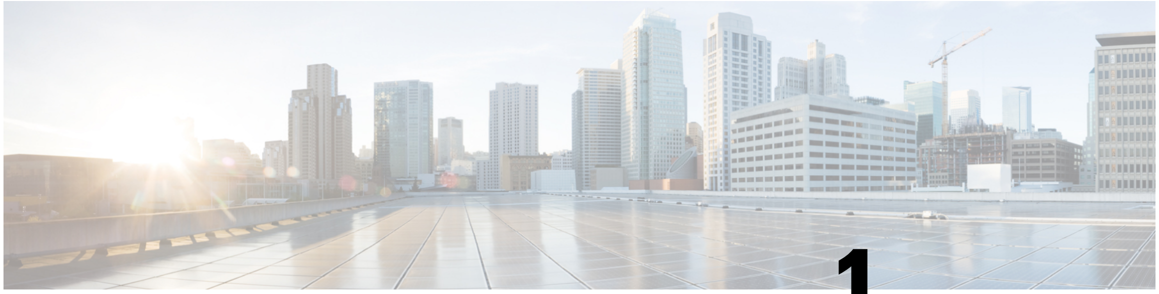
Cisco Crosswork Workflow Manager 1.0 Adapter Developer guide

First Published: 2023-06-01

Last Modified: 2023-07-25

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 two pre-built adapters which are available as part of the CWM offering. These ready-made solutions include: the Network Services Orchestrator adapter and a generic REST API adapter.

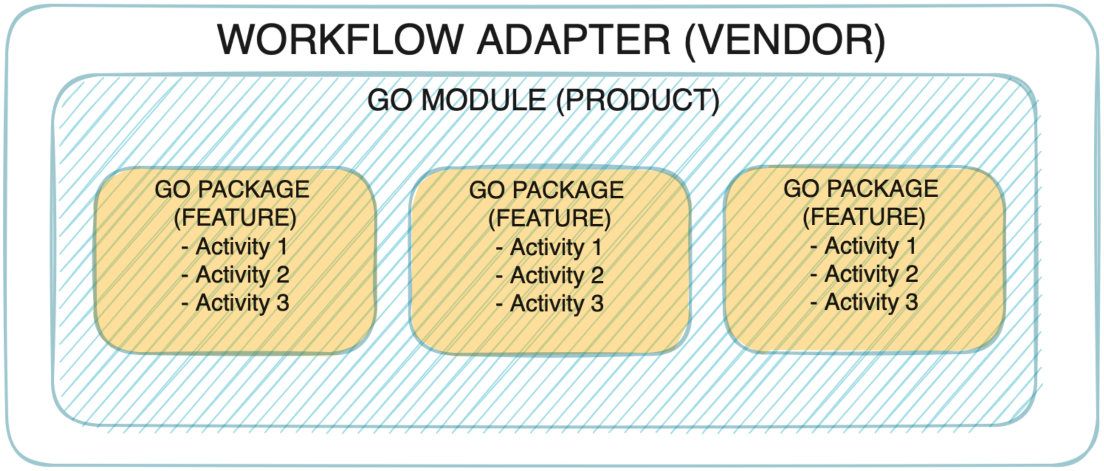
What's in an adapter

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 structure


WORKFLOW ADAPTER (VENDOR)
GO MODULE (PRODUCT)
GO PACKAGE (FEATURE) - Activity 1 - Activity 2 - Activity 3
GO PACKAGE (FEATURE) - Activity 1 - Activity 2 - Activity 3
GO PACKAGE (FEATURE) - Activity 1 - Activity 2 - Activity 3

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

Property	Description
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 Workflow 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 Go

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/.

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 plugins

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
```

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

```
go install github.com/chrusty/protoc-gen-jsonschema/cmd/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 resides.

Include the location of cwm-sdk-binaries by setting the environment variable path:

```
export PATH=/path/to/cwm-sdk-binaries:$PATH
```

Note Remember to replace the /path/to/ with your actual path.

Overview of commands

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

- `cwm-sdk create-adapter` - use it to create a go module with a package and the corresponding .proto files).
- `cwm-sdk extend-adapter` - use it to add a new feature to an existing adapter (go package and .proto files).
- `make generate-model` - generate activities, input and output (go code).
- `make generate-code` - 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 a new adapter

To create an adapter, open a terminal and from the `cwmsdk` repository directory, run:

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

Options

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

- `-exclude-resource` - skip creation of the `.resource.proto` file from template.
- `-go-module` *string* - provide name for the module assigned to the go.mod file (default: "www.cisco.com/cwm/adapters/<vendor>/<adapter-name>").
- `-feature` *string* - provide name for the go package assigned to activities (default: "<adapter-name>").
- `-location` *string* - point to adapter location (default: current directory).
- `-os-architecture` *string* - define architecture in which adapter is developed. Valid options are: 'linux','mac-intel','mac-arm' and 'windows' (default: "linux").
- `-vendor` *string* - provide unique name for the company creating the adapter (default "cisco").

- `-product` *string* - provide name for the go module corresponding to the product name you create an adapter for (required).

Output

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

- `<adapter-name>/go/go.mod`
- `<adapter-name>/proto/<vendor\>.<module\>.<package\>.adapter.proto`
- `<adapter-name>/proto/<vendor\>.<module\>.<package\>.resource.proto` (if `-exclude-resource` option wasn't used)
- `<adapter-name>/Makefile`

Extend adapter with features

To add a feature (a **go package**) for an adapter, open a terminal and from the `cwmsdk` repository directory, run:

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

Options

- `-exclude-resource` - skip creation of the `.resource.proto` file from template.
- `-location` *string* - point to the location of the adapter to be extended by the new package (default: current directory).

Output

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

- `<adapter-name>/proto/<vendor>.<module>.<package>.adapter.proto`
- `<adapter-name>/proto/<vendor>.<module>.<package>.resource.proto` (if `-exclude-resource` option wasn't used)

Generate input and output

To generate the input and output files for the adapter, go to the root directory of your adapter and run:

```
make generate-model
```

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. It should not be altered manually.

Generate activities

To generate the previously defined activities, go to the root directory of your adapter and run: `make generate-code`

Output

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

- go/<package>/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 an adapter

To upgrade the **go module** to contain matching versions for go and required imports, open a terminal and from the `cwmsdk` repository directory, run:

"Linux" `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).

Output

- go/go.mod

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

Release an installable adapter

To create an archive for installing your adapter for different operating systems, open a terminal and from the `cwmsdk` repository directory, run:

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

This generates code based on the proto file.

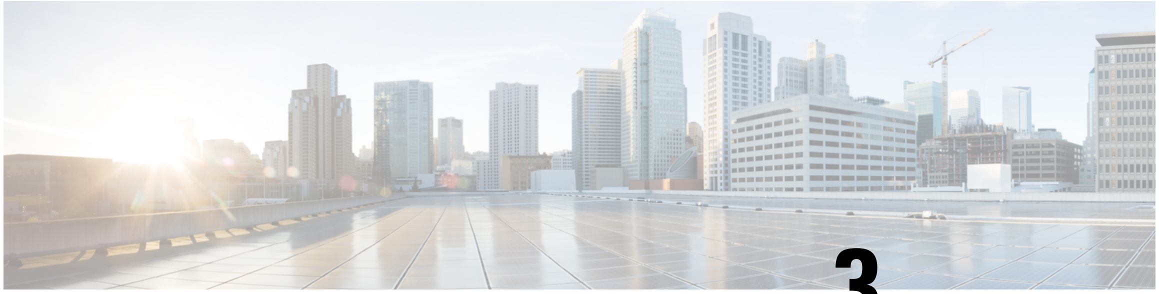
Options

- `-location` *string* - point to location for the adapter installable file (default ".").

Output

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

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:

Adapter example

This tutorial is a step-by-step instruction on building an example adapter using the Workflow Adapter SDK. It gives an idea on the adapter structure and on how you provide input to define adapter activities to be consumed by a workflow worker. Before you start, you need to go through the Prerequisites section to set up your development environment.

Step 1: Create new adapter

In a terminal window, open your `cwmsdk` repository directory and run:

```
cwm-sdk create-adapter -location ~/your_repo/adapters -vendor companyX -feature featureX
-product productX
```

Now you have a directory in `adapters` named `companyX.productX` with the following contents:

```
Makefile
adapter.properties
go
proto

 ./go:
  common
  go.mod
  featureX

  ./go/common:

  ./go/featureX:

 ./proto:
  cisco.cwm.sdk.resource.proto
  companyX.productX.common.adapter.proto
  companyX.productX.featureX.adapter.proto
```

Step 2: Define mock activity

The Adapter SDK has generated the `.proto` files. In the `companyX.productX.featureX.adapter.proto` file, define the interface of the adapter:

Step 1 Open the `companyX.productX.featureX.adapter.proto` file with a text editor or inside a terminal window. The contents are as below.

```
syntax = "proto3";

package productXfeatureX;

option go_package = "www.cisco.com/cwm/adapters/companyX/productX/featureX";

service Activities {
 // NOTE: Activity functions are defined as RPCs here e.g.

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

// NOTE: Messages here e.g.

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

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

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(Request) returns (Response);
}

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

/* Documentation for Response */
message Response {
 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:

```
make generate-model && ls
```

```
.go/
 common
 go.mod
 featureX

 go//common:
 companyX.productX.common.adapter.pb

 go//featureX:
 companyX.productX.featureX.adapter.pb
```

```
The `.adapter.pb.go` files generated using the **Protobufs compiler** define all the messages from
the `adapter.proto` files.
!!! caution
 The `.adapter.pb.go` files should not be edited manually.
```

Step 2 Now generate the **go** code for the defined activities. In the main adapter directory, run:

```
make generate-code && ls
```

```
.go/
 common
 go.mod
 featureX
 main.go

 go//common:
 companyX.productX.common.adapter.pb.go

 go//featureX:
 activities.go
 adapter.go
 companyX.productX.featureX.adapter.pb.go
```

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

```
package featureX

import (
 "context"
 "errors"
 "go.temporal.io/sdk/activity"
)

func (adp *Adapter) Hello(ctx context.Context, req *Request, cfg *Config) (*Response, error) {

 activity.GetLogger(ctx).Info("Activity Hello called")

 var res *Response
 var err error

 if ctx == nil {
  return nil, errors.New("Invalid context")
 }

 if req == nil {
  return nil, errors.New("Invalid request")
 }

 if cfg == nil {
  return nil, errors.New("Invalid config")
 }
 cancel := ctx.Done()
 done := make(chan any)

 go func() {

  //
  // NOTE:
  //
  // Enter activity code to set response and error here...
  //
  // Perform step 1
  //
  // ...
  //
  // activity.activity.RecordHeartbeat(ctx, "Activity completed step 1")
  //
  // Perform step 2
  //
  // ...
  //
  // activity.activity.RecordHeartbeat(ctx, "Activity completed step 2")
  //
  // ...
  //
  // All logic steps are completed
  //

  done <- nil
 }()

 //
 // NOTE
 //
 // For a long running call heartbeats can be recorded in a separate
 //
 // go func () {
 //     for {
 //         activity.RecordHeartbeat(ctx, "Activity is running")
 //         // TODO sleep for some interval
 //     }
 // } ()
 //

 for {
  select {
  case <-cancel:

   //
   // NOTE
   //
   // Execute any cleanup required for a canceled activity here...
   //

   return nil, errors.New("Activity was canceled")
  case <-done:
   return res, err
  }
 }
}
```

Step 3 Edit the file to return a message:

```
go func() {

 res = &Response {Message: "Hello, " + req.GetName() + "!"}
 err = nil

 done <- 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(Request) returns (Response);
 rpc Fancy(Request) returns (Response);
}
```

Step 2 Update the activities go code using the SDK:

```
make generate-code
```

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. Open your `cwmsdk` repository directory in a terminal and run:

```
cwm-sdk extend-adapter -feature featureY
```

Step 1 A new `companyX.productX.featureY.adapter.proto` file has been added for your adapter:

```
.proto/
  cisco.cwm.sdk.resource.proto
  companyX.productX.common.adapter.proto
  companyX.productX.featureY.adapter.proto
  companyX.productX.featureX.adapter.proto
```

Step 2 To define activities for the new feature, open the `companyX.productX.featureY.adapter.proto` file, and modify the contents accordingly

```
syntax = "proto3";

package companyXproductX;

option go_package = "www.cisco.com/cwm/adapters/companyX/productX/featureY";

service Activities {
 /* Documentation for Goodbye Activity */
 rpc Goodbye(Request) returns (Response);
}
```

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

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

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

```
make generate-model && generate-code && ls
```

```
.go/goodbyes
 activities.go
 adapter.go
 companyX.productX.featureY.adapter.pb.go
```

Step 5: Create an installable archive

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

The generated archive contains the all required files of the adapter. The **go** vendor command has been executed in order to eliminate any external dependencies.