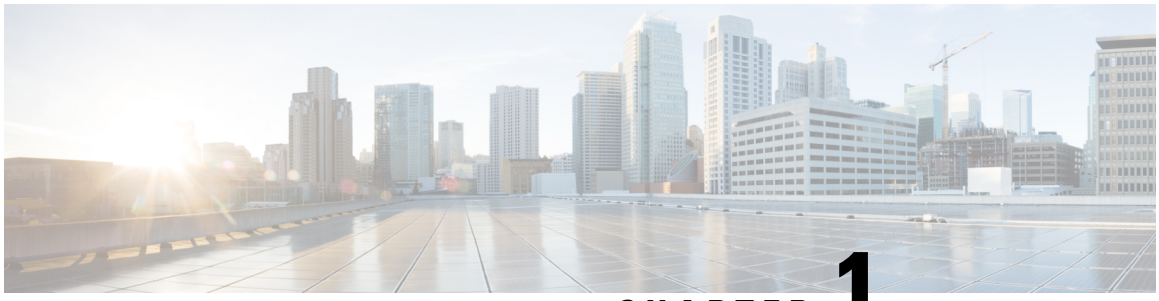
Cisco Crosswork Workflow Manager 1.2 Adapter Developer guide

First Published: 2024-09-27

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883



CHAPTER 1

Overview

This section contains the following topics:

Overview

Workflow Adapters are tools that allow a workflow to interact with systems outside the CWM. You can see them as agents and intermediaries between the CWM platform and any external services. Their role is to cause an action in an outside system that's part of a workflow stream, or to retrieve data required by a workflow to progress.

Every adapter is developed for communicating with an intended target service. Target services can be generic, such as REST APIs over HTTP, or specific, such as vendor products (Cisco's Network Services Orchestrator, for example).

If a workflow needs to access one or more external services, you can develop custom adapters for each of them using the Adapter SDK. You may also want to use four pre-built adapters which are available as part of the CWM offering. These ready-made solutions include: the Network Services Orchestrator adapter, a generic REST API adapter, an Email adapter, and a generic CLI adapter.

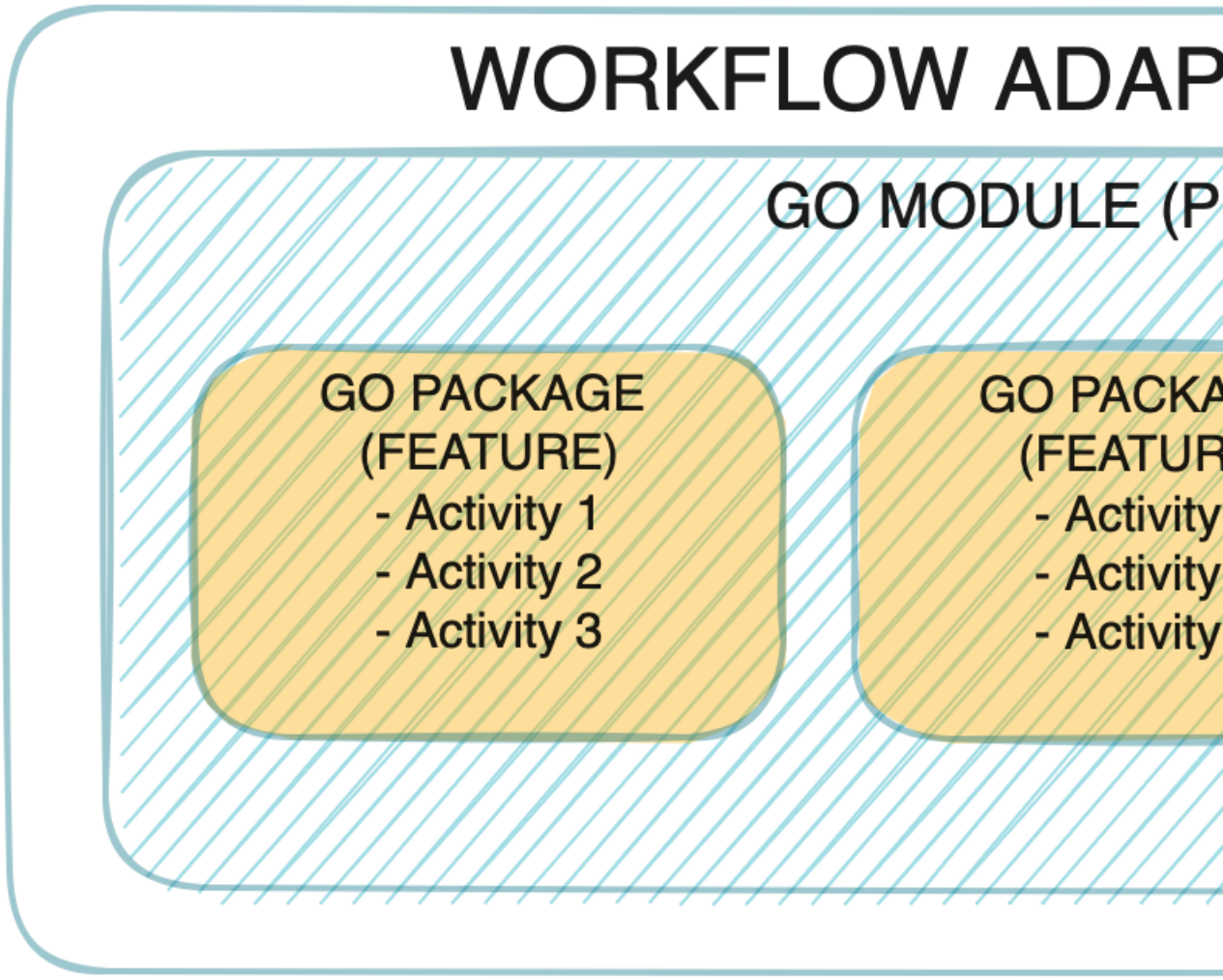
What's in an adapter

An adapter is developed using the Workflow Adapter SDK which uses Golang for defining adapter logic and leverages Protocol Buffers for representing adapter interfaces.

Modules, packages, activities

Every adapter is a **go module** that represents a product by a vendor. The **go module** in turn is a collection of product features organized into **go packages**. Inside the packages you define adapter activities, which are particular actions that the adapter can trigger within a given external system. You can have multiple features inside one adapter by bundling related activities into separate packages.

Figure 1: Adapter schema



As shown in the picture, every adapter follows the vendor, product and feature naming convention which corresponds to a standard **go** project layout with modules and packages as described above.

Interfaces

Each product feature is represented by a protobuf file located in the `proto` folder. The Adapter SDK provides command arguments to create the adapter structure and files.

As mentioned before, the naming convention for the adapter features is `<vendor>.<product>.<feature>`, for example, `cisco.nso.restconf`.

When you create an adapter, the Adapter SDK generates a `.proto` file for each interface (feature) you specified:

```
syntax = "proto3";

package <vendor>.<product>.<feature>;

option go_package = "<module>/<feature>";
```

The interface is defined as a list of RPCs in the service named 'Activities':

```
service Activities {
    rpc <ActOne> (<ActOne>Request) returns (<ActOne>Response);
    rpc <ActTwo> (<ActTwo>Request) returns (<ActTwo>Response);
}
```

Lastly, the input and output of each activity are defined as protobuf messages:

```
message <ActOne>Request {
    ...
}
message <ActOne>Response {
    ...
}
...
```

common.adapter.proto

Besides the `.proto` files representing the adapter interface, there is one additional file: `<vendor>.<product>.common.adapter.proto`.

The *common* `.proto` file is used to define additional configuration required by the adapter as well as information allowing the adapter to connect to a target system. The file is generated automatically by the Adapter SDK, but the developer can do any manual updates required.

Note The *common* `.proto` file must define certain messages to enable the CWM Resource Manager to handle this data correctly. This can be done directly inside the file (default) or by importing another `.proto`.

```
// Can be defined anywhere and imported to common .proto file.
message Resource {
    ...
}
message Secret {
    ...
}

// Must be defined in common .proto file.
message Config {
    Resource resource = 1;
    Secret   secret   = 2;
}
```

Activities

The Adapter SDK generates activities to be implemented in Golang. Each activity is represented as a method with the receiver being a pointer to an adapter struct. Each method definition is based on the activity RPC defined in proto.

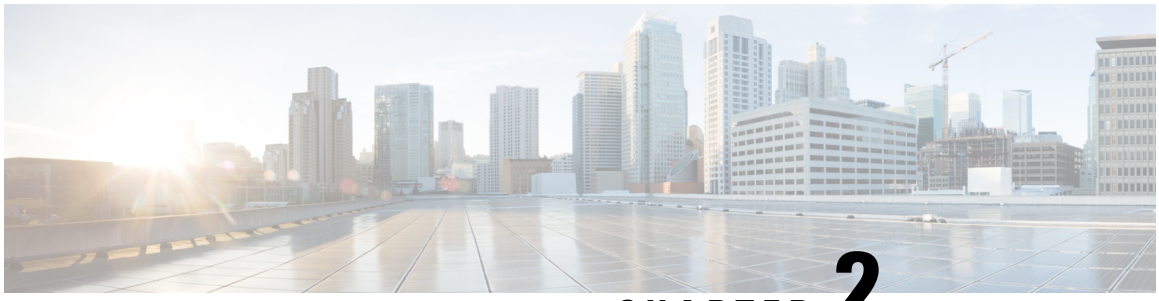
```
func (adp *Adapter) <ActivityName>(
        ctx context.Context,
        req *<ActivityName>Request,
        cfg *common.Config) (*<ActivityName>Response, error) {
    /* Activity implementation */
}
```

Note There are no restrictions on how to implement an activity. The developer is free to import any available go packages. One suggestion is to avoid panics by having robust error handling with the activity returning a meaningful error code.

Properties

Each adapter has a `.properties` file which serves the CWM as the source of basic data about the adapter. Mandatory properties are described below with examples:

Property	Description
author=cisco	Name of adapter developer
vendor=cisco	Name of target system vendor
product=nso	Name of target system
version=1.0.0	Adapter version
cwmsdk=1.0.0	Version of SDK used for developing the adapter
cwmversion=1.0	Compatible CWM version
resourcetype=cisco.nso.resource.v1.0.0	Compatible resource type stored by CWM Resource Manager



CHAPTER 2

Use Adapter SDK

This section contains the following topics:

Prerequisites

To start using the CWM Adapter SDK, you need to install a **Golang** environment, Protocol buffers, dedicated **go** plugins and download the Adapter SDK contained in the CWM software package.

Install Protocol buffers

To define an adapter interface and generate the input and output parameters, you need the Protobufs compiler. Follow the installation instructions dedicated for your OS: https://grpc.io/docs/protoc-installation/. Note that you need at least version **3.15** (proto3).

Install Go and plugins

To develop and test an adapter, you need to install the **Golang** environment. Follow the installation instructions dedicated for your OS: https://grpc.io/docs/protoc-installation/.

Step 1 Install additional protocol compiler plugins for **go**:

```
go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.28
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.2
go install github.com/pseudomuto/protoc-gen-doc/cmd/protoc-gen-doc@latest
```

Step 2 Install protocol compiler plugin for **JSON schema**:

```
go install github.com/pubg/protoc-gen-jsonschema@latest
```

Step 3 Update your system PATH so that the `protoc` compiler can find the plugins:

```
export PATH="$PATH:$(go env GOPATH)/bin"
```

Get CWM Adapter SDK

Go to Cisco Software Download page to download the CWM Software Package, where the Adapter SDK binary resides.

Include the location of adapter developer binaries by setting the environment variable path:

```
export PATH=/path/to/adapter-dev-binaries:$PATH
```

Note Remember to replace the `/path/to/adapter-dev-binaries` with your actual path.

Overview of commands

The Adapter SDK application offers the following set of commands for managing an adapter:

- `cwm-sdk version` - display the version of cwm-sdk application.
- `cwm-sdk create-adapter` - create a go module with a package and the corresponding .proto files.
- `cwm-sdk extend-adapter` - add a new feature to an existing adapter (go package and .proto files).
- `cwm-sdk update-adapter` - update activities, input and output (go code).
- `cwm-sdk upgrade-adapter` - upgrade the adapter to match CWM.
- `cwm-sdk create-installable` - create an archive installable by CWM.

Create new adapter

To create an adapter, open a command-line terminal and run:

```
cwm-sdk create-adapter [options] -product <product-name>
```

Note While the `-product` parameter is required for adapter creation, other options can be skipped.

Options

These are the options you can add to the `create-adapter` command:

Option	Data type	Requirement	Description
`-exclude-resource`	string	optional	skip creation of the `.resource.proto` file from template.
`-feature`	string	optional	provide name for the go package assigned to activities (default: "<adapter-name>").
`-go-module`	string	optional	provide name for the module assigned to the go.mod file (default: "www.cisco.com/cwm/adapters/<vendor>/<adapter-name>").

Option	Data type	Requirement	Description
`-ignore-template`		optional	skip generation of example code in the go and proto files.
`-location`	string	optional	point to adapter location (default: current directory).
`-os-architecture`	string	optional	define architecture in which adapter is developed. Valid options are: 'linux','mac-intel','mac-arm' and 'windows' (default: "linux").
`-product`	string	required	provide name for the go module corresponding to the product name you create an adapter for.
`-vendor`	string	optional	provide unique name for the company creating the adapter (default "cisco").
`-verbose`	string	optional	output progress info. Options are: `off`, `on` and `very` (default "off").

Output

Once the command is executed, verify the generated output inside the new adapter directory:

- `<adapter-name>/adapter.properties`
- `<adapter-name>/go/go.mod`
- `<adapter-name>/proto/<vendor\>.<product\>.common.adapter.proto`
- `<adapter-name>/proto/<vendor\>.<product\>.<feature\>.adapter.proto` (if you defined the `-feature` option)
- `<adapter-name>/Makefile`

Extend adapter with features

To add a feature (a **go package**) for an adapter, open a terminal and from your main adapter directory, run:

```
cwm-sdk extend-adapter [options] -feature <feature_name>
```

Options

Option	Data type	Requirement	Description
`-activity`	string	optional	Provide name for a new activity to add.
`-feature`	string	required	Provide name for the feature to add (default: "<adapter-name>").
`-ignore-template`		optional	Skip generation of example code in the go and proto files.
`-location`	string	optional	Point to the location of adapter to which you add the feature (default: current directory).
`-verbose`	string	optional	Output progress info. Options are: `off`, `on` and `very` (default "off").

Output

Once the command is executed, verify the generated output inside the new adapter directory:

- `<adapter-name>/proto/<vendor>.<module>.<package>.adapter.proto`

Generate/update activity

Once you have an adapter with at least one feature added, you can proceed to creating activities. Activities are defined within the `.proto` file for a specific feature (**go package**). You can do this manually or use the OASX extension for OpenAPI-enabled services to automatically build of message logic in the `.proto` files.

Once the activities are defined, you can generate the input and output files for the adapter. Go to your main adapter directory and run:

```
cwm-sdk update-adapter
```

Options

- `-features` *string* - provide a comma-separated list of features to update.
- `-location` *string* - point to location of adapter to update (default: current directory).
- `-verbose` *string* - output progress info. Options are: `off`, `on` and `very` (default "off").

Output

Once the command is executed, verify the generated output inside the adapter directory:

- `go/<feature\>/<vendor>.<product>.<feature>.adapter.pb.go`
- `go/common/<vendor>.<product>.common.adapter.pb.go`

The `.pb.go` files contain **go** structs defining the input and output parameters of the adapter. They shouldn't be altered manually.

Once the command is executed, verify the generated output inside the adapter directory:

- `go/<feature\>/activities.go`

The `activities.go` file contains stubs for the gRPCs defined in the `.adapter.proto`. Once generated, you can add functionality to the activities by defining the message.

Upgrade adapter

To upgrade the **go module** to contain matching versions for **go** and required imports, go to the root directory of your adapter and run:

```
cwm-sdk upgrade-adapter [options]
```

Options

- `-cwm-version` *string* - provide the version of CWM to upgrade to (default is latest).
- `-location` *string* - point to location of adapter to upgrade (default: current directory).

- `-verbose` *string* - output progress info. Options are: `off`, `on` and `very` (default "off").

Output

- go/go.mod

The `go.mod` file module will be modifed allowing the adapter to be installed correctly.

Export library to local directory

The `cwm-sdk` uses the SDK go module for performing tasks. In certain cases you might want to have the SDK go module created in the adapter directory beforehand. For this purpose, use the `export-lib` command.

The `export-lib` command comes with the following options:

Option	Data type	Description	Status
-location	string	Provide location where SDK lib should be created. (default: current directory)	optional
-verbose	string	Show command progress info. Options are: off, on, or very.	optional

Create adapter installable version

To create a `tar.gz` archive for installing your adapter for different operating systems, go to the root directory of your adapter and run:

```
cwm-sdk create-installable [options]
```

Generates code based on the proto file.

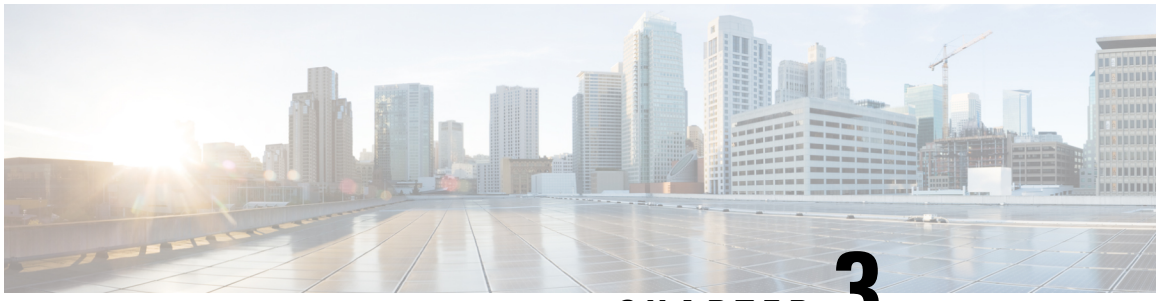
Options

- `-cwmversion` *string* - provide a CWM version to match the created installable (default is latest).
- `-location` *string* - point to where the installable should be created (default: current directory).
- `-verbose` *string* - output progress info. Options are: `off`, `on` and `very` (default "off").

Output

- out/<vendor>-<product>-v<X.Y.Z>.tar.gz

!!! note The generated archive contains the adapter **go** module and proto files. The **go** module is modified using the **go** vendor command in order to not have any external dependencies.



CHAPTER 3

Adapter example

This section contains the following topics:

Step 1: Create a new adapter

In a terminal window, open a command-line terminal and run:

```
cwm-sdk create-adapter -vendor vendor1 -product product1 -feature feature1
```

Now you have a new catalog named `vendor1.product1` at your home dierctory with the following contents:

```
Makefile
adapter.properties
docs
go
proto

 ./docs:
  index.html
 ./go:
  common
  go.mod
  feature1

  ./go/common:
   errors.go
   logger.go
  ./go/feature1:

./proto:
 vendor1.product1.common.adapter.proto
 vendor1.product1.feature1.adapter.proto
```

Step 2: Define mock activity

The Adapter SDK has generated the `.proto` files. In the `vendor1.product1.feature1.adapter.proto` file, define the interface of the adapter:

Step 1 Open the `vendor1.product1.feature1.adapter.proto` file with a text editor or inside a terminal window. The contents are as below.

```
syntax = "proto3";

package vendor1.product1.feature1;

option go_package = "cisco.com/cwm/adapters/vendor1/product1/feature1";

import "google/protobuf/struct.proto";

service Activities {

 // CWM SDK NOTE: Activity functions are defined as RPCs here e.g.

 /* Documentation for MyActivity */
 rpc MyActivity(MyRequest) returns (MyResponse);
}

// CWM SDK NOTE: Messages here e.g.

/* Documentation for MyRequest */
message MyRequest {
 string                stringInput  = 1;
 int32                 integerInput = 2;
 bool                  booleanInput = 3;
 google.protobuf.Value anyInput     = 4; // CWM SDK NOTE: Useful for accepting a json object from
the workflow definition
}

/* Documentation for MyResponse */
message MyResponse {
 string                stringOutput  = 1;
 int32                 integerOutput = 2;
 bool                  booleanOutput = 3;
 google.protobuf.Value anyOutput     = 4; // CWM SDK NOTE: Useful for returning a json object to
the workflow definition
}
```

Step 2 To define your activity, replace the placeholder 'MyActivity' with a mock 'Hello' activity, along with the MyRequest and MyResponse placeholder names and message parameters as shown below:

```
service Activities {
 /* Documentation for Hello Activity */
 rpc Hello(MyRequest) returns (MyResponse);
}

/* Documentation for MyRequest */
message MyRequest {
 string name = 1;
}

/* Documentation for MyResponse */
message MyResponse {
```

```
 string message = 1;
}
```

Step 3: Generate adapter source code

Step 1 Based on the `adapter.proto` file that you have edited and on the remaining `.proto` files, generate the source **go** code for the adapter and inspect the files. In the main adapter directory, run:

```
cwm-sdk update-adapter && ls
```

```
The output will look like:

 .go/
  common
  go.mod
  feature1
  main.go

  go//common:
  errors.go
  logger.go
  vendor1.product1.common.adapter.pb.go

  go//feature1:
  activities.go
  adapter.go
  vendor1.product1.feature1.adapter.pb.go
```

Step 2 **Note** The `.adapter.pb.go` files should not be edited manually.

The `.adapter.pb.go` files generated using the **Protobufs compiler** define all the messages from the `adapter.proto` files.

Step 3 The generated `activities.go` file contains stubs for all the RPCs you have defined in the `.adapter.proto` file. Open the file:

```
 package feature1

 import (
  "cisco.com/cwm/adapters/vendor1/product1/common"
  "context"
 )

 func (adp *Adapter) Hello(ctx context.Context, req *MyRequest, cfg *common.Config) (*MyResponse,
error) {

  var res *MyResponse
  var err error

  // CWM SDK NOTE: Implement your activity logic here...

  return res, err
 }
```

Step 4 Edit the file to return a message:

```
func (adp *Adapter) Hello(ctx context.Context, req *MyRequest, cfg *Config) (*MyResponse, error) {
 return &MyResponse {Message: "Hello, " + req.GetName() + "!"}, nil
}
```

Define another activity

If you wish to add another activity to the existing feature set (**go** package):

Step 1 Open and edit the `adapter.proto` file and define another activity underneath the existing one:

```
service Activities {
 rpc Hello(MyRequest) returns (MyResponse);
 rpc Fancy(MyRequest) returns (MyResponse);
}
```

Step 2 Update the activities **go** code using the SDK:

```
cwm-sdk extend-adapter -activity fancy -feature feature1
```

After you update the **fancy** activity part of the `.adapter.proto` file with a sample logic, update the adapter:

```
cwm-sdk update-adapter
```

Once the code is generated, the `activities.go` file is updated with the new 'Fancy' activity stub, while the code for the 'Hello' activity remains.

Step 4: Add another feature

If you wish to add another feature (**go** package) to the example adapter, use the `extend-adapter` command. In the main adapter directory, run:

```
cwm-sdk extend-adapter -feature feature2
```

Step 1 A new `vendor1.product1.feature2.adapter.proto` file has been added for your adapter:

```
.proto/
  vendor1.product1.common.adapter.proto
  vendor1.product1.feature2.adapter.proto
  vendor1.product1.feature1.adapter.proto
```

Step 2 To define activities for the new feature, open the `vendor1.product1.feature2.adapter.proto` file, and modify the contents accordingly:

```
syntax = "proto3";

package vendor1.product1.feature2;

option go_package = "cisco.com/cwm/adapters/vendor1/product1/feature2";

import "google/protobuf/struct.proto";

service Activities {
   /* Documentation for Goodbye Activity */
```

```
 rpc Goodbye(MyRequest) returns (MyResponse);
}

/* Documentation for MyRequest */
message MyRequest {
 string name = 1;
}

/* Documentation for MyResponse */
message MyResponse {
 string message = 1;
}
```

Step 3 Generate the code for the 'feature2' package and activities.

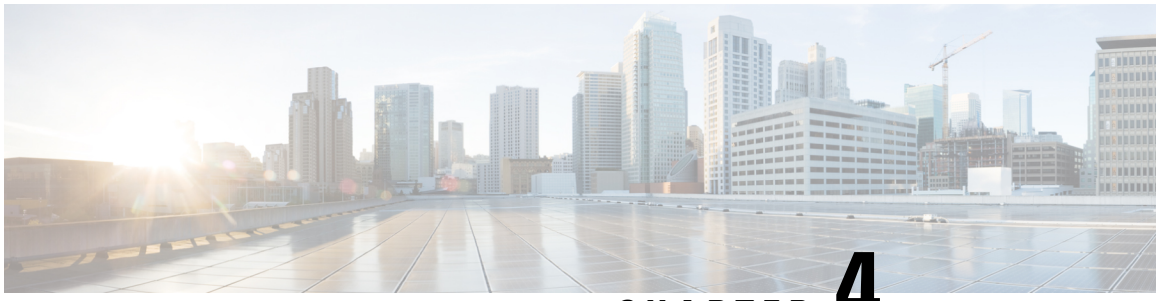
```
cwm-sdk update-adapter -features feature2
```

```
.go/goodbyes
 activities.go
 adapter.go
 vendor1.product1.feature2.adapter.pb.go
```

Step 5: Create an installable archive

```
cwm-sdk create-installable
```

The generated `tar.gz` archive contains the all required files of the adapter and can be installed in CWM. The **go** vendor command has been executed in order to eliminate any external dependencies.



CHAPTER 4

Adapter XDK

This section contains the following topics:

Adapter XDK for Cisco NSO

The Adapter XDK for NSO (`cwm-nsox`) is an application that helps you generate interfaces and logic for custom adapters intended to interact with the Cisco Network Services Orchestrator(NSO).

The primary purpose of `cwm-nsox` is to reduce the time-consuming and error-prone manual process of constructing paths and payloads required for CWM to communicate with NSO.

The tool complements the Adapter SDK and is able to automatically define interfaces in .proto files and implementation of logic in the adapter go module. This is achieved by parsing yang files and *points of interest* provided by the Adapter Developer.

Prerequisites

- Installed Adapter SDK and dependencies.
- The NSO `src/ncs/yang` folder for yang module imports. If you don't have it, you may install Cisco NSO to get it as part of a full installation.

Get `cwm-nsox`

The `cwm-nsox` is a binary that comes with the Crosswork Workflow Manager Software Package.

Go to Cisco Software Download page to download the .tar file with the CWM Software Package, where the `cwm-nsox` resides. Unpack the .tar and move the contents of the `adapters` folder to a desired location

!!! tip It's recommended that you put the binary in a common folder, with the `cwm-sdk` and other extension binaries like `cwm-oasx`.

Remember to include the location of the `cwm-nsox` binary by setting the environment variable path:

```
export PATH=/path/to/adapter-dev-binaries:$PATH
```

Use `cwm-nsox` for creating custom NSO adapter

The `cwm-nsox` works with yang models of NSO services and NEDs files to identfiy yang paths that you'd like to address using the adapter.

Step 1: Create an adapter stub with Adapter SDK

Run the following command to create a new adapter using SDK:

```
cwm-sdk create-adapter \
-vendor cisco \
-product nsox \
-feature services \
-ignore-template
```

Note The `ignore-template` option eliminates tips and descriptions from the generated .proto files.

Step 2: Display yang paths and adapter code

`display-paths`

Use the cwm-nsox `display-paths` command to extract paths for activities from a source yang file. With the `-src` option, you specify the desired yang configuration file:

```
cwm-nsox display-paths -src ../path/to/source/file.yang
```

You will see a list of yang paths based on which you can generate adapter activities.

`display-proto`

Optionally, use the `display-proto` command with the `-poi` option to display the output for the activity based on your chosen point of interest:

```
cwm-nsox display-proto \
-src ../path/to/source/file.yang \
-poi your-nsoservice:your-nsoservice-list=%s/example-leaf
```

The output will look similar to this:

```
service Activities {
    /*
     * Description for activity NsoActivity
     */
    rpc NsoActivity (NsoActivityRequest) returns (cisco.cwmlib.nso.Response);
}

/*
 * Description for NsoActivityRequest
 */
message NsoActivityRequest {
    string deviceName = 1; // devices/device={deviceName}
    optional string dummyLeaf = 2; // tailf-ned-cwm-dmycli:dummy-leaf
    cisco.cwmlib.nso.PutQuery queries = 3;
}
```

The following options are available:

Option	Data type	Description	Status
-deps	string	Define paths to yang imports (comma separated list).	optional
-poi	string	Point to desired yang path (point of interest)	required
-src	string	Point to path of yang configuration file.	required
-verbose	string	Show command progress info. Options are: off, on, or very.	optional

display-json

Optionally, use the `display-json` command with the required `-poi` and `-src` options to display the data payload for the activity based on your chosen point of interest (path):

```
cwm-nsox display-json \
-src ../path/to/source/file.yang \
-poi your-nsoservice:your-nsoservice-list=%s/example-leaf
```

Option	Data type	Description	Status
-deps	string	Define paths to yang imports (comma separated list).	optional
-poi	string	Point to desired yang path (point of interest)	required
-src	string	Point to path of yang configuration file.	required
-verbose	string	Show command progress info. Options are: off, on, or very.	optional

Step 3: Generate activity

Using the path defined in the previous section, you can now run the `generate-activity` command.

Go to the main directory of your adapter and execute the following command (adjust the path, activity name, request type and point-of-interest name accordingly):

```
 cwm-nsox generate-activity \
 -src ../path/to/source/file.yang \
 -feature services \
 -activity NsoActivity \
 -request PUT \
 -poi your-nsoservice:your-nsoservice-list=%s/example-leaf
```

This will generate a new adapter activity with a predefined rpc and I/O messages in the .proto files (see example in the section above), as well as a ready-to-execute implementation in the .go files.

Here's an example of the output generated by the `cwm-nsox` and inserted in the activities.go file:

```
package services

import (
 "context"

 "www.cisco.com/cwm/adapters/cisco/nsox/common"
 "www.cisco.com/cwm/sdk/adapters/logger"
 "www.cisco.com/cwm/sdk/adapters/nso"
)
```

```
func (adp *Adapter) NsoActivity(ctx context.Context,
    req *NsoActivityRequest, cfg *common.Config) (*nso.Response, error) {
 logger.GetLogger(ctx).Info("Activity cisco.nsox.services.NsoActivity called...")
 return nso.SendCustomRequest(ctx, req, cfg)
}
```

Create activity (optional)

Optionally, you can create a new activity for a selected feature but without indicating the source file. This will create an activity implementation in the .go files and a stub for you to fill in the logic inside the .proto file:

```
cwm-nsox create-activity \
-feature device \
-activity TestActivity \
-request GET \
```

Here's an example of the activity stub generated by the `cwm-nsox` inserted in the .proto file:

```
service Activities {

...

    /*
     * Description for activity Testactivity
     */
    rpc Testactivity (TestactivityRequest) returns (TestactivityResponse);

}
...

/*
 * Description for TestactivityRequest
 */
message TestactivityRequest {
    // NOTE: Developer needs to set vars
}

message TestactivityResponse {
    // NOTE: Developer needs to set vars
}
```

Step 4: Test your adapter

To test your adapter, generate an installable file and install the adapter in CWM.

Generate installable

Go to the main directory of your adapter and run the following command:

```
cwm-sdk create-installable
```

Test adapter activity

This will create a .tar file that can be then uploaded into CWM. Follow the detailed instructions in the Install NSO adapter section to install and deploy the adapter, then run a workflow that uses the newly added adapter activity.

Generate installable

Go to the main directory of your adapter and run the following command:

```
cwm-sdk create-installable
```

Test adapter activity

The command will produce a .tar file that can be then installed in CWM and tested for proper functioning.

Adapter XDK for OpenAPI

Use Adapter XDK for OpenAPI (`cwm-oasx`) to automatically build interfaces and message logic for custom adapters that require communicating with OpenAPI-based systems. With the `cwm-oasx` tool, you point to an OpenAPI operation defined in JSON, which `cwm-oasx` will then use to generate a new adapter activity with a predefined rpc and I/O messages in the .proto files, as well as a ready-to-execute implementation in the adapter .go files.

Prerequisites

- Installed Adapter SDK and dependencies.
- A JSON or YAML schema file of an OpenAPI or Swagger-enabled API.

Get `cwm-oasx`

The `cwm-oasx` is a binary that comes with the Crosswork Workflow Manager Software Package.

Go to Cisco Software Download page to download the .tar file with the CWM Software Package, where the `cwm-oasx` resides. Unpack the .tar and move the contents of the `adapters` folder to a desired location.

!!! tip It's recommended that you put the binary in a common folder, with the `cwm-sdk` and other extension binaries like `cwm-nsox`.

Remember to include the location of the cwm-oasx binary by setting the environment variable path:

```
export PATH=/path/to/adapter-dev-binaries:$PATH
```

Use `cwm-oasx` for implementing adapter activities

The `cwm-oasx` works with OpenAPI JSON/YAML schemas to identify endpoint paths and methods that you'd like to call using the adapter. Follow this instruction to create a single adapter activity based on a single API path and method.

Note As an example, we'll use the NetBox REST API schema in YAML format downloaded from the NetBox Swagger API. Using a JSON schema is also supported.

Step 1: Create an adapter stub with Adapter SDK

Run the following command to create a new adapter using SDK:

```
cwm-sdk create-adapter \
-vendor cisco \
-product oasx \
-feature services \
-ignore-template
```

Note The `ignore-template` option eliminates tips and descriptions from the generated .proto files.

Step 2: Display paths and adapter code

display-paths

Use the cwm-oasx `display-paths` to extract paths and methods for activities from a source JSON/YAML file. Use the `-src` option to point to the desired JSON/YAML API schema file:

```
cwm-oasx display-paths -src ../path/to/source/NetBox_REST_API.yaml
```

A list of paths will be displayed. In the example, we're interested in first of the four that define operations on `ipam/prefixes`:

```
/api/ipam/prefixes/ : [GET POST PUT PATCH DELETE]
/api/ipam/prefixes/{id}/ : [GET PUT PATCH DELETE]
/api/ipam/prefixes/{id}/available-ips/ : [GET POST]
/api/ipam/prefixes/{id}/available-prefixes/ : [GET POST]
```

display-proto

Optionally, use the `display-proto` command with the `-oper`, `-path`, and `-src` options to display the output for the activity based on your chosen point of interest:

```
cwm-oasx display-proto \
-oper POST \
-path /api/ipam/prefixes/ \
-src ../path/to/source/NetBox_rest.yaml
```

The output will look similar to this:

```
Proto messages for activity:
 message ProtoRequest {
  message Data {
   optional string comments = 1;
   optional string customFields = 2;
   optional string description = 3;
   optional bool isPool = 4; // All IP addresses within this prefix are considered usable
   optional bool markUtilized = 5; // Treat as 100% utilized
   string prefix = 6;
   optional int32 role = 7; // The primary function of this prefix
   optional int32 site = 8;
   optional string status = 9; // Operational status of this prefix\n\n* `container` -
Container\n* `active` - Active\n* `reserved` - Reserved\n* `deprecated` - Deprecated
   message Tags {
    optional string color = 1;
    string name = 2;
    string slug = 3;
   }
   repeated Tags tags = 10;
   optional int32 tenant = 11;
   optional int32 vlan = 12;
   optional int32 vrf = 13;
  }
  Data data = 3;
 }

 message ProtoResponse {
  int32 status = 1;
```

```
  google.protobuf.Value data = 2;
 }
```

Note that three of four available options are required:

Option	Data type	Description	Status
-oper	string	Point to specific operation: GET, POST, PUT, PATCH or DELETE.	required
-path	string	Point to specific API path.	required
-src	string	Point to desired JSON API schema file.	required
-verbose	string	Show command progress info. Options are: off, on, or very.	optional

display-json

Optionally, use the `display-json` command with the required `-oper`, `-path`, and `-src` options to display the data payload for the activity based on your chosen point of interest (path):

```
cwm-oasx display-json \
-oper POST \
-path /api/ipam/prefixes/ \
-src ../path/to/source/NetBox_rest.yaml
```

The output will look similar to this:

```
Data payload for activity:
{
 "comments": "%s",
 "custom_fields": "{{'{'}}%s{{'}'}}",
 "description": "%s",
 "is_pool": %t,
 "mark_utilized": %t,
 "prefix": "%s",
 "role": %d,
 "site": %d,
 "status": "%s",
 "tags": [
  "color": "%s",
  "name": "%s",
  "slug": "%s"
 ],
 "tenant": %d,
 "vlan": %d,
 "vrf": %d
}
```

Note that three of four available options are required:

Option	Data type	Description	Status
-oper	string	Point to specific operation: GET, POST, PUT, PATCH or DELETE.	required
-path	string	Point to specific API path.	required
-src	string	Point to desired JSON API schema file.	required
-verbose	string	Show command progress info. Options are: off, on, or very.	optional

Step 3: Generate activity

Using the path defined in the previous section, you can now run the `generate-activity` command.

Go to the main directory of your adapter and execute the following command (adjust the feature and activity name, operation, path, and source accordingly):

```
cwm-oasx generate-activity \
-feature services \
-activity PostPrefix \
-oper POST \
-path /api/ipam/prefixes/ \
-src ../path/to/source/NetBox_rest.yaml
```

This will generate a new adapter activity with a predefined rpc and I/O messages in the .proto files (see example in the `display-proto` section above), as well as a ready-to-execute implementation in the .go files. Here's an example of the function generated by the `cwm-oasx` and inserted in the activities.go file:

```
func (adp *Adapter) PostPrefix(ctx context.Context, req *PostPrefixRequest, cfg
*common.Config) (*PostPrefixResponse, error) {
 return oas.SendRequest[*PostPrefixResponse](ctx, req, cfg.GetResource(), cfg.GetSecret())
}
```

Create activity (optional)

Optionally, you can create a new activity for a selected feature but without indicating the source json/yaml file. This will create an activity implementation in the .go files and a stub for you to fill in the logic inside the .proto file:

```
cwm-oasx create-activity \
-feature services \
-activity TestActivity \
-oper POST \
```

Here's an example of the activity stub generated by the `cwm-oasx` inserted in the .proto file:

```
service Activities {

...

    /*
     * Description for activity Testactivity
     */
    rpc Testactivity (TestactivityRequest) returns (TestactivityResponse);

}
...

/*
 * Description for TestactivityRequest
 */
message TestactivityRequest {
    // NOTE: Developer needs to set vars
}

message TestactivityResponse {
    // NOTE: Developer needs to set vars
}
```

Step 4: Generate feature (optional)

Use this command to bulk create activities for new or existing features. If you point to a path, `generate-feature` will pick all the endpoints existing down this path and generate activity code based on each, for all available methods. Set `verbose` to `on` or `very` to see details of command execution.

For example, let's use the CWM JSON API specification and pass `/secret` as the path parameter for feature `services`.

```
cwm-oasx generate-feature \
-src ../path/to/source/cwm.json \
-feature services \
-path /secret \
-verbose on
```

See the **sample** to see what the generated output will look like in the activities.go file:

go package services

```
import (
 "context"

 "cisco.com/cwm/lib/xdk/oas"
 "cisco.com/cwm/adapters/cisco/oasx/common"
)

func (adp *Adapter) GetType(ctx context.Context, req *GetTypeRequest, cfg *common.Config)
(*GetTypeResponse, error) {
 return oas.SendRequest[*GetTypeResponse](ctx, req, cfg.GetResource(), cfg.GetSecret())
}

func (adp *Adapter) Get(ctx context.Context, req *GetRequest, cfg *common.Config)
(*GetResponse, error) {
 return oas.SendRequest[*GetResponse](ctx, req, cfg.GetResource(), cfg.GetSecret())
}

func (adp *Adapter) Post(ctx context.Context, req *PostRequest, cfg *common.Config)
(*PostResponse, error) {
 return oas.SendRequest[*PostResponse](ctx, req, cfg.GetResource(), cfg.GetSecret())
}

func (adp *Adapter) GetWithSecretId(ctx context.Context, req *GetWithSecretIdRequest, cfg
*common.Config) (*GetWithSecretIdResponse, error) {
 return oas.SendRequest[*GetWithSecretIdResponse](ctx, req, cfg.GetResource(),
cfg.GetSecret())
}

func (adp *Adapter) PatchWithSecretId(ctx context.Context, req *PatchWithSecretIdRequest,
cfg *common.Config) (*PatchWithSecretIdResponse, error) {
 return oas.SendRequest[*PatchWithSecretIdResponse](ctx, req, cfg.GetResource(),
cfg.GetSecret())
}

func (adp *Adapter) DeleteWithSecretId(ctx context.Context, req *DeleteWithSecretIdRequest,
 cfg *common.Config) (*DeleteWithSecretIdResponse, error) {
 return oas.SendRequest[*DeleteWithSecretIdResponse](ctx, req, cfg.GetResource(),
cfg.GetSecret())
}

func (adp *Adapter) GetTypeWithSecretTypeId(ctx context.Context, req
*GetTypeWithSecretTypeIdRequest, cfg *common.Config) (*GetTypeWithSecretTypeIdResponse,
error) {
 return oas.SendRequest[*GetTypeWithSecretTypeIdResponse](ctx, req, cfg.GetResource(),
```

```
cfg.GetSecret())
}
```

The `generate-feature` command comes with the following options:

Option	Data type	Description	Status
-src	string	Point to desired JSON/YAML API schema file.	required
-feature	string	Give name of adapter feature to be updated.	required
-path	string	Point to specific API path.	required
-verbose	string	Show command progress info. Options are: off, on, or very.	optional

Export XDK module to local directory

The `cwm-oasx` uses the XDK go module for performing tasks, and some of them can share some of the resources with the NSOX extension. While the XDK module is exported to the directory of your adapter upon executing the `generate-activity` command, in certain cases you might want to have the XDK go module created in the adapter directory beforehand. For this purpose, use the `export-lib` command.

The `export-lib` command comes with the following options:

Option	Data type	Description	Status
-location	string	Provide location where XDK lib should be created. (default: current directory)	optional
-verbose	string	Show command progress info. Options are: off, on, or very.	optional

Generate installable

Go to the main directory of your adapter and run the following command:

```
cwm-sdk create-installable
```

Test adapter activity

The command will produce a .tar file that can be then installed in CWM and tested for proper functioning.