Catena-X
Your Automotive Network

Catena-X Test Suite · Development · 29 Jul 2026

Writing a TCK *syntax guide.*

How to declare a Test Compatibility Kit for the Catena-X Test Suite: package layout, `index.yaml`, test files, steps and validations.

Guide

What you will be able to write

01	The package — what a TCK is made of	Folders, files, compilation input
02	The manifest — `index.yaml`, block by block	Metadata, dataspace, env, tests
03	The test file — phases, steps, validations	GitHub Actions-style building blocks
04	Build it — worked example, field reference, checklist	From CAC to passing test

Part one

The package

Anatomy

An uncompiled TCK is a folder with four things in it

You author human-readable YAML. The engine checks the syntax, translates it to a machine-readable form and compresses it into a `.tck` package for execution.

`index.yaml`

The TCK manifest, and the single input to compilation. Metadata, dataspace and infrastructure requirements, environment config, and the ordered list of tests.

`/tests`

One YAML file per test, imported by id from the manifest. Each file declares its own setup, execution and teardown phases.

`/testdata`

Payloads reusable across tests. Any format — this is why every entry declares an explicit `type`.

`/schemas`

JSON Schemas used to validate requests and responses — for example a CX data model at a pinned version.

— Anatomy, assembled

What the folder actually looks like

The manifest names every file below it. Nothing is discovered by scanning the directory — if it is not declared in `index.yaml`, it does not exist.

```
cx-0135-ccm-tck/
├─ index.yaml   manifest · the only compilation input
├─ tests/
│  ├─ ccm-request-happy-path.yaml
│  └─ ccm-request-negative.yaml
├─ testdata/
│  ├─ ccm-request.json
│  └─ bpn-list.csv
└─ schemas/
   └─ cx-ccm-v0.3.0.json
```

1 · Declare

Write the manifest, then one YAML file per test. Test data and schemas are referenced by id, never by path from inside a test.

2 · Check

The engine runs an integrity and syntax check across the whole folder before anything is packaged.

3 · Compile

Translation to the machine-readable format, then compression into a single package — optionally encrypted.

→ `cx-0135-ccm-tck.tck`

Lifecycle

From YAML you write to a report a CAB can read

Your syntax is checked once, compiled once, and then executed many times. Everything downstream of compilation is machine-readable — the YAML never reaches the live environment.

01
Declare
Manifest, tests, test data and schemas, written by the expert group.

02
Check
Integrity and syntax check across the whole folder. Bad references fail here.

03
Compile
Translation from human-readable YAML to the machine-readable format.

04
Package
Compressed into one `.tck` file, optionally encrypted.

05
Execute
Run by the service provider against a live Catena-X environment.

06
Report
Execution logs and a conformity report the assessment body can verify.

Manifest map

Five blocks make up `index.yaml`

01

Header

Declares what the file is and which syntax version parses it.

`kind · syntax · id`

02

Metadata

Name, version, authors, license and the standards covered.

`metadata:`

03

Dataspace

Which ecosystem and which dataspace version is under test.

`dataspace:`

04

Infrastructure

What the engine and the system under test must provide.

`infrastructure:`

05

Env + tests

Runtime variables, schemas, test data — then the ordered test list.

`env: · tests:`

index.yaml · blocks 1–2

Header and metadata

```yaml
kind: tck
syntax: v1-alpha
id: <id from tck>

metadata:
  name: <name of the TCK / label for the user>
  version: <for version control>
  description: <detailed text if needed>
  authors:
    - name: Certificate Management Expert Group
      email: ccm-eg@catena-x.net
      company: Catena-X Automotive Network e.V.
  copyright_holders:
    - <list of strings: [YEAR] [COMPANY]>
  license: LicenseRef-Proprietary
  standards:
    - id: CX-0135
      version: <the version>
  tags:
    - <list of strings, used for search>
```

`kind`

Fixed. Present in every declaration file so the parser knows what it is reading.

`syntax`

The engine library syntax version — it pins which technical capabilities are available. It should not vary often.

`id`

Descriptive, human-legible. Every test file references it as its `namespace`.

`authors`

Not decoration. This is the contact person or group when a TCK misbehaves in the field.

Dataspace and infrastructure requirements

```yaml
dataspace:
  ecosystem: Catena-X
  version: <dataspace version>

infrastructure:
  engine:
    connector:
      required: true
      standard:
        - id: CX-0018
          version: 4.0.2
  sut:
    connector: …
    dtr:
      required: true
      standard:
        - id: CX-0002
          version: 1.0.2
```

`ecosystem`

Reserved so the same testing framework can serve other dataspaces later.

`version`

Mandatory. It attaches the conformity result to a specific dataspace version.

`engine`

What the backend must stand up for you — this is what configures its EDC clients at boot.

`sut`

The system under test. Connector and DTR can each be declared; DTR may be optional.

index.yaml · block 5a

Environmental configuration — variables, schemas, test data

```yaml
env:
  variables:
    - id: <variable key>
      uses: <definition>
      with:
        source: [ input | generated | value ]
      returns:
        value:
          type: [ string | object | number | array | bool ]
          class: <optional>

  schemas:
    - id: <id from schema>
      source: <file the content lives in>

  testdata:
    - id: <id from test data>
      source: <file the content lives in>
      type: <file type>
```

`variables`

Configured at runtime. `uses` names the variable definition; each definition has its own fixed key names under `with`. More types exist and the set can be extended.

`returns`

Same shape as a step's returns — it tells the person configuring the TCK what comes out.

`testdata.type`

Explicit because test data is not required to be JSON.

Everything declared here gets an `id` — that id is the handle tests use at runtime.

index.yaml · block 5b

Tests, in execution order

```yaml
tests:
  - id: <name of the test file>
    name: <short name from test>
  - id: …
    name: …
```

An ordered array. `id` resolves to the matching YAML file in `/tests`.

Test 1 → Test 2 → Test 3 → Test 4 → Test n

Tests run sequentially, in cascade, in the order you list them.

Each test must be independent of the others. If one fails, the rest can still succeed.

Part two

The test file

Phases

Three phases, inspired by JUnit and PyTest

A step is a building block. Blocks are not interchangeable across phases — a step can be used in one or more phases, and each phase accepts the steps declared for it.

`setup:`
Pre-steps · no validate

Pre-step 1 → Pre-step 2 → Pre-step n

`execution:`
Steps + validations

Test step 1
validations 1…n
→
Test step 2
validations 1…n
→
Test step n
validations 1…n

Continues to the next step only if every validation passed. One failure aborts the test.

`teardown:`
After-steps · no validate

After-step 1 → After-step 2 (clean-up) → After-step n

Test file · header

Header and metadata of a test

```yaml
kind: test
syntax: v1-alpha
id: <id from test>
namespace: <id from tck>

metadata:
  name: <name of the test / label for the user>
  version: <for version control>
  description: <detailed text if needed>
```

`kind: test`

The only structural difference in the header — everything else mirrors the TCK declaration.

`namespace`

Must be exactly the `id` of the TCK manifest that imports this test.

Fewer fields than the manifest on purpose: authors, license, standards and dataspace all live in `index.yaml`.

The building block

Step syntax — modelled on GitHub Actions

```
# setup: same syntax, without "validate"
execution:
  - id: <unique-key-in-test-for-step>
    uses: <function-key mapped to a @step function>
    name: <what is being executed / tested>
    with:
      input-key-1: <input config data from standard>
      input-key-2: <can be taken from a variable>
      input-key-n: …
    returns:
      return-key-1:
        type: [ string | object | number | bool | array ]
        class: <semantic type, optional>
      return-key-n: …
    validate:
      - uses: <validation-function-key>
        with:
          input: <a returned variable to validate>

# teardown: same syntax, without "validate"
```

`execution`

An array of steps, executed in sequential order.

`uses`

The function key. It maps to a backend function carrying the `@step` annotation.

`with`

The input parameters of that function. Which keys are allowed depends on the function you selected.

`returns`

Declares the outputs the step will generate, so whoever configures the step knows what is available downstream.

`validate`

Same shape as a step, but returns nothing. Its inputs are the step's return variables.

Naming

How to read a function key

Root capability, then an optional module path, then the action. Modules may nest into sub-modules; simple capabilities skip them entirely.

`connector` / `provider` / `create_asset`

Root capability
Which subsystem

Module
Optional grouping

Function / action
What it does

`connector / provider / create_asset`

Creates an asset in the EDC you specify.

`http / http_request`

Executes an HTTP request against a URL you configure.

`mock / api`

Mocks an API HTTP response for the system under test to call.

From standard to syntax

A CAC becomes an action plus its MUST criteria

Conformity Assessment Criteria are written **WHEN … WITH … MUST …**, optionally guarded by **IF**. The WHEN and WITH become the step; each MUST becomes one validation.

CAC, as written in the standard

WHEN

an HTTP request is sent to `/certificatemanagement/request`

WITH

the payload defined by the standard

MUST

contain key `dct:type` with operator = and value `https://catena-x…`

IF

the result is `failed`, the status must be `503`

The same CAC, in step syntax

```
# WHEN + WITH -> the action
- id: request-certificate
  uses: http/http_request
  with:
    path: /certificatemanagement/request
    body: ${{ testdata.ccm_request }}
  # each MUST -> one validation
  validate:
    - uses: json/must_contain_key
      with:
        key: dct:type
        operator: "="
        value: https://catena-x…
```

Part three

Build one, end to end

Worked example

A complete test file, from setup to teardown

```yaml
kind: test
syntax: v1-alpha
id: ccm-request-happy-path
namespace: cx-0135-ccm-tck

metadata:
  name: Certificate request returns a valid certificate
  version: 1.0.0
  description: Verifies CX-0135 request flow end to end.

setup:
  - id: publish-asset
    uses: connector/provider/create_asset
    name: Publish the CCM asset on the engine EDC
    with:
      asset_id: ${{ env.asset_id }}
      base_url: ${{ env.sut_data_plane }}
    returns:
      asset:
        type: object

teardown:
  - id: remove-asset
    uses: connector/provider/delete_asset
    with:
      asset_id: ${{ env.asset_id }}
```

```yaml
execution:
  - id: request-certificate
    uses: http/http_request
    name: Request a certificate from the SUT
    with:
      method: POST
      path: /certificatemanagement/request
      body: ${{ testdata.ccm_request }}
    returns:
      response:
        type: object
        class: http_response
    validate:
      - uses: http/must_have_status
        with:
          input: ${{ steps.request-certificate.response }}
          status: 200
      - uses: schema/must_validate
        with:
          input: ${{ steps.request-certificate.response }}
          schema: cx-ccm-v0.3.0
```

Illustrative function keys and variable references — check the Tractus-X Test Lab library for the capabilities available in your `syntax` version.

What comes back

Execution logs — Cloud Events, one object per line

The same JSONL structure serves live execution tracking in the UI and later debugging. Your step ids and function keys are what make these events readable — name them well.

```
specversion: <cloud events attribute>
id: <tck-id>/<test-id>/<step-id>/
    <event-type>/<hash-from-data>
source: <function used in the step, or system>
type: <event type>
time: <time the event was emitted>
sequence: <emission order in the log>
data: <structure depends on the event type>
```

Live execution tracking

Streamed from backend to frontend over SSE while the test runs, so the mission-control view updates without a refresh.

Tracing and debug

The stored log keeps the HTTP requests and responses, error traces and recommendations that explain an incompatibility.

Part four

Field reference

Notation and grammar

Notation	Meaning
`<value>`	Placeholder you replace
`[ a | b ]`	Closed set — pick exactly one
`key[]`	Array; order is significant
`a.b`	Nested key `b` under `a`
required	Compilation fails without it
optional	May be omitted entirely

Every declaration file is YAML, UTF-8, two-space indentation. Keys are lower snake case; ids are lower kebab case.

```
# Declaration grammar (EBNF)
declaration   = header , metadata , body ;
header        = "kind:" kind , "syntax:" version ,
                "id:" identifier ;
kind          = "tck" | "test" ;
body          = tck-body | test-body ;

tck-body      = dataspace , infrastructure , env ,
                tests ;
test-body     = namespace , [ setup ] , execution ,
                [ teardown ] ;

setup         = step* ;      (* no validate *)
execution     = step* ;
teardown      = step* ;      (* no validate *)
step          = "id:" identifier ,
                "uses:" function-key ,
                "name:" text , "with:" mapping ,
                [ "returns:" return-map ] ,
                [ "validate:" validation* ] ;
validation    = "uses:" function-key , "with:" mapping ;
function-key  = root , { "/" module } , "/" action ;
reference     = "${" "{" scope "." path "}" "}" ;
scope         = "env" | "testdata" | "schemas" | "steps" ;
```

Header and metadata fields

Key	Presence	Type	Rule
`kind`	required	`enum`	Literal `tck`. Tells the parser how to read the file.
`syntax`	required	`string`	Engine library syntax version, e.g. `v1-alpha`. Pins the available capabilities.
`id`	required	`string`	Descriptive, unique. Every test repeats it as `namespace`.
`metadata.name`	required	`string`	Label shown to the user in the Test Suite.
`metadata.version`	required	`string`	Version of the TCK itself, for version control.
`metadata.description`	optional	`string`	Detailed text where the name is not enough.
`metadata.authors[]`	required	`object[]`	Each entry: `name`, `email`, `company`. The contact for the TCK.
`metadata.copyright_holders[]`	required	`string[]`	Format `[YEAR] [COMPANY]`, one per line.
`metadata.license`	required	`string`	SPDX identifier or a `LicenseRef-` value.
`metadata.standards[]`	required	`object[]`	Each entry: `id` + `version` of a standard this TCK certifies.
`metadata.tags[]`	optional	`string[]`	Free keywords, used for search in the catalogue.

Dataspace and infrastructure fields

Key	Presence	Type	Rule
`dataspace.ecosystem`	required	`string`	`Catena-X` today. Reserved so the framework can serve other dataspaces.
`dataspace.version`	required	`string`	The dataspace version the conformity result is attached to.
`infrastructure.engine`	required	`object`	What the Test Suite must provide. Configures the backend's clients at boot.
`infrastructure.sut`	required	`object`	What the system under test must expose. Same component shape as `engine`.
`<scope>.connector`	required	`object`	The EDC. Declared for the engine, the SUT, or both.
`<scope>.dtr`	optional	`object`	Digital Twin Registry. Omit it when the TCK does not touch twins.
`<component>.required`	required	`bool`	Whether execution may start without the component present.
`<component>.standard[]`	required	`object[]`	`id` + `version`, e.g. CX-0018 / 4.0.2 for the connector, CX-0002 / 1.0.2 for the DTR.

Environment and test-list fields

Key	Presence	Type	Rule
`env.variables[].id`	required	`string`	The key tests use to reach the value at runtime.
`env.variables[].uses`	required	`string`	The variable definition. It fixes which keys are legal under `with`.
`env.variables[].with.source`	required	`enum`	`input` (asked from the user), `generated`, or `value`.
`env.variables[].returns.value.type`	required	`enum`	string · object · number · array · bool
`env.variables[].returns.value.class`	optional	`string`	A semantic type on top of the raw type, for example a BPN.
`env.schemas[].id · .source`	required	`string`	Runtime handle, and the file in `/schemas` holding the JSON Schema.
`env.testdata[].id · .source`	required	`string`	Runtime handle, and the file in `/testdata`.
`env.testdata[].type`	required	`string`	The file type. Explicit, because test data is not required to be JSON.
`tests[].id`	required	`string`	Resolves to the YAML file of that name in `/tests`.
`tests[].name`	required	`string`	Short label. Array order is the execution order.

Test document fields

Key	Presence	Type	Rule
`kind`	required	`enum`	Literal `test`.
`syntax`	required	`string`	Must match the version declared in the manifest.
`id`	required	`string`	Unique within the TCK, and equal to the entry in `tests[]`.
`namespace`	required	`string`	Exactly the `id` of the owning TCK manifest.
`metadata.name · .version`	required	`string`	Label for the user, and the version of this test.
`metadata.description`	optional	`string`	What the test proves, in the words of the standard.
`setup`	optional	`step[]`	Pre-steps. Step syntax without `validate`.
`execution`	required	`step[]`	The tested behaviour. The only phase that may carry validations.
`teardown`	optional	`step[]`	Clean-up. Runs so the next test starts from a clean state.

Step and validation fields

Key	Presence	Type	Rule
`id`	required	`string`	Unique key in the test. It appears verbatim in every event log line.
`uses`	required	`function-key`	Maps to a backend function annotated `@step`. Form: root / module / action.
`name`	required	`string`	What is being executed or tested, in plain language. Shown live during execution.
`with`	per function	`mapping`	Input parameters of the selected function. Values may be literals or references.
`returns`	optional	`mapping`	Declares the outputs. Anything not declared here cannot be referenced downstream.
`returns.<key>.type`	required	`enum`	string · object · number · bool · array
`returns.<key>.class`	optional	`string`	Semantic type of the return, so validations can be offered contextually.
`validate[]`	execution only	`object[]`	One entry per MUST criterion. The step fails on the first failed entry.
`validate[].uses · .with`	required	`string · mapping`	A validation function and its inputs. Same shape as a step, but it returns nothing.

Before you compile

Author's checklist

One `syntax` version everywhere

Manifest and every test file declare the same engine syntax version.

Tests are independent

No test relies on state left behind by another. Provision in setup, clean up in teardown.

Versions are pinned

Dataspace version, connector and DTR standard versions, and every schema id.

No validate outside execution

Setup and teardown steps take the same shape minus `validate`.

Namespaces match

Every test's `namespace` equals the manifest `id`.

Every MUST has a validation

Each CAC criterion maps to a validation function — including the negative and conditional cases.

Ids are descriptive

Step ids appear verbatim in the event log. Vague ids make failures hard to read.

Authors are reachable

A named expert group and a working email address, not an individual who may rotate off.

Catena-X
Your Automotive Network

Write the manifest, then one test at a time.

The capability catalogue lives in the Tractus-X Test Lab library. Syntax questions and new capability requests go to the Certificate Management Expert Group.

eclipse-tractusx/tractusx-testlab ccm-eg@catena-x.net catena-x.net