt*)

~Statement()

Statement &**operator=**(**const** Statement &*stmt*)

Statement &**operator=**(Statement &&*stmt*)

DataNode

class DataNode

Class represents the `DataNode`.

virtual ~DataNode()
The destructor.

Note: A DataNode represents a containment hierarchy. So invocation of the destructor will lead to the children of this node being destroyed.

---

virtual const SchemaNode &**schema**() **const**
Returns the SchemaNode associated with this DataNode.

Returns SchemaNode associated with this DataNode.

virtual std::string **path**() **const**
Returns the path expression representing this Node in in the NodeTree.

Returns std::string representing the path to this Node.

virtual DataNode &**create**(**const** std::string &*path*)
Creates a DataNode corresponding to the path and set its value.

This methods creates a DataNode tree based on the path passed in. The path expression must identify a single node. If the last node created is of schema type list, leaf-list or anyxml that value is also set in the node.

The returned DataNode is the last node created (the terminal part of the path).

The user is responsible for managing the memory of this returned tree. Use root() to get the root element of the this tree and use that pointer to dispose of the entire tree.

Note: In the case of list nodes the keys must be present in the path expression in the form of predicates.

Parameters

- **path** – The XPath expression identifying the node.
- **value** – The string representation of the value to set.

Returns Pointer to DataNode created.

Raises YCPPInvalidArgumentError In case the argument is invalid.

Raises YCPPPathError In case the path is invalid.

virtual DataNode &**create**(**const** std::string &*path*, **const** std::string &*value*)
Create a DataNode corresponding to the path and set its value.

This methods creates a DataNode tree based on the path passed in. The path expression must identify a single node. If the last node created is of schema type list, leaf-list or anyxml that value is also set in the node.

The returned DataNode is the last node created (the terminal part of the path).

The user is responsible for managing the memory of this returned tree. Use root() to get the root element of the this tree and use that pointer to dispose of the entire tree.

Note: In the case of list nodes the keys must be present in the path expression in the form of predicates.

Parameters **path** – The XPath expression identifying the node.

Returns Pointer to `DataNode` created.

Raises `YCPPInvalidArgumentError` In case the argument is invalid.

Raises `YCPPPathError` In case the path is invalid.

virtual void **set**(**const** std::string &*value*)
Set the value of this `DataNode`.

Note:

- The `DataNode` should represent a `leaf` , `leaf-list` or `anyxml` element for this to work. The value should be the string representation of the type of according to the schema.
- This method does not validate the value being set. To validate please see the `ValidationService`.

Parameters **value** – The value to set. This should be the string representation of the YANG type.

Raises `YCPPInvalidArgumentError` if the `DataNode`'s value cannot be set (for example it represents a container)

virtual std::string **get**() **const**
Returns a copy of the value of this `DataNode`.

Returns A `std::string` representation of the value.

virtual std::vector<std::shared_ptr<DataNode>> **find**(**const** std::string &*path*) **const**
Finds nodes that satisfy the given path expression. For details about the path expression see *how to path*.

Parameters **path** – The path expression.

Returns Vector of `DataNode` that satisfy the path expression supplied.

virtual DataNode ***parent**() **const**

Returns Pointer to the parent of this `DataNode` or `nullptr` if None exist.

virtual std::vector<std::shared_ptr<DataNode>> **children**() **const**

Returns Pointer to the children of this `DataNode`.

virtual const DataNode &**root**() **const**

Returns Pointer to the root `DataNode` of this tree.

virtual void **add_annotation**(**const** Annotation &*an*)
This method adds the annotation to this `Datanode`.

Parameters **an** – The annotation to add to this `DataNode`.

Raises `YCPPInvalidArgumentError` in case the argument is invalid.

virtual bool **remove_annotation**(**const** Annotation &*an*)
This method will remove the annotation from the given node.

Note: The `m_val` for annotation is ignored.

Parameters **an** – The reference to the annotation.

Returns `bool` If true the annotation was found and removed, false otherwise.

virtual std::vector<Annotation> **annotations**()
Get the annotations associated with this data node.

Returns Vector of `Annotation` for this node.

SchemaNode

RootSchemaNode

Instances of this class represent the Root of the SchemaTree. A `RootSchemaNode` can be used to instantiate a `DataNode` tree or an `Rpc` object. The children of the `RootSchemaNode` represent the top level `SchemaNode` in the YANG module submodules.

class **RootSchemaNode** : **public** SchemaNode

Instances of this class represent the Root of the `SchemaNode` Tree.

virtual **~RootSchemaNode**()
Destructor for the `RootSchemaNode`

std::string **path**() **const**
Get the path expression representing this Node in in the NodeTree.

Returns `std::string` representing the path to this Node.

virtual std::vector<SchemaNode *> **find**(**const** std::string &*path*) **const**
Finds descendant nodes that match the given xpath expression.

This API finds descendant nodes in the `SchemaNode` tree that satisfy the given path expression. See *how to path*.

Parameters **path** – The path expression.

Returns Vector of `SchemaNode` that satisfies the criterion.

Raises `YCPPPathError` if the path expression in invalid, See error code for details.

Raises `YCPPInvalidArgumentError` if the argument is invalid.

virtual SchemaNode ***parent**() **const noexcept**
Get the parent node of this `SchemaNode` in the tree.

Returns Pointer to the parent node or `nullptr` in case this is the root.

virtual std::vector<SchemaNode *> **children**() **const**
Get the children of this `SchemaNode` in the NodeTree.

Returns The children of this node.

virtual const SchemaNode ***root**() **const noexcept**
Get the root of NodeTree this node is part of.

Returns The pointer to the root.

virtual DataNode ***create**(**const** std::string &*path*, **const** std::string &*value*) **const**
Create a `DataNode` corresponding to the path and set its value.

This methods creates a `DataNode` tree based on the path passed in. The path expression must identify a single node. If the last node created is of schema type `list`, `leaf-list` or `anyxml` that value is also set in the node.

The returned `DataNode` is the last node created (the terminal part of the path).

The user is responsible for managing the memory of this returned tree. Use `root()` to get the root element of the this tree and use that pointer to dispose of the entire tree.

Note in the case of List nodes the keys must be present in the path expression in the form of predicates.

Parameters

- **`path`** – The XPath expression identifying the node relative to the root of the schema tree.
- **`value`** – The string representation of the value to set.

Returns Pointer to `DataNode` created.

Raises `YCPPInvalidArgumentError` In case the argument is invalid.

Raises `YCPPPathError` In case the path is invalid.

virtual DataNode ***`create`**(**const** std::string &*path*, **const** std::string &*value*) **const**
Create a `DataNode` corresponding to the path and set its value.

This methods creates a DataNode tree based on the path passed in. The path expression must identify a single node. If the last node created is of schema type `list`, `leaf-list` or `anyxml` that value is also set in the node.

The returned `DataNode` is the last node created (the terminal part of the path).

The user is responsible for managing the memory of this returned tree. Use `root()` to get the root element of the this tree and use that pointer to dispose of the entire tree.

Note in the case of List nodes the keys must be present in the path expression in the form of predicates.

Parameters **`path`** – The XPath expression identifying the node.

Returns `DataNode` created or `nullptr`.

Raises `YCPPInvalidArgumentError` In case the argument is invalid.

Raises `YCPPPathError` In case the path is invalid.

virtual Statement **`statement`**() **const**
Return the `Statement` representing this `SchemaNode`.

Note the `RootSchemaNode` has no YANG statement representing it.

So this method returns an empty statement.

Returns An empty statement.

virtual Rpc ***`rpc`**(**const** std::string &*path*) **const**
Create an `Rpc` instance.

The path expression should point to a `SchemaNode` that represents the `Rpc`.

Parameters **`path`** – The path to the rpc schema node

Returns Pointer to `Rpc` or `nullptr`.

Raises `YCPPInvalidArgumentError` if the argument is invalid.

Raises `YCPPPathError` if the path is invalid.

class `SchemaNode`

Represents a Node in the SchemaTree.

`~SchemaNode``()`
The destructor.

Note: A `SchemaNode` represents a containment hierarchy. So invocation of the destructor will lead to the children of this node being destroyed.

---

virtual std::string **`path`**`()` **const**
Get the path expression representing this Node in in the NodeTree.

Returns `std::string` representing the path to this Node.

virtual std::vector<SchemaNode *> **`find`**`(`**const** std::string &*path*`)` **const**
Finds descendant nodes that match the given xpath expression.

This API finds descendant nodes in the `SchemaNode` tree that satisfy the given path expression. See *how to path*.

Parameters **`path`** – The path expression.

Returns Vector of `SchemaNode` that satisfies the criterion.

Raises `YCPPPathError` if the path expression in invalid, see error code for details.

Raises `YCPPInvalidArgumentError` if the argument is invalid.

virtual const SchemaNode ***`parent`**`()` **const noexcept**
Get the Parent Node of this SchemaNode in the tree.

Returns `SchemaNode*` the children of this node.

virtual std::vector<std::unique_ptr<SchemaNode>> &**`children`**`()` **const**
Returns the children of this `SchemaNode` in the NodeTree.

Returns Child schema nodes.

virtual const SchemaNode &**`root`**`()` **const noexcept**
Get the root of NodeTree this node is part of

Returns The pointer to the root.

virtual Statement **`statement`**`()` **const**
Returns the YANG statement associated with this `SchemaNode`

Returns The pointer to the yang statement for this `SchemaNode`

virtual std::vector<Statement> **`keys`**`()` **const**
Returns vector of YANG statement corresponding the the keys.

Returns Vector of `Statement` that represent keys.

Namespaces

List of all namespaces with brief descriptions

ydk

List of all nested namespaces with brief descriptions

ydk::path

Data Structures

Enumerations

ValidationError ValidationError

enumerator	documentation
SUCCESS	No error
SCHEMA_NOT_FOUND	Entity's schema node is not found
INVALID_USE_OF_SCHEMA	If element cannot have children as per schema (leaf, leaf-list, anyxml)
TOOMANY	Too many instances of some object
DUPLEAFLIST	Multiple instances of leaf-list
DUPLIST	Multiple instances of list
NOUNIQ	Unique leaves match on 2 list instances (data)
OBSDATA	Obsolete data instantiation (data)
NORESOLV	No resolvents found for an expression (data)
INELEM	Nvalid element (data)
MISSELEM	Missing required element (data)

Continued on next page

Table 1.1 – continued from previous page

INVAL	Invalid value of an element (data)
INVALATTR	Invalid attribute value (data)
INATTR	Invalid attribute in an element (data)
MISSATTR	Missing attribute in an element (data)
NOCONSTR	Value out of range/length/pattern (data)
INCHAR	Unexpected characters (data)
INPRED	Predicate resolution fail (data)
MCASEDATA	Data for more cases of a choice (data)
NOMUST	Unsatisfied must condition (data)
NOWHEN	Unsatisfied when condition (data)
INORDER	Invalid order of elements (data)
INWHEN	Irresolvable when condition (data)
NOMIN	Min-elements constraint not honored (data)
NOMAX	Max-elements constraint not honored (data)
NOREQINS	Required instance does not exits (data)
NOLEAFREF	Leaf pointed to by leafref does not exist (data)

Continued on next page

Table 1.1 – continued from previous page

NOMANDCHOICE	No mandatory choice case branch exists (data)
INVALID_BOOL_VAL	Invalid boolean value
INVALID_EMPTY_VAL	Invalid empty value
INVALID_PATTERN	Pattern did not match
INVALID_LENGTH	Length is invalid
INVALID_IDENTITY	Invalid identity
INVALID_ENUM	Invalid enumeration

class	`Service`
class	`CodecService`
class	`CrudService`
class	`CodecServiceProvider`
class	`NetconfServiceProvider`
class	`NetconfModelProvider`
class	`Entity`
class	`YLeaf`
class	`YLeafList`
class	`Bits`
class	`Empty`
class	`Enum`
class	`Identity`
class	`Decimal64`

Y