Build your own

You can build your own Conduit standalone processors in Go using theProcessor SDK .

We currently only provide a Go SDK for processors. However, if you'd like to use another language for writing a processor, feel free toopen an issue and request a specific language SDK. You can also readhow standalone processors work under the hood and build an SDK yourself.

TheProcessor SDK exposes two ways of building processors, one for simple processors without configuration parameters, and another that gives you full control over the processor.

Usingsdk.NewProcesorFunc

If the processor is very simple and can be reduced to a single function (e.g. no configuration needed), then we can usesdk.NewProcessorFunc() to create a processor as below:

//go:build wasm

package main

import

(sdk "github.com/conduitio/conduit-processor-sdk")

func

main ( )

{ sdk . Run (& sdk . NewProcessorFunc (sdk . Specification { Name :

"simple-processor" }) , func (ctx context . Context , rec opencdc . Record)

(opencdc . Record ,

error)

{ // do something with the record return rec } ,) } However, if the processor needs configuration, or is more complicated than only one function, then we should use the full processor approach.

Usingsdk.Processor

To build the full-blown processor, the SDK contains an interface calledsdk.Processor that contains some methods to be implemented. These methods are:

Specification

Specification contains the metadata for the processor, which can be used to define how to reference the processor, describe what the processor does and the configuration parameters it expects. Here's a list of the fields in thesdk.Specification struct and their descriptions:

- Name
- : the name of the processor. Note that the name should be unique across all processors, as it's used to
- reference the processor in a pipeline (seereferencing processors
-).
- Version
- : the version of the processor. This should be a validsemver
- version and needs to be
- updated whenever the processor's behavior changes.
- Summary
- : a short description of what the processor does (ideally a one-liner).
- Description
- : a more detailed description of what the processor does. This field can contain markdown.
- Author
- : the author of the processor.
- Parameters

- : a map of the processor's configuration parameters. Each parameter should have a name, a type, a
- description, and a list of validations. Note that the validations defined on a parameter are automatically executed
- insdk.ParseConfig
- (seeConfigure
-).

Conduit also providesparamgen , a helpful tool that generates theParameters map from a Go struct. This allows us to create a configuration struct that contains the processor's parameters, define default values and validations using struct tags, and generate theParameters map. Check out theParamGen readme for more details.

info Note that a processor's name and version need to be unique across all processors, as they are used toreference the processor in a pipeline. If two processors have the same name and version, Conduit will refuse to load them.

Example without PramGen

You can define theSpecification method as below and manually define the parameters map:

package example

import

("context"

"github.com/conduitio/conduit-commons/config" sdk "github.com/conduitio/conduit-processor-sdk")

func

(p * AddFieldProcessor)

Specification (context . Context)

(sdk . Specification ,

error)

{ return sdk . Specification { Name :

"myAddFieldProcessor" , Summary :

"Add a field to the record." , Description :

This processor lets you configure a field that will be added to the record into field. If the payload is not structured data, this processor will panic. ,
Version :

"v1.0.0" , Author :

"John Doe" , Parameters :

map [string] config . Parameter { "field" :

{ Type : config . ParameterTypeString , Description :

"Field is the target field that will be set." , Validations :

[ ] config . Validation { config . ValidationRequired { } , } , } , "name" :

{ Type : config . ParameterTypeString , Description :

"Name is the value of the field to add." , Validations :

[ ] config . Validation { config . ValidationRequired { } , } , } , } , } ,

nil }

Example with PramGen

1. Add a struct that contains the needed parameters:

//go:generate paramgen -output=addField_paramgen.go addFieldConfig

type addFieldConfig struct

{ // Field is the target field that will be set. Field string

json:"field" validate:"required" // Name is the value of the field to add. Name string

json:"value" validate:"required" } 1. Generate the parameters by running:

paramgen -output=addField_paramgen.go addFieldConfig This will generate a file calledaddField_paramgen.go that contains the generated parameters map, which in turn can be used underspecification to make it simpler and shorter, example:

//go:generate paramgen -output=addField_paramgen.go addFieldConfig

type addFieldConfig struct

{ // Field is the target field that will be set. Field string

json:"field" validate:"required" // Name is the value of the field to add. Name string

json:"value" validate:"required" }

func

(p * AddFieldProcessor)

Specification (context . Context)

(sdk . Specification ,

error)

{ return sdk . Specification { Name :

"myAddFieldProcessor" , Summary :

"Add a field to the record." , Description :

This processor lets you configure a field that will be added to the record into field. If the payload is not structured data, this processor will panic. , Version :

"v1.0.0" , Author :

"John Doe" , Parameters : addFieldConfig { } . Parameters ( ) ,

// generated by paramgen } ,

nil }

Configure

Configure is the first function to be called in a processor. It provides the processor with the configuration that needs to be validated and stored to be used in other methods. This method should not open connections or any other resources. It should solely focus on parsing and validating the configuration itself.

To add custom validations, simply validate the parameters manually under this method, and return an error if theconfig map is not valid. On the other hand, using the utility function below would apply the builtin validations to the configuration.

TheProcessor SDK provides some useful utility functions to help implementing this method:

- sdk.ParseConfig
- : used to sanitize the configuration, apply defaults, validate it using builtin validations, and copy
- the values into the target object.
- sdk.NewReferenceResolver
- : creates a new reference resolver from the input string. The input string is a reference
- to a field in a record, checkReferencing record fields
- for more details.
- The method will return aresolver
- that can be used to resolve a reference to the specified field in a record and
- manipulate that field (get
- ,set
- anddelete
- the value, orrename
- the referenced field).

Using these utility functions, most of theConfigure method implementations would look something like:

```
func

( p * AddFieldProcessor )

Configure ( ctx context . Context , m map [ string ] string )

error

{ err := sdk . ParseConfig ( ctx , m ,

& p . config , addFieldConfig { } . Parameters ( ) ) if err !=

nil

{ return fmt . Errorf ( "failed to parse configuration: %w" , err ) }

resolver , err := sdk . NewReferenceResolver ( p . config . Field ) if err !=

nil

{ return fmt . Errorf ( "failed to parse the %q param: %w" ,

"field" , err ) } p . referenceResolver = resolver return

nil }
```

Open

This function is used to open connections, start background jobs, or initialize resources that are needed for the processor.

Note that implementing this function isoptional .

Process

Process is the main show of the processor, here we would manipulate the records received and return the processed ones.

After processing the slice of records that the function got, and if no errors occurred, it should return a slice ofsdk.ProcessedRecord that matches the length of the input slice. However, if an error occurred while processing a specific record, then it should be reflected in theProcessedRecord with the same index as the input record, and should return the slice at that index length.

For the interfacesdk.ProcessedRecord , there are three main processed records types:

1. sdk.SingleRecord
2. : is a single processed record that will continue down the pipeline.
3. sdk.FilterRecord
4. : is a record that will be acked and filtered out of the pipeline.
5. sdk.ErrorRecord
6. : is a record that failed to be processed and will be nacked.

Example:

```
func

( p * AddFieldProcessor )

Process ( ctx context . Context , records [ ] opencdc . Record )

[ ] sdk . ProcessedRecord { out :=

make ( [ ] sdk . ProcessedRecord ,

0 ,

len ( records ) ) for

_ , record :=

range records { resolver , err := p . referenceResolver . Resolve ( & record ) if err !=

nil
```

{ return

append (out , sdk . ErrorRecord { Error : err }) } err = resolver . Set (p . config . Name) if err !=

nil

{ return

append (out , sdk . ErrorRecord { Error : err }) } out =

append (out , sdk . SingleRecord (record)) } return out } Note thatProcess should be idempotent, as it may be called multiple times with the same records (e.g. after a restart when records were not flushed).

Teardown

This function acts like a counterpart toOpen , use this function to close any open connections or resources that were initialized underOpen .

Note that implementing this function is alsooptional .

Entrypoint

Since the processor will be run as a standaloneWASM plugin, we need to add an entrypoint to it. Also, we should add ago:build tag to ensure that this file is only included in the build when targeting WebAssembly.

the entrypoint will have to be in a separate package (i.e. folder), by Go convention it's normally undercmd/my-binary-name , so it would look something like:

. ├── my-processor.go # actual processor implementation └── cmd └── processor └── main.go # entrypoint Entry point example:

//go:build wasm

package main

import

(sdk "github.com/conduitio/conduit-processor-sdk" "github.com/conduitio/my-processor/example")

func

main ( )

{ sdk . Run (example . NewProcessor ( )) } CheckCompiling the processor for what to do next, and how to compile the processor.

Logging

You can get azerolog.logger instance from the context using thesdk.Logger function. This logger is pre-configured to append logs in the format expected by Conduit.

Keep in mind that logging in the hot path (i.e. in theProcess method) can have a significant impact on the performance of the processor, therefore we recommend using theTrace level for logs that are not essential for the operation of the processor.

Example:

func

(p * AddFieldProcessor)

Process (ctx context . Context , records [ ] opencdc . Record)

[ ] sdk . ProcessedRecord { logger := sdk . Logger (ctx) logger . Trace ( ) . Msg ("Processing records") // ... }

Compiling the processor

Conduit usesWebAssembly to run standalone processors. This means that we need to build the processor as a WebAssembly module. You can do this by setting the environment variablesGOARCH=wasm andGOOS=wasip1 when runninggo build . This will produce a WebAssembly module that can be used as a processor in Conduit.

So, to compile the processor, run:

GOARCH

wasm GOOS = wasip1 go build -o processor.wasm cmd/processor/main.go Congratulations! Now you have a new standalone processor. CheckStandalone processors for details on how to use your standalone processor in a Conduit pipeline.

note To see more standalone processor examples, check out ourexample processor repository . Edit this page Previous Standalone Processors Next How it works