Thought Machine

Thought Machine
Configuration Layer Utility User Guide

March 2023

Table of Contents

About this guide

Background

The Configuration Layer Utility (CLU) is the utility for applying configuration layer resources into Vault and:

- Automates the extraction and creation of resources, with robust, public Vault APIs offering consistent behaviour and greater control over created resources
- Provides greater visibility of import outcomes, with these displayed for each supported resource
- Uses a consistent and generic resource file format with a payload field that maps one-to-one to existing examples in the API documentation
- Allows dependencies between resource fields in different resources to be referenced after import

About the Configuration Layer Utility

The Configuration Layer Utility is a Command Line Interface for configuring resources in Vault. CLU provides a convenient interface which sits between the user and the Vault APIs. Instead of constructing and executing HTTP(S) requests to the Vault APIs directly, you instead specify a collection of resources which you can then apply using CLU. This manages many common pain points such as external file references, resource dependencies and request validation.

Vault Version compatibility

Before running any command, CLU checks whether it is compatible with the version of Vault it is being run against. A compatible Vault version needs to have the same major version as the version that CLU is designed to work with. In order to do this, the Core API URL must be specified. If there is a problem with compatibility, a warning message is displayed.

NOTE
Version 4.1 of CLU has been made available from Vault 4.6. While the tool works best in conjunction with Vault 4.6, it can be used in conjunction with other Vault versions of the 4.x release.

Purpose

This document describes:

- How to use CLU
- The resources supported in CLU
- How to define Vault resources for CLU to read

Scope

This document applies to version 4.1 of CLU only.

Audience

This document should be used by engineers managing the import of configuration layer to Vault.

Disclaimer

CLU quick reference

Technical details and prerequisites

CLU binary

- Linux binary: clu-linux-amd64
- macOS binary: clu-darwin-amd64

Binary prerequisites

CLU binary:

- Was tested on Ubuntu 18.04-amd64 and macOS Monterey
- Must be run on a machine which can access Vault's APIs

NOTE
The macOS binary is currently in a beta stage and therefore not signed. If you want to run the binary, you need to mark it as executable. MacOS Gatekeeper will also show warnings that it is not able to verify the authenticity of the binary.

Command overview

Command	Description
import	• Validates the configuration pack • Orders the resources by type and by dependencies • Synchronously creates resources in Vault
validate	• Reads all resources files from manifest.yaml file directory • Validates that fields in resource definition exist in the API resource • Validates that dependency references are defined against configuration

Supported global options

Command flag	Environment variable	Config file parameter	Description
`--config {filepath}`	N/A	N/A	The path to the file containing the CLU config.
`---nocolor`	CLU_NOCOLOR	nocolor	Removes all colour from the output.
`--output {text | json}`	CLU_OUTPUT	output	The output format for the command results.

CLU workflow

1. Check the resources you want to use are supported in the CLU.
2. Add new resources and/or copy each resource from the current configuration tool and format for the CLU.
3. Add the resource IDs to manifest.yaml.
4. Run the validate command to check all resources are valid.
5. Run the import command to synchronously create the resources in Vault.

   This displays the status of each resource, including whether it is valid, imported or if an error is generated.

   When the CLU binary finishes the import, the status (SUCCESS, FAILURE or PARTIAL SUCCESS) is displayed on the screen; the binary returns an exit code.

CLU configuration

Configuration methods

You can configure the CLU using any combination of command line arguments, environment variables and the configuration file. The order is also important due to the precedence used. The configuration options available for each CLU command can be found in the Commands section.

Precedence of configuration methods

Order of precedence	Configuration method
1.	Command flag
2.	Environment variables
3.	Configuration file

Configuration file types

The CLU supports configuration files written in JSON or YAML.

⚠ **CAUTION**
Due to security concerns, we strongly advise that you do not store the auth token in the configuration file. If you do so, please ensure that the machine the configuration file is stored on is secure and the file has the correct permissions set.

Configuration file example: JSON

```
{
"core_api": "http://tm:@vault:31020",
"nocolor": true,
"output": "json",
}
```

Configuration file example: YAML

```
core_api: "http://tm:@vault:31020"
nocolor: true
output: "json"
```

Exit codes

Exit status (screen display)	**Description**	**Exit code**
SUCCESS	The executed action finished successfully. All resources were imported into Vault.	0
FAILURE	The executed action failed. No resources were imported into Vault.	1
PARTIAL SUCCESS	The executed action partially succeeded. Some of the resources were not imported successfully.	2

Self-signed certificates

CLU can work with self-signed certificates. If you would like to use the self-signed certificates, you need to install your Certificate Authority (CA) certificates in the correct place. CLU will be able to pick these self-signed certificates automatically.

Installing certificates on Linux

The way you install self-signed certificates depends on the platform you are using. For example, all Debian-based distributions have all system-wide certificates located in the `/etc/ssl/certificates` directory. The installation may require administrator rights - please consult your IT department.

Installing certificates on MacOS

Installing self-signed certificates requires access to the Keychain Access app on your Macbook, and the administrator rights to modify the keychain certificates. It should be sufficient to to just drag and drop them into the Keychain Access app. Consult your IT department for more details.

CLU features

Import command

Command

```
./clu import [path to the manifest file (required)] [option(s)]
```

Supported options for CLU import

Command flag	Environment variable	Document History	Description
`--config {filepath}`	N/A	N/A	Set the path to the file containing the CLU config.
`--auth-token {token}`	CLU_AUTH_TOKEN	auth_token	The auth token. Required.
`--jwt {jwt-token}`	CLU_JWT	jwt	The JWT authentication token. An alternative to the auth token. NOTE: You must use the service account auth token if you want to interact with Workflows API resources. You can still use JWT for all other APIs. You can provide both auth tokens at the same time.
`--activate-on-import`	CLU_ACTIVATE_ON_IMPORT	activate_on_import	Activate resources marked with 'activate' after the import step completes successfully.
`--core-api {API}`	CLU_CORE_API	core_api	Set the address of the Core API. Required to determine the CLU's compatibility with Vault. Required.
`--access-control-api {URL}`	CLU_ACCESS_CONTROL_API	access_control_api	Set the address of the Access Control API.
`--workflows-api {URL}`	CLU_WORKFLOWS_API	workflows_api	Set the address of the Workflows API.

Resources supported for import

Access Control API

Resource Name in Documentation Hub	Configuration Type	Supported actions	Supported from CLU version
Data Permission	DATA_PERMISSION	Create	1.0.0
Role Data Permission Assoc	ROLE_DATA_PERMISSION_ASSOC	Create	1.0.0
Role Vault Permission Assoc	ROLE_VAULT_PERMISSION_ASSOC	Create	1.0.0
Role	ROLE	Create	1.0.0

Core API

Resource Name in Documentation Hub	Configuration Type	Supported actions	Supported from CLU version
Account Schedule Tag	ACCOUNT_SCHEDULE_TAG	Create	1.3.0
Calendar	CALENDAR	Create	1.1.0
Calendar Event (see 1 below)	CALENDAR_EVENT	Create	1.1.0
Flag Definition	FLAG_DEFINITION	Create	1.1.0
Global Parameter	GLOBAL_PARAMETER	Create	1.3.0
Global Parameter Value	GLOBAL_PARAMETER_VALUE	Create	1.3.0
Internal Account	INTERNAL_ACCOUNT	Create	1.3.0
Payment Device	PAYMENT_DEVICE	Create	1.3.0
Payment Device Link	PAYMENT_DEVICE_LINK	Create	1.3.0
Plan Migration (see 2 below)	PLAN_MIGRATION	Create	1.2.0
Postings API Client	POSTINGS_API_CLIENT	Create	1.3.0
Product Version (see 3 and 4 below)	SMART_CONTRACT_VERSION	Create	1.0.0
Restriction Set Definition Version (see 3 below)	RESTRICTION_SET_DEFINITION_VERSION	Create	1.3.0

Resource Name in Documentation Hub	Configuration Type	Supported actions	Supported from CLU version
Schedule Tag	SCHEDULE_TAG	Create	1.0.0
Smart Contract Module Versions Link	SMART_CONTRACT_MODULE_VERSIONS_LINK	Create	1.6.0
Supervisor Contract	SUPERVISOR_CONTRACT	Create	1.0.0
Supervisor Contract Version	SUPERVISOR_CONTRACT_VERSION	Create	1.0.0

NOTE
Table footnotes for Core API:

1. To create the Calendar Event resource successfully, specify all required attributes in the resource payload. If the CLU returns a client error "Unable to connect to the API", then one or more required attributes is missing from the payload.
2. Plan Migration Resource support is deprecated as of CLU version 1.2.0.
3. To create and update Restriction Set Definitions, you must use this resource.
4. The Core API Product Version resource is referred to as a Smart Contract Version resource in this document.
5. The migration strategy specified within the Smart Contract Version resource is context sensitive. If the resource cannot be created with the specified migration strategy, the value will be reset to PRODUCT_VERSION_MIGRATION_STRATEGY_NEW_PRODUCT and the creation will be retried.

Workflows API

Resource Name in Documentation Hub	Configuration Type	Supported actions	Supported from CLU version
Policy	POLICY	Create Update	1.0.0
Workflow Definition Version	WORKFLOW_DEFINITION_VERSION	Create	1.3.0

Activation of resources on import

CLU supports the activation of versioned resources in Vault. When running the CLU import command with the `activate_on_import` flag set, CLU will execute the activation step after successfully importing the manifest into Vault. To set a resource version as active, set the `activate` flag of the resource definition in the manifest to `true`. You can only set one corresponding version per resource as active.

Supported resources for activation

Resource	Configuration Type	Supported from CLU version
Workflow Definition Version	WORKFLOW_DEFINITION_VERSION	1.4.0

NOTE

- When creating a Workflow Definition Version (WDV), if the Workflow Definition (WD) already exists, the created WDV will be set as the default version of that WD. If the WD does not exist, a new WD will be created, with the newly-imported WDV set as the default version.
- The CLU does not currently validate whether multiple WDVs have been marked as activatable for a single WD. If multiple WDVs are marked as activatable for a single WD, the WDV that is actually activated is non-deterministic.

Example of a resource marked for activation

```
type: WORKFLOW_DEFINITION_VERSION
id: worklow_resource_id
activate: True
payload: |
    workflow_definition_version: 2.5
    workflow_definition_id: workflow1
    Workflow specification: '@{worklow_definition_spec.yaml}'
```

Validate command

Description

- Reads all resource files from the *manifest.yaml* file's directory and subdirectories.
- Validates only resource files with IDs set in *manifest.yaml* against their corresponding Vault API resources.
- Validates that the fields in the resource definition exist in the API resource; does not validate whether all required API fields have been specified.
- Validates that dependency references are defined against existing configuration resources and existing fields in those resources.

Command

```
./clu validate [path to the manifest file (required)] [global option(s)]
```

Supported options for CLU validate

The validate command only supports global command flags. For more information see Supported global options.

Request idempotency

CLU is *idempotent*; if you import exactly the same resource twice, the second operation will have no effect, and the output will be the same as for the first request. Internally, CLU deterministically generates a request ID from the payloads of your resources and uses these in the Vault API requests. The Vault API endpoint should then return the exact same response each time the same request ID is provided. This allows you to add new resources to your existing CLU configuration packs and run them without having to delete the old resources.

However, there is a caveat to idempotency in CLU. Other tools are likely to use different request IDs, so if you import a resource with another tool (for example, hitting the APIs directly) and then add the same resource to your CLU configuration pack, you may see conflict errors. In this case, Vault returns a 409 error code on conflict. You can define the behaviour of CLU when encountering resource conflicts by setting the desired `on_conflict` parameter.

Request retry mechanism

CLU has a built-in retry mechanism which tries to prevent any network or load related timeouts. This mechanism works by waiting for a second and applying an exponential waiting period between every failed try. It waits at most 30 seconds for a retry to succeed. Currently, this mechanism is hardwired and can not be configured.

Output formats

Ordering of the output

The ordering of the output represents the order in which the resources are imported into Vault. However, the order in which the resources are imported into Vault is non-deterministic and thus it can change in between the runs. This behaviour does not affect the correctness of the import of resources into Vault.

Supported formats

You can configure the output format of the CLU using the output configuration variable. Supported formats are:

- *Text*: This is the default and provides an easy-to-read but unstructured output
- *JSON*: Must be explicitly specified, formats the output using a structured, machine-readable, JSON format and is the preferred choice for integration with CI/CD pipelines

JSON output format

When using JSON output format, errors that have caused CLU to retry an action (for example, intermittent network errors while using the Vault APIs, or waiting for plan migrations to complete) will be logged to the standard error stream. The standard output stream will contain a single JSON object with fields as defined in JSON output format fields.

JSON output fields

Field	**Description**

`notes`
`array [string]`
	An array of notes included for debugging purposes. Omitted if no notes are produced.

`validate`
`map[string: object]`
	A map of resource IDs to the validation result objects. Omitted if no validation results are produced.

`validate[KEY].`
`resource_type`
`string`
	The type of the validated resource.

`validate[KEY].`
`valid`
`boolean`
	The result of the validation attempt.

`validate[KEY].`
`error`
`string`
	The failure reason for the validation attempt. Omitted for successful validation attempts.

`import`
`Map [string: object]`
	A map of resource ID to the import result objects. Omitted if validation fails or when running the validate command.

Field	Description

`import[KEY].`
`resource_type`
`string`
	The type of the imported resource.

`import[KEY].`
`imported`
`boolean`
	The result of the import attempt.

`import[KEY].`
`response`
`object`
	An object which contains the API response for the import attempt of the imported resource.

`import[KEY].`
`response.`
`body`
`object`
	The body of the API response for the import attempt. The format of the body is resource-dependent and is documented in the API documentation associated with each resource. Omitted for unsuccessful import attempts.

`import[KEY].`
`response.`
`action`
`string`
	The action of the import attempt, the action used is determined by the configuration specified in the resource(s).yaml file. Possible values are:
	• "create": Indicates that the CLU attempted to create the resource
	• "update": Indicates that the CLU attempted to update the resource

`import[KEY].`
`response.`
`error`
`string`
	The failure reason of the import attempt. Omitted for successful import attempts.

`status` `string`	The aggregate status of the command. Possible values: • "SUCCESS": The import and validate commands can both have this value. Indicates that all attempts were successful. • "FAILURE": The import and validate commands can both have this value. Indicates that at least one validation attempt was unsuccessful or that all import attempts were unsuccessful. • "PARTIAL SUCCESS": Only the import command can have this value. Indicates that at least one import attempt was unsuccessful and at least one import attempt was successful.
`error` `string`	An error message. Omitted if no error occurs. NOTE: If an error is present, the status field is always "FAILURE".

Output format examples: JSON

Validate command

```
{
"notes": ["Config path successfully read config.json"],
 "validate": {
   "test_flag_defition_resource_id": {
     "resource_type": "FLAG_DEFINITION",
     "valid": true,
    },
   "test_contract_resource_id": {
     "resource_type": "SMART_CONTRACT_VERSION",
     "valid": false,
     "error": "This resource is invalid for example"
   },
   ...
  },
  "status": "FAILURE" # FAILURE/SUCCESS
 }
```

Import command

```
{
  "notes": ["Config path successfully read config.json"],
  "validate": {
    "test_smart_contract_resource_id": {
      "resource_type": "SMART_CONTRACT_VERSION",
      "valid": true,
    },
    "test_flag_definition_resource_id": {
      "resource_type": "FLAG_DEFINITION",
      "valid": true,
    },
    ...
  },
  "import": {
    "account_schedule_tag_resource_id": {
      "resource_type": "ACCOUNT_SCHEDULE_TAG",
      "imported": true,
      "response": {
        "body": {
          "description": "Test Account Schedule Tag 1 Description",
          "id": "test_account_schedule_tag_1",
          "schedule_status_override":
 "ACCOUNT_SCHEDULE_TAG_SCHEDULE_STATUS_OVERRIDE_TO_ENABLED",
          "schedule_status_override_end_timestamp": "2021-01-01T00:00:00Z",
          "schedule_status_override_start_timestamp": "2020-12-31T00:00:00Z",
          "sends_scheduled_operation_reports": true,
          "test_pause_at_timestamp": null
        },
        "action": "create"
      }
    },
    "smart_contract_version_resource_id": {
      "resource_type": "SMART_CONTRACT_VERSION",
      "imported": false,
      "response": {
        "action": "create",
        "error": "received error response code 404: A Product Version ID could not be found"
      }
    }, }
    },
    ...
  },
  "status": "PARTIAL SUCCESS"   # FAILURE/SUCCESS/PARTIAL SUCCESS
}
```

Creating a config pack

A configuration pack represents one self-contained logical unit of a configuration layer. It consists of a manifest file and a collection of resource files.

Input files

File	Description	Fields
manifest.yaml	• Input file used as an argument to CLU. • Contains a whitelist of configuration pack resource IDs that are to be imported. • The location of manifest.yaml defines the root directory of the configuration pack; CLU scans this directory and any of its subdirectories for resource files.	Manifest.yaml fields
.resource.yaml .resources.yaml	• One or more YAML files read by CLU, where each YAML file defines one or more Vault configuration resource definitions. • Resources may be placed in the same directory as the manifest.yaml or in any subdirectory of the manifest file. • Resource files can be given any name but must have the suffix .resource.yaml (for a single resource definition) or .resources.yaml (for multiple resource definitions).	Resource(s).yaml fields
External	• Any external code or markup files, such as Smart Contract Version Python files, that can be referenced in resource definitions. • Useful when a resource's field accepts a multiline input, such as a Smart Contract Version code field. • External files must be placed in the same directory or subdirectory as the .resource.yaml or .resources.yaml file that references it.	External files

Manifest file

A manifest file represents an entry point to a configuration pack. It lists all configuration resources associated with the configuration pack. All resource IDs within a configuration pack must be unique and all resources must be located in the same folder, or in a subfolder.

Manifest.yaml fields

Field	Description
`pack_version`	The semantic version of the configuration pack. Required.
`pack_name`	The name of the configuration pack. Required.
`resource_ids`	A list of configuration resource IDs to import. Required.

Example manifest.yaml file

```
---pack_version: 1.0.0
pack_name: Configuration Pack Name
resource_ids:
 - current_account
 - supervisor_contract_resource_id
 - supervisor_contract_version_resource_id
```

Resource files

Resource files outline the specific details of resources. All resources follow the specification outlined in the Documentation Hub. The fields of the Resource files are described in Resource(s).yaml fields.

Resource(s).yaml fields

Field	Description
`type`	The type of the configuration resource. Required.
`id`	The configuration ID of the resource. Required if vault_id is not set.
`vault_id`	The ID of the resource in Vault. Will also be set as the configuration ID if id is not set. Required if on_conflict is set to UPDATE.
`vault_version`	Unused.
`on_conflict`	The action to take if the resource already exists. Possible values are: • IGNORE: ignore this setting and generate an error. • UPDATE: update the resource (if the resource type supports this action). • SKIP: retrieves the duplicate resource from Vault and continues the command without erroring. Optional.
`payload`	The Vault API payload for this resource. This should be a string formatted as YAML, which is converted to JSON and sent to the relevant API endpoint. Please refer to the relevant method in the API Documentation (available on the Documentation Hub) for further reference. Required.
`resources`	A list of configuration resource definitions. Used in resources.yaml files only. Required.

Example .resource.yaml file

```
---
type: SMART_CONTRACT_VERSION
id: smart_contract
payload: |
   product_version:
       display_name: Current Account
       product_id: smart_contract_id
       code: '@{code_files/account.py}'
   migration_strategy: PRODUCT_VERSION_MIGRATION_STRATEGY_NEW_PRODUCT
```

NOTE
The '@{}' used in the code field is a special CLU notation that can be used to reference external files.

Example .resources.yaml file

```
---
resources:
 - type: SUPERVISOR_CONTRACT
   id: account_supervisor
   vault_id: account_supervisor_1
   payload: |
       supervisor_contract:
           id: account_supervisor_1
           display_name: account supervisor contract
 - type: SUPERVISOR_CONTRACT_VERSION
   id: account_supervisor_version
   vault_id: account_supervisor_version_1
   payload: |
       supervisor_contract_version:
           id: account_supervisor_version_1
           supervisor_contract_id: '&{account_supervisor}'
           display_name: account supervisor contract version
           description: an account supervisor version
           code: '@{code_files/supervisor.py}'
```

See also Additional resource examples.

External files

The notation used by the CLU to find external file references is a @{} wrapper around the external file path, relative to the location of the resource definition that references it.

Because “@” is a reserved symbol in YAML, if an external file reference is included in any of your resources, third party YAML tools may not recognise the file as valid YAML. From CLU version 1.5 onwards, we recommend wrapping external references in quotes as shown in the example. This will ensure that any third party YAML tools will still recognise the file as valid YAML.

External file example

In the example .resources.yaml file, if the Smart Contract Version is located under /configuration/smart_contracts/, then the external file it references should be in /configuration/smart_contracts/code_files/account.py.

Adding JSON payloads

If you need to add a JSON payload to any resource file, it must be escaped by single quotes. For example:

```
- type: minimum_balance_threshold
  payload: '{"DEFAULT":"0"}'
```

CLU bundles

A CLU bundle is a collection of CLU packs that have been grouped together so they can be acted upon (for example, imported into Vault) simultaneously. They allow for more structured organisation of large numbers of Vault resources.

Bundle format

A bundle is a directory or an archive of a directory (in .zip, .tar or .tar.gz archive format). The structure of the bundle is determined by files within its directory tree; any such file which is either named manifest.yaml or has the extension .manifest.yaml defines a pack within that bundle.

The contents of these packs is determined as usual (its resources must be defined in files with the extension .resource.yaml or .resources.yaml, and must be located within the directory tree of the directory containing the manifest). All resource IDs must be unique across the entire bundle.

References within bundles

Resources within a bundle can only reference resources from within their own pack. There are no restrictions on the external files that they can reference (for the archive format, any such files must be present within the archive).

Using bundles

Directory bundles can be created simply by grouping together several packs in the same directory. They can then be acted upon by any of the CLU commands (validate, import, or convert) by providing the path to the directory to CLU:

```
clu import path/to/my/bundle/
```

For convenience (for example, of distribution), an archive bundle can then be created from this directory:

```
tar czf bundle.tar.gz path/to/my/bundle
```

And once more imported using CLU:

```
clu import bundle.tar.gz
```

Instead of tar files, it is also possible to create and use zip files:

```
zip -r bundle.zip path/to/my/bundle
```

You can import it as follows:

```
clu import bundle.zip
```

These commands perform their actions upon all resources specified in all packs within the bundle.

Additional resource examples

```
- type: SMART_CONTRACT_VERSION
    id: smart_contract_resource_id
    payload: |
      product_version:
          display_name: ProdCat example smart contract
          product_id: smart_contract_resource_id
          code: '@{contracts/example_contract.py}'
          params:
              - name: denomination
                value:  'GBP'
      is_internal: false
  - type: GROUP
    id: uk_group
    payload: |
      group:
        id: uk_products_group
        name: UK Products group
  - type: GROUP_VERSION
    id: uk_group_version
    payload: |
      group_version:
        group_id: '&{uk_group}'
        version:
          major: 1
          minor: 0
          patch: 0
          label: ''
        custom_attributes:
          COUNTRY:
            string_value: UK
          REGION:
            string_value: EMEA
  - type: CONTRACT
    id: uk_example_contract
    payload: |
      contract:
        smart_contract_version_id: '&{smart_contract_resource_id}'
  - type: CONTRACT_VERSION
    id: uk_example_contract_version
    payload: |
      contract_version:
        contract_id: '&{uk_example_contract}'
        version:
          major: 1
          minor: 0
          patch: 0
          label: ''
      group_ids:
        - '&{uk_group}'
  - type: PRODUCT
    id: uk_example_contract_child
    payload: |
      product:
        id: uk_example_contract_child
        name: UK Example Contract Child
  - type: PRODUCT
    id: uk_example_contract_student
    payload: |
      product:
        id: uk_example_contract_student
        name: UK Example Contract Student
  - type: PRODUCT
    id: uk_example_contract_standard
    payload: |
      product:
        id: uk_example_contract_standard
        name: UK Example Contract Standard
  - type: PRODUCT
    id: uk_example_contract_premium
    payload: |
      product:
        id: uk_example_contract_premium
        name: UK Example Contract Premium
  - type: PRODUCT_VERSION
    id: uk_example_contract_child_product_version
    payload: |
      product_version:
        product_id: '&{uk_example_contract_child}'
        contract_id: '&{uk_example_contract}'
        version:
          major: '1'
          minor: '0'
          patch: '0'
          label: ''
        parameters:
          maintenance_fee_monthly: '0'
        custom_attributes:
          ACCOUNT_TIER:
            string_value: CHILD
          MAX_AGE:
            int_value: '18'
```

Creating and importing resources tutorial

About this tutorial

This tutorial will walk you through the process of creating CLU resources and using CLU to import and activate these resources in Vault. By the end of this tutorial you will have:

- Defined and imported a Core API Product Version resource
- Defined and imported a Contract Resource that has a dependency on the Core API Product Version
- Defined, imported and activated a Contract Version Resource

An example of how to import resources using CLU

1. Write a manifest file.
2. Write a Core API Product Version resource.
3. Apply resources using CLU.

Writing a manifest file

About the manifest file

The manifest file allows you to specify the resources you want to control with CLU; it is essentially a named and versioned list of CLU resource IDs. The location of the manifest defines the root of the CLU configuration pack.

For further information about manifest files, see Manifest.yaml fields.

About the configuration pack

The layout of the CLU configuration pack is very flexible. In our examples we keep all resources in the same resources directory but this is not a requirement. Resource files can be written in the same directory as the manifest or in subdirectories (of any depth) beneath the manifest.

External files (e.g. Smart Contract code files and Workflow definitions) should also be contained somewhere in the same directory tree (at the same level or below the manifest file) to ensure that the configuration pack is self-contained and portable.

How to write the manifest file

1. In the root of your project, specify a manifest YAML file.

   There are no restrictions on the name of the manifest file; for example purposes we have named it manifest.yaml.

2. Within the manifest YAML file, specify a:

   - pack_version
   - pack_name
   - resource_ids placeholder for the resource IDs to be added to

   For example:

```
pack_version: 1.0.0
pack_name: overdraft configuration pack
resource_ids:
```

NOTE
The pack_version and pack_name fields are not used internally by CLU; they exist only to facilitate user management of resource packs.

Writing a Core API Product Version resource

NOTE
For more information about resources and file references, see Resource and file reference.

1. Create a resources directory and, within it, a code_files sub-directory.

   There are no restrictions on the name of the resources directory; for example purposes we have named it “resources”.

2. To create the Product Version, write the Contract code. For example, write a simple Contract in resources/code_files named overdraft.py:

   ```
   # resources/code_files/overdraft.py"
   ""Overdraft""

   "display_name = 'Overdraft'
   api = '3.0.0'
   version = '1.0.0'
   summary = 'An Overdraft Account'
   ```

3. Create the Smart Contract resource YAML file in the resources directory. In the file, specify:

   - type: The type of resource to import: SMART_CONTRACT_VERSION
   - id: This field is used by CLU to reference resources. Each resource you import should have a unique id.

     NOTE
     This field is distinct from the ID that is applied to the resource in Vault. If the resource you are creating allows specifying a Vault ID, this can be done using the payload.id field.

   - payload: This field contains a YAML block. The fields of this YAML block should correspond to the payload of the resource’s create request. The fields and values for each create request are documented comprehensively in the API documentation. For example:

     ```
     # resources/overdraft_version.resource.yaml
     ---
     type: SMART_CONTRACT_VERSION
     id: overdraft_scv_resource_id
     payload: |
        product_version:
            display_name: Current Account
            code: '@{code_files/overdraft.py}'
            supported_denominations:
     ```

```
          - GBP
product_id: test_smart_contract_1
   migration_strategy: PRODUCT_VERSION_MIGRATION_STRATEGY_NEW_PRODUCT
```

> **NOTE**
> For more information about external references in the code field, see External files.

4. Save the file at resources/overdraft_version.resource.yaml.

> **NOTE**
> - All resource files must end in .resource.yaml or .resources.yaml.
> - The external file reference's path must be written relative to the resource file, not relative to the project root.

5. Update the manifest file to include the CLU resource ID for the overdraft Smart Contract version:

```
# manifest.yaml

---

pack_version: 1.0.0
pack_name: overdraft configuration pack
resource_ids:
- overdraft_scv_resource_id
```

6. Validate that all of the resources in the manifest have been defined correctly using the clu validate command:

```
./clu validate manifest.yaml
```

> **NOTE**
> For this example (and all of the following examples), we assume that both the CLU binary and the manifest.yaml file are in the current directory (the project root). However, you can provide any path to the CLU binary and to the manifest file.

The validation should pass successfully and the output should look like the following:

```
SMART_CONTRACT_VERSION with ID overdraft_scv_resource_id is VALID
SUCCESS All resources were validated successfully.
```

> **NOTE**
> If any errors occur, the output should provide descriptive error messages to help you resolve the issues. To explore the various commands, options, and arguments for the commands you can use:
>
> - ./clu --help
> - ./clu validate --help

7. Continue defining the rest of your resource pack. You can import all of the resources into a Vault instance together once all of them have been defined.

Applying resources using CLU

For further information about commands and CLU configuration, see:

- Commands
- CLU configuration

Option	Action
1.	Import resources.
2.	Import and activate resources using the config file.

Importing resources

1. Import the resources using the following command:

```
./clu import manifest.yaml \
--core-api=https://core.api.url \
--auth-token=<TOKEN>
```

2. Substitute your own values for:

```
https//:core.api.url
<TOKEN>
```

3. The import attempt should be successful and the output should look like the following (although your Vault IDs will vary as they are generated by Vault):

```
SMART_CONTRACT_VERSION with ID overdraft_scv_resource_id is VALID
SUCCESS All resources were validated successfully
SMART_CONTRACT_VERSION with ID overdraft_scv_resource_id was IMPORTED
 successfully using a create action. ID in Vault: "26237"
SUCCESS All resources were imported successfully
```

NOTE
All of the resources are validated again because this is a prerequisite of the import command.

The resource has now been imported into Vault.

Importing and activating resources using the config file

NOTE
The flags provided to the import command in Importing resources were quite long and not particularly easy to manage. CLU provides other more convenient ways to load in the configuration. The following example shows how to write a YAML config file to store the URLs and the activate-on-import configuration variable.

1. Create a YAML config file at the root of the project with the two API URLs and the activate_on_import field; for example:

```
# clu_config.yaml
---
```

```
core_api: https://core.api.url
activate_on_import: true
```

NOTE
When specified in config files, the fields use underscores (_) rather than hyphens (-). For further information about the names for the configuration variables and their relationships see Supported options for CLU Import.

CAUTION
Due to security concerns, we strongly advise that you do not store the auth token in the configuration file. If you do so, please ensure the machine the configuration file is stored on is secure and the file has the correct permissions set.

2. Import and activate your resources using the following command:

```
./clu import manifest.yaml --config=clu_config.yaml --auth-token=<TOKEN>
```

3. The import and activation attempt should be successful and the output should look similar to the following; indicating that resources have been validated and imported and now also indicating that the contract has been activated successfully:

```
SMART_CONTRACT_VERSION with ID overdraft_scv_resource_id is VALID
CONTRACT with ID contract_resource_id is VALID
CONTRACT_VERSION with ID contract_version_resource_id is VALID
SUCCESS All resources were validated successfully
SMART_CONTRACT_VERSION with ID overdraft_scv_resource_id was IMPORTED
 successfully using a create action. ID in Vault: "26237"
CONTRACT with ID contract_resource_id was IMPORTED successfully using a create
 action. ID in Vault: "074623c8-e01f-4cd1-915e-e43ed6105a11"
CONTRACT_VERSION with ID contract_version_resource_id was IMPORTED successfully
 using a create action. ID in Vault: "fce230fa-a3c2-4229-9e28-09bf360bf271"
SUCCESS All resources were imported successfully
CONTRACT_VERSION with ID contract_version_resource_id was ACTIVATED for CONTRACT
 with Vault ID "074623c8-e01f-4cd1-915e-e43ed6105a11" .
SUCCESS All resources were activated successfully
```

NOTE
The Vault IDs of the resources are identical for both import commands and no duplicate resources are created. This is due to idempotency in CLU as explained in Request idempotency.

How to apply the Configuration Layer Release bundle

Every Vault release includes a tar file containing CLU packs required for Vault to function as intended. To extract the tar file, run the command `tar -xzf [tarname].tar.gz` which outputs two folders (`admin_app_core_workflows_outs`, `audit_policies_out`). Each of these folders contains a manifest.yaml and some resource files. Next, run CLU on each of these manifest.yaml files separately.

Document history

Version	Date	Author/reviewer	Comment
0.1	29/07/2020	A Yossifoff	First version.
0.2	31/07/2020	A Yossifoff	Address first comments.
0.2	19/08/2020	A O'Neill	Editorial review.
0.3	20/08/2020	A O'Neill	Review comments discussed with A Yossifoff and document updated.
0.3	25/08/2020	R Sirisomphone	Review comments added.
0.4	27/08/2020	A O'Neill	Discussed with A Yossifoff and updated to incorporate review comments.
1.0	01/09/2020	A O'Neill	Updated to reflect approval.
1.1	05/10/2020	P Bocan	Added Group, Group Version, Contract, Contract Version, Product, Product Version, Calendar, Calendar Event and Flag Definition resources (CLU version 1.1.0).
1.2	22/10/2020	J Measures	Discussed with A Yossifoff and P Bocan and updated to incorporate review comments.
2.0	23/10/2020	J Measures	Updated to reflect approval.
2.1	16/11/2020	C Barnard	Added Plan Migration resource (CLU version 1.2.0).
2.2	25/11/2020	J Davey	Minor wording/formatting changes, Resources table reordered to be alphabetical.
2.3	01/12/2020	P Bocan	Added a paragraph about inserting JSON payloads into a resource file.
3.0	02/12/2020	J Davey	Updated to reflect approval.
3.1	11/01/2021	P Bocan	Added Account Schedule Tag, Global Parameter, Global Parameter Value, Internal Account, Payment Device, Payment Device Link, Postings API Client, Restriction Set Definition Version and Workflow Definition Version resources. Added support for exit status when importing resources (CLU version 1.3.0).
3.2	11/01/2021	J Davey	Discussed with P Bocan and updated to incorporate review comments.
3.3	14/01/2021	R Marincu	Added resource examples for Smart Contract Version, Group, Group Version, Contract, Product, Product Version.
3.4	15/01/2021	J Davey	Simplified Running CLU, and combined Exit status step.

3.5	15/01/2021	R Marincu	Moved examples to a new section.
3.6	18/01/2021	C Peer	Reviewed and approved.
4.0	18/01/2021	J Davey	Updated to reflect approval.
4.1	19/01/2021	J Scott	Added configuration file, environment variable and command flag-related documentation and examples.
4.2	26/01/2021	C Hart	Added documentation for output format configuration under Output Format subheading under Commands.
4.3	08/02/2021	P Bocan	Added section for the activation for individual resources.
4.4	09/02/2021	C Hart	Updated output format documentation to include “notes” field.
4.5	11/02/2021	L Hurt	Terminology aligned and template updated.
4.6	12/02/2021	C Peer	Added section on Migration from old configuration tool.
4.7	25/02/2021	A O’Neill	Minor editorial changes made and comments added.
4.8	25/02/2021	C Peer	Final comments reviewed and resolved, approval given.
5.0	25/02/2021	A O’Neill	Versioning updated to reflect approval.
5.1	17/03/2021	C Hart	• Added “Creating and importing resources tutorial.” • Modified Request Idempotency section to resolve inaccuracies • Added recommendation to wrap external file references in quotes in External Files section
5.2	22/03/2021	P Bocan	Added “Migrating from the old configuration tool“.
5.3	26/03/2021	J Measures	Checked and edited the addition of: • Migrating from the old configuration tool • Creating and importing resources tutorial • Information about CLU convert command
5.4	30/03/2021	A O’Neill	Writing and editorial review. Comments added.
5.5	01/04/2021	C Peer	Reviewed and approved.
6.0	01/04/2021	J Measures	Versioning updated to reflect approval.
6.1	22/04/2021	R Marincu	Added Contract Modules to supported resources.
6.2	14/05/2021	C Peer	Approved.

7.0	14/05/2021	J Measures	Versioning updated to reflect approval.
7.1	04/06/2021	C Hart	Updated human-readable output examples in tutorials to reflect new format for human-readable output.
7.2	13/08/2021	R Marincu, P Bocan, J Scott, C Peer, J Davey	Added `on_conflict``SKIP, updated behaviour on selecting Smart Contract migration strategy, added Appendix D. General formatting and editorial changes.
7.3	16/08/2021	P Bocan, C Peer	Minor changes.
8.0	16/08/2021	J Davey	Versioning updated to reflect approval.
8.1	02/09/2021	J Davey	Changes to add product_id in Example equivalent internal account resources; increment versioning of CLU to 1.7.1.
8.2	23/09/2021	J Davey	Minor editorial changes.
9.0	27/09/2021	J Davey	Versioning updated to reflect approval.
9.1	25/10/2021	J Davey	Major usability reorganisation in conjunction with C Peer and P Bocan.
9.2	04/11/2021	P Bocan	Approved.
9.3	08/11/2021	J Davey	Added omitted Smart Contract Module Versions Link
10.0	04/11/2021	J Davey	Versioning updated to reflect approval.
10.1	13/12/2021	P Bocan	Added wording for request retry mechanism, ordering of output, and miscellaneous small changes.
10.2	16/12/2021	P Bocan	Approved.
11.0	16/12/2021	J Davey	Versioning updated to reflect approval.
11.1	25/01/2022	J Davey	Vault version changed to 3.2.
12.0	11/02/2022	J Davey	Versioning updated to reflect approval.
12.1	09/03/2022	P Bocan	Changes to Vault Version Compatibility.
13.0	10/03/2022	J Davey	Versioning updated to reflect approval.
13.1	01/05/2022	P Bocan	Revisions for Vault 4.0 throughout.
13.2	12/05/2022	J Davey	Minor editorial changes.
14.0	13/05/2022	J Davey	Versioning updated to reflect approval.
14.1	14/06/2022	J Davey	Changed wording in Vault Version compatibility to accommodate new Vault version.
15.0	17/06/2022	J Davey	Versioning updated to reflect approval.

16.0	18/07/2022	J Davey	Versioning updated to reflect approval.
16.1	09/08/2022	P Bocan	Removed sections *Convert Command*, *Writing a Product API Contract Resource*, *Writing a Product API Contract Version Resource*. Added Self-signed certificate details.
16.2	12/08/2022	L Dix / J Davey	Changes to "Additional resource examples" to remove product hub file references.
17.0	17/08/2022	J Davey	Versioning updated to reflect approval.
18.0	13/09/2022	J Davey	Versioning updated to reflect approval.
19.0	26/09/2022	J Davey	Versioning updated to reflect approval.
20.0	08/12/2022	J Davey	Versioning updated to reflect approval.
20.1	07/02/2023	P Bocan	Added support for JWT tokens.
20.2	07/02/2023	J Davey	Updated version of CLU and Vault version compatibility.
20.3	13/03/2023	P Bocan	Approved
21.0	13/03/2023	J Davey	Versioning updated to reflect approval.

Thought
Machine