European Organisation for Astronomical Research in the Southern Hemisphere

Programme: ELT

Project/WP: Core Integration Infrastructure / CCS

ELT CII - User Manual

Document Number: ESO-XXXXXX

Document Version: 1

Document Type: Manual (MAN)

Released on: -

Document Classification: Public

Owner:	Schilling, Marcus
Validated by PM:	
Validated by SE:	
Validated by PE:	
Appoved by PGM:	
	Name

Authors

Name	Affiliation
Sekoranja, Matej, et al.	Cosylab
Schilling, Marcus, et al.	ESO/DOE/CSE

Change Record from previous Version

Affected Section(s)	Changes / Reason / Remarks
All	All sections updated

CONTENTS:

CHAPTER

ONE

INTRODUCTION

The CII is the Common Integration Infrastructure of the ELT Central Control System.

1.1 Scope

This document is the user manual for the ELT CII.

1.2 Overview

List of CII user manuals:

- Middleware Abstraction Layer: The Middleware Abstraction Layer (MAL) is a part of Core Integration Infrastructure (CII) and serves to decouple the CII applications from the communication middleware, permitting transparent functioning over multiple middleware stacks as well easing the substitution of one middleware implementation for another.
  - *MAL API*: The applications do not need to know any specifics of the various middleware software, MAL hides all the specifics via its API. The MAL API is an API only; it does not contain any actual mapping or dependency to any communication middleware product. MAL supports two standardized communication patterns on top of which all applications shall be built. These are: - Data centric publish-subscribe: decoupled peers based on data producers and consumers. - Request-response: coupled, state-full, peer-to-peer communication.
  - *MAL Mappings*: Describes configuration options for the following MAL mappings: DDS, ZPB and OPC UA.
  - *MAL Python Mappings*: This document provides information on use of Python MAL mappings interface also referred to as Python MAL API. Python MAL API builds on MAL C++ API foundation. It provides programming interface that is very similar to the programming interface that is exposed by MAL C++ API.
  - *MAL ICD Generation*: Manual for developing Interface Control Documents, specifically Type Definition Documents holding ICD types.
- *Configuration*: Configuration provides the mechanisms for distributing configuration data to the control system application software and to allow for the collection of configuration data from these systems.
- *Online Database* The Core Integration Infrastructure (CII) Online Database (OLDB) provides distributed data publishing and access to actual or live data for user interface and control applications that do not have low-latency or real-time performance requirements. The term “online” refers to the fact that the database provides current and live values for the monitor points of the control system.
- *Error Handling*: Error Handling in CII provides the APIs and tools necessary to conveniently implement a common error handling mechanism across all EELT Control System applications and thus streamline the diagnosis of abnormal behavior of the system.
- *Logging*: CII Log provides the logging and tracing services of the Control System. Logging pertains to all aspects of CII services. Its purpose is to provide basic and detailed information of the different services and systems in the CII in order to have a better understanding of the behaviour of the distributed system, and in particular to troubleshoot the system in case of problems.
- *Alarm*: CII Alarm system uses Integrated Alarm System (IAS) [4] as its alarm system. IAS is a software whose main purpose is to get alarms and monitor points from different sources and

generate alarms to present to the users who can range from operators in the control room, up to engineers sittings at their desks. The alarm system is a message-passing facility that routes information about abnormal situations detected from hardware or software to the user.

- *Telemetry*: The purpose of the CII Telemetry Service is archiving the periodic, event-based and ad-hoc data that will be used for calculation of statistics, and the long-term trends and performance of the ELT telescope.

CHAPTER

TWO

MAL API

Document ID:	
Revision:	2.11
Last modification:	March 18, 2024
Status:	Released
Repository:	https://gitlab.eso.org/cii/info/cii-docs
File:	mal_api.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
0.1	2017-10-25	msekoranja	All	Created.
1.0	2017-11-06	msekoranja	All	Update on API, release.
1.1	2018-02-23	msekoranja	3.1, 3.3	Update on API changes, C++ MrvSubscrib er example.
2.0	2019-02-22	msekoranja	All	Complete rewrite of a document.
2.1	2019-03-04	msekoranja	All	Example code revised.
2.2	2022-03-31	jpribosek	2.5.2	Example code revised.
2.3	2022-04-21	dkumar	2.6.2	Exception handling details
2.4	2022-12-14	mschilli	2.7 - 2.8	New section on API of ICD entity
2.5	2022-12-27	dkumar	2.7	Eigen example
2.6	2023-01-11	jrepinc	2	Python subscriptio n example update/note
2.7	2023-02-17	dkumar	Entites	Parent child relation on cloning
2.8	2023-02-17	dkumar	Subscriber	Subscriber close guaranty
2.9	2023-03-07	dkumar	Subscriber	Subscriber method call warning
2.10	2023-07-04	nkornwei	All	Remove Java ,Update DDS URIs
2.11	2024-18-03	mschilli	0	Public doc

Confidentiality

This document is classified as Public.

Scope

This document is a manual for the Middleware Abstraction Layer of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

Glossary of Terms

Abb	Meaning
API	Application Programming Interface
MAL	Middleware Abstraction Layer
QoS	Quality of Service
CII	Core Integration Infrastructure
ICD	Interface Control Document
XML	Extensible Markup Language

References

1. Cosylab, Middleware Abstraction Layer Design document, CSL-DOC-17-147260 v1.5
2. Cosylab, ELT CII MAL Transfer document, CSL-DOC-18-168015, version 1.8
3. Cosylab, Data Addressing Specification, CSL-DOC-17-147264, version 1.2

2.1 Overview

This document is a user manual for MAL API. The document provides descriptions of the API and instructions to programmers how to use MAL API. All the details on the API and its functionality are covered by the MAL design document [1].

2.2 Installation

See ELT CII MAL Transfer document [2], Chapter 1.2 on how to build and install MAL.

2.3 Introduction

MAL is an abstraction API that provides complete independence from the communication middleware software. This is achieved by dynamically loadable MAL middleware implementation (mapping) that implements MAL API. In addition, data entities (classes, structures, interfaces) need to be independent from the communication-middleware too. This is achieved by using generation: user defines its data entities in ICD definition language (i.e. XML), the generator then produces middleware-agnostic entities that are used by the applications and middleware-specific code that needs to be generated. The specific code is dynamically loaded by the MAL mapping implementations when needed. This part is completely hidden from the programmer. The programmer must never use or reference any middle-ware specific code it only uses pure MAL API and agnostic data entities.

Middleware selection is done using Uniform Resource Identifier (URI) addressing. A URI is a string of characters that unambiguously identifies a particular resource. The scheme part of the URI (e.g. “dds.ps://” prefix) determines what MAL mapping to use, whereas the rest of the URI is completely middleware specific. For more info on the URIs please refer to the Data Addressing Specification ([3]).

MAL API supports two standardized communication patterns: data centric publish-subscribe (decoupled peers based on data publishers and subscribers) and request-response (coupled peer-to-peer client-server communication).

The following sections provide the description and examples of the API for both communication patterns. The examples demonstrate usage of the API, they do not do anything meaningful. Source code of the examples is available in elt-mal/mal module.

2.4 CiiFactory

CiiFactory is an entry-point class of the MAL API. Usually it is accessed as a singleton instance via getInstance() method. This provided a shared, thread-safe instance to be used among the threads in one process. However, in case of a more complex application where an insulation among different "sub-processes" is required, the API provides a createInstance() method that instantiates a completely new CiiFactory instance on every call of this method. Avoid using this method unless there is a good reason for it.

A CiiFactory instance has 2 groups of methods: methods for MAL mapping implementation (un-)registration and factory methods of entities (publisher, subscriber; client, server) for both communication patterns. The main task of CiiFactory is to delegate factory method calls to the appropriate registered MAL mapping implementation depending on the URI, given as a parameter to all the factory methods. CiiFactory is instantiated without any MAL mapping registered. A programmer needs to register MAL mappings. The registration is done by calling registerMal method that takes 2 parameters: a reference to the instance of MAL mapping implementation and an URI schema string to be handled by the implementation. CiiFactory takes no responsibilities on registered mappings lifecycle; it is responsibility of the programmer to take care of it, still CiiFactory provides a close() method that calls close() on all registered MAL mappings and unregisters them. A programmer has also an ability to unregister previously registered MAL mappings. Un-registration only removes a MAL instance from CiiFactory dictionary; any previously created entities (e.g. publishers) will remain active. The entities lifecycle is responsibility of the MAL mapping implementation, meaning releasing a MAL mapping implementation also releases all its entities.

Registration of a MAL mapping introduces a compile and runtime dependency to the mapping. CiiFactory.loadMal method dynamically loads the mapping and avoids any dependency on any MAL mapping. CiiFactory.installMal calls CiiFactory.loadMal and registers newly loaded mapping to the shared CiiFactory instance. Optional properties parameter can be passed as parameter to configure loaded MAL. Both methods accept MAL mapping (schema) name, e.g.: dds, zpb, opcua.

All factory methods accept 3 common parameters: a URI, a quality-of-service (QoS) parameters and a string name-value list of MAL mapping specific properties. MAL API defines a common set of QoS parameters per communication pattern (described in detail in the following sections). If none (null) are supplied defaults are used.

The list of MAL mapping specific properties allows specific/fine tuning of the middleware in a uniform way. The properties in a list are specific and documented per MAL mapping implementation, usually they are provided via some kind of configuration subsystem. If none (null) are supplied defaults are used.

A registration of another MAL instance with the same schema string will override any previous registration (un-registration no longer needed). Un-registration of non-registered scheme results in no operation action. A call to a factory method with a URI scheme that is not registered will result in SchemeNotSupportedException exception.

The following code examples show recommended initialization of MAL mapping with creation of a publisher.

2.4.1 C++

C++ implementation uses RAII idiom using smart pointers. Resources are closed automatically when their enclosing scope ends. To close resource explicitly call close() method.

```
try {

  // load DDS MAL mapping with (optional) properties
  std::shared_ptr<::elt::mal::Mal> ddsMal = mal::loadMal("dds",␣
↪mal::Mal::Properties{
    {}});
  ::elt::mal::CiiFactory &factory = mal::CiiFactory::getInstance();

  // register DDS MAL mapping with CiiFactory
  factory.registerMal("dds", ddsMal);

  // create URI for dds MAL on domain Id 100, topic "m1/CabinetTelemetry"
  mal::Uri uri("dds.ps://100/m1/CabinetTelemetry");

  // create publisher instance with default QoS and no specific properties
  std::unique_ptr<ps::Publisher<mal::example::Sample>> publisher = factory.
↪getPublisher<mal::example::Sample>(
     uri,
     mal::ps::qos::QoS::DEFAULT,
     {});
  // put publisher code here...

} catch (std::exception& exc) {
  std::cerr << "EXCEPTION: " << exc.what() << std::endl;
}
```

2.4.2 Python

In Python all generated entities are prefixed with a “Mod”<name of the ICD file in camelCase> string. The module also loads appropriate C++ binding classes and libraries. Submodule structure follows ICD package structure with capitalized name for each package.

Python MAL API supports context manager protocol to automatically release the resources after use. Closing resources explicitly with close() is also supported.

```
# Import MAL API.
import elt.pymal as mal

# Import Data Entity class from binding module.
from ModManualExamples.Elt.Mal import Example

# dds mal for domain id 100, topic m1/CabinetTelemetry.
uri = 'dds.ps://100/m1/CabinetTelemetry'
```

(continues on next page)

(continued from previous page)

```
factory = CiiFactory.getInstance()

# Load DDS MAL mapping.
ddsMal = mal.loadMal("dds", {})
factory.registerMal('dds', ddsMal)

# Use context manager interface of the publisher
# to ensure underlying instance of the publisher is
# closed when context is destroyed.
# Create publisher instance with default QoS and no
# specific properties.
with factory.getPublisher(uri,
                          Example.Sample,
                          qos=mal.ps.qos.DEFAULT) as publisher:
    # Put publish code here...

# Publisher is automatically closed at this point.
```

2.5 Publish-subscribe

Publish-subscribe is a data-centric decoupled communication pattern. Publishers (data producers) publish data to a topic, and subscribers (data consumers) subscribe to the topics they are interested in to receive the data published. This is not a coupled (connected) peer-to-peer communication – there can be many subscribers per one topic. MAL prescribes a limit of only one publisher per topic. Order of instantiation/startup also does not matter. Subscriber will miss any data sent to the channel prior their subscription. Subscribers can subscribe to a topic even if there is no publisher available. Once it became available the subscribers will start receiving the newly published data. The paradigm does not guarantee that the first n-messages are will be received by the subscribers, even if the subscribers are already running when the publisher starts. It takes some time to establish communication mechanism on startup of the publishers and subscribers leading to non-instant subscriptions to the topics and consequently missed data. In addition, MAL does not guarantee reliable communication (and queueing) – the data might be lost during the transmission or dropped if send/receive queues are full. MAL mapping specific properties can be used to set reliability QoS, if underlying communication middleware supports it.

MAL however does support 2 QoS settings:

- Deadline

  The deadline QoS shall be the maximum age of a sample, i.e. t(now) – t(sample) < deadline_time, sample time is a time of creation of a sample. The sample time is given as parameter to the publish method call on publisher. If not given, current time is used. Note that this also implies that if a sample is left too long in the subscribers receive queue it might expire (and is removed from the queue).

  If deadline QoS is not defined, it is assumed to be infinite.

- Latency

  The latency QoS shall be the maximum time a sample may remain in-transit between the publisher and subscriber. That is, the arrival time minus sample time cannot be greater than latency QoS. If latency QoS is not defined, it is assumed to be infinite.

If QoS is violated a corresponding sample is discarded.

It is assumed that time among different network nodes where publishers and subscribers run is synchronized.

MAL also supports a concept of instances. Every structure (defined in ICD) can be assigned a key. A key is a set of structure fields (of primitive types) that uniquely identifies one instance. This allows programmers to publish/retrieve/filter data based on their instance.

To create data entities use createDataEntity methods available on both, publisher and subscriber.

2.5.1 Publisher

Publisher has only one task – to publish data within one topic. There are 2 publish methods available: one with and one without sample timestamp. In case of method without a source timestamp current time is used as a sample timestamp.

Timestamp is a double value representing seconds past 1.1.1970 UTC (to micro precision). There are utility classes that help conversion from/to standardized timestamp values/structures.

Publish methods accept timeout parameter. The method blocks until the message is sent. If the method execution time exceeds timeout time limit TimeoutException is thrown. A non-blocking behavior can be achieved by using timeout value of zero.

Depending on whether data structure contains a key there are two ways of publishing, as shown on the following examples.

This class is thread-safe.

C++

```
try {
   std::shared_ptr<::elt::mal::Mal> ddsMal = mal::loadMal("dds",␣
↪mal::Mal::Properties{
     });
   ::elt::mal::CiiFactory &factory = mal::CiiFactory::getInstance();
   factory.registerMal("dds", ddsMal);

   mal::Uri uri("dds.ps://100/m1/CabinetTelemetry");
   // Create publisher with deadline and latency QoS
   std::unique_ptr<ps::Publisher<mal::example::Sample>> publisher = factory.
↪getPublisher<mal::example::Sample>(
     uri,
     { std::make_shared<mal::ps::qos::Latency>(
         std::chrono::milliseconds(100)),
       std::make_shared<mal::ps::qos::Deadline>(
         std::chrono::seconds(1)) }, {});

   //
   // publish code for keyless topic
   // and user provided source timestamp
   //
   std::shared_ptr<mal::example::Sample> sample = publisher->createDataEntity();
   sample->setDaqId(0);
   sample->setValue(12.8);
   double timestamp = 1550791384.1;
   publisher->publish(timestamp, *sample, std::chrono::seconds(3));

   //
   // publish code for keyed topic
```

(continues on next page)

(continued from previous page)

```
   //
   std::shared_ptr<mal::example::Sample> keySample = publisher->
↪createDataEntity();
   // set key field(s)
   keySample->setDaqId(1);
   {
     std::unique_ptr<::elt::mal::ps::InstancePublisher<mal::example::Sample>>␣
↪daq1Publisher =
         publisher->createInstancePublisher(*keySample);
     std::shared_ptr<mal::example::Sample> daq1Sample = daq1Publisher->
↪createDataEntity();
     // daqId is already set as it is created by InstancePublisher
     daq1Sample->setValue(3);
     daq1Publisher->publish(*daq1Sample, std::chrono::seconds(3));
   }
   // daq1Publisher is destroyed here, since its enclosing scope ended
 } catch (std::exception& exc) {
   std::cerr << "EXCEPTION: " << exc.what() << std::endl;
 }
```

Python

Python does not support method overloading. Publisher provides publish() and publishWithTimestamp() methods. publishWithTimestamp() takes timestamp value as first argument. Type of the timestamp value must be float (as returned from time.time()).

Durations and timeouts must be specified as instances of standard datetime.timedelta class.

```
# dds mal for domain id 100
uri = 'dds.ps://100/m1/CabinetTelemetry'

ddsMal = mal.loadMal('dds', {})
factory = mal.CiiFactory.getInstance()
factory.registerMal('dds', ddsMal)

# Create publisher with deadline and latency QoS.
qosList = [mal.ps.qos.Deadline(datetime.timedelta(seconds=1)),
           mal.ps.qos.Latency(datetime.timedelta(milliseconds=100))]
with factory.getPublisher(uri, Example.Sample,
                          qos=qosList) as publisher:
    #
    # Publish code for keyless topic
    # and user provided source timestamp.
    #

    sample = publisher.createDataEntity()
    sample.setDaqId(0)
```

(continues on next page)

(continued from previous page)

```
    sample.setValue(12.8)
    timestamp = time.time()
    publisher.publishWithTimestamp(sample,
                                   timestamp,
                                   datetime.timedelta(seconds=1))

    #
    # Publish code for keyed topic.
    #

    keySample = publisher.createDataEntity()

    # Set key field(s).
    keySample.setDaqId(1)

    # Use Publisher as context manager.
    with publisher.createInstancePublisher(keySample) as daq1Publisher:
        daq1Sample = daq1Publisher.createDataEntity()
        daq1Sample.setValue(3.14)
        daq1Publisher.publish(daq1Sample, DEFAULT_TIMEOUT)
    # daq1Publisher is closed here.
```

2.5.2 Subscriber

Each subscriber instance has its own queue. The queue holds the events related to the subscribed topic ordered by the time of insertion into the queue (e.g. data arrival). The events are usually data events, they simply contain the data published, however there are also events that indicate e.g. that there is no publisher for given instance available. The events are represented using DataEvent interface.

The interface via its methods provides access the data (if available), its source timestamp, InstanceState and InstanceEvent enumerations. InstanceState always reports current (not the event was recorded) state of an instance: ALIVE or NOT_ALIVE. InstanceEvent describes the reason why event was created: CREATED (instance has been created/revived), UPDATED (updated data event), REMOVED (instance has been destroyed, i.e. no more publishers available for the instance). A programmer should never assume that data is always available, e.g. in case of REMOVED InstanceEvent there is no data. A programmer should always call hasValidData method to check whether the event holds (valid) data.

NOTE: REMOVED instance event is not guaranteed to be generated when instance is being destroyed (due to differences, or lack of notification, in middleware communication software), therefore software should not rely its logic on arrival of this event.

All the data received by subscriber instance (registered to one topic) is first processed by subscriptions on the subscriber. If any of the data matches a subscription a subscription callback is called and the data is provided in the callback, otherwise the data gets into the queue. A programmer has to take care of emptying the queue, otherwise it might get full and no new data can be received. In order to

empty queue user need to invoke one of the read methods, or set deadline QoS property to instruct removal of (too) old data from the queue.

NOTE: You should never close the subscription within the subscription callback as that may cause deadlock, depending on the implementation. If you want your callback to stop processing but you don't want to close the subscription yet, you should pass in an atomic variable and early return from the callback depending on the state of the variable.

NOTE: You should never try to close the subscription or call any subscriber method within the subscription callback as that may cause a deadlock. See *one-shot subscriber* example in icd-demo module on how to handle this case.

NOTE: Subscription listener's methods are never invoked after a call to close the subscription has successfully completed.

The following text describes some of the use-cases shown in the example code below:

1. Read data from all instances. Maximum number of data is limited by maxSamples parameter. To read all the data (and clear the queue) use 0 as maxSamples parameter.
2. If there is no data, an empty (not null) list is returned. This method creates new entities and copies their data; avoid using this method when dealing with large arrays.
3. Read data from one instance. The instance to be read is given as parameter. To create an instance parameter call createDataEntity and set fields that form a key. If queue for specified instance is empty a null reference is returned. This method creates a new instance (there is an exception in case of C++ by-reference variation the method) and copies its data (in all method variations); avoid using it when dealing with large arrays.

Whenever there is a need to avoid any copies of the data (e.g. large arrays), there are load read methods available. If underlying middleware and mapping implementation support no-copy behavior this methods "loan" the data from the middleware and pass the reference to the data to the programmer. However, this is not always possible. If this is not possible (not supported by the implementation) a copy is made. Also be careful that some middleware might support this only for certain data types: if mapping from middleware data type matches MAL data type and no conversion is needed, then no-copy behavior is possible, otherwise a copy (during conversion) is made.

Synchronous read of events (not just data!) can be done using readDataEvents method. The method returns a count of added events to the list provided as an argument. The method blocks until one of the condition is met: number of events added to the list reaches eventCountLimit or collectTime time is passed. Zero eventCountLimit imposes no limit on number of events added, and zero collectTime turns method to non-blocking and only reads what is currently available in the queue (still limited by eventCountLimit count limit). Do not forget that events might not always data event, check by calling hasValidData method if data is valid first.

For asynchronous subscriptions there are 2 methods: one that provides the data in the callback and the one without. The former method guarantees only that data is valid for the time of the callback. The implementation might avoid doing a copy in this case. The later method only notifies about the presence of new data, the data can be read by using one of synchronous read methods later. However, read call must not be invoke inside the callback. If you want to get the data, use the first

method. The callback methods are being executed by some MAL mapping implementation or even middleware threads. Creating asynchronous subscription returns a Subscription instance that is used to cancel the subscription.

Lifetime of subscriptions is limited by the lifetime of a subscriber. If subscriber is being closed the subscriptions are also closed.

In addition there is a method that allows event-pump based subscription. Once subscription is being instantiated a pool method needs to be called in order to get the callback invoked. The callback is being invoked by the same thread that calls poll.

Subscriptions and readDataEvents methods support filtering. Filtering can be done on instance, InstanceState and InstanceEvent fields. Setting a filter on a field means selecting/limiting event to set value. Filter on multiple fields implies all fields must match their set value. Default filter allows all the events. A cached (shared) instance it accessed via DataEventFilter::all() method.

This class is thread-safe.

C++

```
std::shared_ptr<::elt::mal::Mal> ddsMal = mal::loadMal("dds",␣
↪mal::Mal::Properties{
  });
::elt::mal::CiiFactory &factory = mal::CiiFactory::getInstance();
factory.registerMal("dds", ddsMal);

# dds mal for domain id 100
mal::Uri uri("dds.ps://100/m1/CabinetTelemetry");

try {
  std::unique_ptr<ps::Subscriber<mal::example::Sample>> subscriber = factory.
↪getSubscriber<mal::example::Sample>(
      uri, mal::ps::qos::QoS::DEFAULT, {});

  //
  // Read all alive instances w/o sample count limit.
  //
  std::vector<std::shared_ptr<mal::example::Sample>> allAliveInstances =␣
↪subscriber->read(0);

  //
  // Read one instance, i.e. daqId == 1.
  //
  std::shared_ptr<mal::example::Sample> keySample = subscriber->
↪createDataEntity();
  keySample->setDaqId(1);
  std::shared_ptr<mal::example::Sample> daq1Sample = subscriber->
↪readInstance(*keySample);
```

(continues on next page)

(continued from previous page)

```
  // ... or by providing sample by reference
  std::shared_ptr<mal::example::Sample> sample = subscriber->createDataEntity();
  bool sampleRead = subscriber->readInstance(*keySample, *sample);

  //
  // Loaned (no-copy) example.
  // Loan is returned when close() method is called on LoanedDataEntity
  // or automatically when scope containing loaned data entity
  // terminates.

  {
    std::shared_ptr<::elt::mal::ps::LoanedDataEntity<mal::example::Sample>>␣
↪loanedSample = subscriber->loanedRead();
    if (loanedSample) {
      mal::example::Sample &sampleData = loanedSample->getData();
      // use data here
    }
  }

  //
  // Read data events, filtered by instance and instance state.
  // Block until maxEvents are read or timeout occurs.
  //
  using Sample = mal::example::Sample;
  std::vector<std::shared_ptr<mal::ps::DataEvent<Sample>>> events(1000);
  mal::ps::DataEventFilter<Sample> filter;
  filter.setInstance(std::shared_ptr<Sample>(std::move(keySample)));
  filter.setInstanceState(mal::ps::InstanceState::ALIVE);
  filter.setInstanceEvents(
   std::set<mal::ps::InstanceEvent>{mal::ps::InstanceEvent::CREATED,
                                    mal::ps::InstanceEvent::UPDATED});
  std::size_t samplesRead = subscriber->readDataEvents(
      filter,
      events, events.capacity(),
      std::chrono::seconds(3));

  //
  // Synchronous (aka event pump) subscription, no filtering.
  // Poll for 1 second w/ no event count limit.
  //
  {
    std::unique_ptr<mal::ps::Subscription> subscription = subscriber->subscribe(
      mal::ps::DataEventFilter<Sample>::all(),
      [](mal::ps::Subscriber<Sample>& subscriber,
         const mal::ps::DataEvent<Sample>& event) -> void {
           // NOTE: you should never close subscription from
           // within a callback, since that may cause deadlock
           std::cout << event.getData() << std::endl;
           });
```

(continues on next page)

(continued from previous page)

```
    subscriber->poll(10, std::chrono::seconds(1));
  }
  // Note that subscription is destroyed when leaving the scope.

  //
  // Asynchronous subscription, no filtering.
  // 1 second sleep added to simulate some work.
  //
  {
    std::unique_ptr<mal::ps::Subscription> subscription = subscriber->
↪subscribeAsync(
      mal::ps::DataEventFilter<Sample>::all(),
      [](mal::ps::Subscriber<Sample>& subscriber,
         const mal::ps::DataEvent<Sample>& event) -> void {
           // NOTE: you should never close subscription from
           // within a callback, since that may cause deadlock
           std::cout << event.getData() << std::endl;
           });
    std::this_thread::sleep_for(std::chrono::seconds(1));
  }

  //
  // Asynchronous notification subscription, no filtering.
  // In callback sync read is called to read all alive samples.
  //
  {
    std::promise<void> eventPromise;
    std::future<void> eventFuture = eventPromise.get_future();

    std::unique_ptr<mal::ps::Subscription> subscription = subscriber->
↪subscribeAsync(
      mal::ps::DataEventFilter<Sample>::all(),
      [&eventPromise](mal::ps::Subscriber<Sample>& subscriber) -> void {
        // NOTE: you should never close subscription from
        // within a callback, since that may cause deadlock
        eventPromise.set_value();
        });

    std::future_status status = eventFuture.wait_for(std::chrono::seconds(1));
    if (status == std::future_status::ready) {
      std::cout << subscriber->read(0).size() << std::endl;
    } else {
      std::cout << "No event was received\n";
    }
  }
} catch (std::exception& exc) {
  std::cerr << "EXCEPTION: " << exc.what() << std::endl;
}
```

Python

Python subscriber offers (as described above) three versions of the subscribe method:

- subscribe()
  - Creates synchronous subscription. Received data events are passed to provided callback method via poll() call on the subscriber object synchronously.
- subscribeAsync()
  - Creates asynchronous subscription. Provided callback method is called from another thread as new data events arrive.
- subscribeAsyncNotifier()
  - Creates asynchronous subscription. Callback method receives change notifications. Method is called from another thread.

When creating data event filter, data entity class (not instance) must be provided as argument to factory function mal.ps.DataEventFilter.all().

```
DEFAULT_TIMEOUT = datetime.timedelta(seconds=3)
# Poll interval in seconds.
POLL_INTERVAL = datetime.timedelta(seconds=1)
uri = 'dds.ps://100/m1/CabinetTelemetry'
ddsMal = mal.loadMal('dds', {})
factory = mal.CiiFactory.getInstance()
factory.registerMal('dds', ddsMal)
with factory.getSubscriber(uri,
                           Example.Sample,
                           qos=mal.ps.qos.DEFAULT) as subscriber:

    # Read all alive instances w/o sample count limit.

    allAliveInstances = subscriber.read(0)

    # Read one instance i.e. daqId == 1.
    keySample = subscriber.createDataEntity()
    keySample.setDaqId(1)
    daq1sample = subscriber.readInstance(keySample)


    # Loaned (no-copy) example.
    # Loan is returned when close() method is
    # called on LoanedDataEntity. Note: there is
    # no context management protocol available for
    # loaned samples. Use try, finally block.
    loanedSample = subscriber.loanedRead()
    try:
        sampleData = loanedSample.getData()
    finally:
```

(continues on next page)

(continued from previous page)

```
        # Return loan
        loanedSample.close()

    # Read data events, filtered by instance and instance state.
    # Block until maxEvents (1000) are read or timeout (3s) occurs.

    maxEvents = 1000

    # Create new filter for Sample data entity.
    filter = mal.ps.DataEventFilter.all(Sample)
    filter.setInstance(keySample)
    filter.setInstanceState(mal.ps.InstanceState.ALIVE)
    filter.setInstanceEvents(set(mal.ps.InstanceEvent.CREATED,
                                 mal.ps.InstanceEvent.UPDATED))
    # List of samples is returned.
    samples = subscriber.readDataEvents(filter,
                                        maxEvents,
                                        timeout=DEFAULT_TIMEOUT)

    # Asynchronous data subscription, no filtering,
    # 1 second sleep added to simulate some work.

    def dataEventFunction(subscription, dataEvent):
        # NOTE: you should never close subscription from within
        # a callback, since that may cause a deadlock

        # Data event processing method
        if dataEvent.hasValidData():
            sample = dataEvent.getData()
            # use sample
        else:
            # Note that in Python it is possible to create a subscriber
            # that expects samples with different data type that
            # are provided by publisher. This is especially difficult
            # to detect without this code (and the only way to find
            # out is to study the log files).
            # Usually subscriber receives non valid data when
            # publisher instance is not alive anymore, so in case
            # publisher instance is still alive, our subscription might
            # not be correct. The code below handles this case.
            instance_state = dataEvent.getInstanceState()
            if instance_state == mal.ps.InstanceState.ALIVE:
               print('WARNING: There is something wrong with data, please '
                     ' check your publisher and subscriber data types')

    with subscriber.subscribeAsync(
        mal.ps.DataEventFilter.all(Example.Sample),
                                   dataEventFunction) as subscription:
        # Suspend main thread, dataEventFunction is
```

(continues on next page)

(continued from previous page)

```
        # called from another thread.
        time.sleep(1)
    # Note that subscription is destroyed once with block terminates.

    # Asynchronous data subscription, no filtering,
    # using lambda callback.
    # DO NOT CALL read methods n the callback!
    # 1 second sleep added to simulate some work.
    with subscriber.subscribeAsync(mal.ps.DataEventFilter.all(),
                                   lambda subscriber, dataEvent:
                                   print(dataEvent.getData())) as subscription:
        # Suspend main thread, data is printed from another thread.
        time.sleep(1)

    # Synchronous (aka event pump) subscription, no filtering.
    # Poll with 1000 events / 1 seconds limit, endless loop.
    with subscriber.subscribe(mal.ps.DataEventFilter.all(ExampleSample),
                              dataEventFunction) as subscription:
        while True:
            # dataEventFunction will be called synchronously
            # when new event arrives.
            subscriber.poll(maxEvents,
                            datetime.timedelta(seconds=1))
```

In Python it is possible to create subscriber that does not expect data with data type equal to data type of published event. This case is hard to detect by only checking whether event has valid data. The situation is detected within subscription monitor and indicated in the event with publisher instance state set to ALIVE, so additional check can be done in the event processing method to handle this case. See Python example above.

2.5.3 MrvSubscriber

MrvSubscriber is a convenience variant of a subscriber intended to be used by GUIs. It has only one method that always blocks for a given amount of time. This might be used to implement a refresh loop. Use zero time to read samples from a queue and do not block for any more time. The method always returns a list of most-recent values for each instance. It does not return only changed instances over specified amount. If instance becomes unavailable it is returned in the list anymore. This method creates new entities and copies their data; avoid using this method when dealing with large arrays.

This class is thread-safe.

C++

```
std::shared_ptr<::elt::mal::Mal> ddsMal = mal::loadMal("dds", {});
::elt::mal::CiiFactory &factory = mal::CiiFactory::getInstance();
factory.registerMal("dds", ddsMal);

mal::Uri uri("dds.ps://100/m1/CabinetTelemetry");

try {
  std::unique_ptr<ps::MrvSubscriber<mal::example::Sample>> mrvSubscriber =
     factory.getMrvSubscriber<mal::example::Sample>(uri,␣
↪mal::ps::qos::QoS::DEFAULT, {});

  //
  // Every 1 second get most-recent-values of alive instance
  //
  while (true) {
    std::vector<std::shared_ptr<mal::example::Sample>> mrvSamples =
      mrvSubscriber->readMostRecent(std::chrono::seconds(1));
    for (const auto &sample : mrvSamples) {
      std::cout << "Sample " << sample->getDaqId() << " / " <<
        sample->getValue() << std::endl;
    }
  }
} catch (std::exception &exc) {
  std::cerr << "EXCEPTION: " << exc.what() << std::endl;
}
```

Python

```
uri = 'dds.ps://100/m1/CabinetTelemetry'
# Load DDS MAL mapping.
ddsMal = mal.loadMal('dds', {})
factory = mal.CiiFactory.getInstance()
factory.registerMal('dds', ddsMal)

with factory.getMrvSubscriber(uri,
                              Example.Sample,
                              qos=mal.ps.qos.DEFAULT) as mrvSubscriber:

    # Every second get most-recent-value of alive instances.

    timeout = datetime.timedelta(seconds=1)
    while True:
        mrvSamples = mrvSubscriber.readMostRecent(timeout)
        if mrvSamples:
            print(mrvSamples)
```

2.6 Request-response

Request-response is a coupled peer-to-peer client-server communication pattern. A server exposes to the network one or more instances of ICD-defined interfaces (services). A client connects to the server and invokes one or more methods on the services. MAL guarantees reliable QoS (i.e. request cannot be lost) and exactly-once delivery (i.e. for each method invocation exactly one request is made on the server-side; the request can neither be lost nor duplicated). MAL completely takes care of connection management. There is no need to explicitly issue a connect request, monitor connection status and do a reconnect. MAL takes care of this in the background. A programmer only needs to specify a timeout limit on the request. (Re-)connection time is included in the total execution time of the request). If connection cannot be (re-)established in that time or response is not delivered in time a TimeoutException is thrown. If response is delivered after the timeout expired the response is ignored. Any connection loss during the invocation of the method results in DisconnectedException, i.e. MAL does not try to invoke the method on the server side again after the reconnection.

In order to explicitly wait for connection to be established a connect method returns a future object that completes when the connection is established for the first time.

In cases where a programmer is interested in connection state it can register a connection listener on a client instance. The callbacks are called by MAL mapping or middleware thread.

When multiple responses are needed asynchronous method invocation (AMI) MAL support can be used. Basically, AMI is an asynchronous request with one or more asynchronous responses grouped as request. There is no limitation on number of responses nor time limit on lifetime. Lifetime is limited for the time of connection. Once connection is lost AMI request gets canceled (reported to the client as cancellation exception).

Note: With migration to FastDDS, Request-Response communication with MAL DDS is no longer supported.

2.6.1 ICD Interface

The example implementations below are based on the following ICD

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

  <package name="elt">
    <package name="mal">
      <package name="example">

        <struct name="Sample" trace="true">
          <member name="daqId" type="int64_t" key="true" />
          <member name="value" type="double" />
        </struct>
```

(continues on next page)

(continued from previous page)

```
        <struct name="SpeedArray" trace="true">
          <member name="values" type="float" arrayDimensions="[10]"/>
        </struct>

        <struct name="SampleStringWithInt">
          <member name="stringValue" type="string" key="true" />
          <member name="intValue" type="int64_t" />
        </struct>

        <struct name="SampleVector">
          <member name="arrayValue" type="float" arrayDimensions="(16)" />
        </struct>

        <struct name="SampleMultidimension">
          <member name="matrixValue" type="string" arrayDimensions="[10,2]"/>
        </struct>

        <enum name="Enumeration">
          <enumerator name="value1"/>
          <enumerator name="value2"/>
          <enumerator name="value3"/>
        </enum>

        <struct name="SampleEnum">
          <member name="enumValue" type="nonBasic" nonBasicTypeName="Enumeration
↪" />
        </struct>

        <enum name="Command">
          <enumerator name="START_COMMAND" />
          <enumerator name="STOP_COMMAND" />
        </enum>

        <exception name="TooFast" />

        <interface name="RobotControl">
          <method name="command" returnType="void">
            <argument name="com" type="nonBasic" nonBasicTypeName="Command" />
          </method>
          <method name="setSpeed" returnType="float" throws="TooFast">
            <argument name="value" type="float" />
          </method>
          <method name="getSpeed" returnType="float" />
          <method name="validSpeedValues" returnType="boolean">
            <argument name="values" type="float" arrayDimensions="[10]" />
          </method>
          <method name="systemCheck" returnType="string" methodType="ami" />
          <method name="exitRequest" returnType="void"/>
        </interface>
```

(continues on next page)

(continued from previous page)

```
      </package>
    </package>
  </package>
</types>
```

2.6.2 Client

When an interface is defined in ICD, a synchronous and asynchronous client variants of the interfaces are generated. A programmer has a choice to select at client instantiation what interface to use. Invoking a method on synchronous interface will block and return its value via method return mechanism, or exception will be thrown. A request timeout can be set via ReplyTime QoS or via client instance created using timeout method. A TimeoutException is thrown in case of a timeout condition. Asynchronous methods always return a future object. A programmer then gets a response, or an exception, back via a future object. There can be more than one asynchronous requests issued/pending concurrently on one client instance.

Type of the interface on the client side does not affect (in any way) how a request is processed on the server side.

AMI methods can be recognized as methods that return Ami<returnType> object. The object is basically an iterator of future objects of returnType type. The future objects behave exactly the same as in case of asynchronous methods – a future object is a promise waiting for the next response. AMI call is completed when the iterator runs out of elements or when exception is being thrown (also thrown in case of disconnection).

The example code below illustrates the usage of a client interface and using all three types of methods.

C++

```
std::shared_ptr<::elt::mal::Mal> zpbMal = mal::loadMal("zpb", {});
::elt::mal::CiiFactory &factory = mal::CiiFactory::getInstance();

factory.registerMal("zpb", zpbMal);

mal::Uri uri("zpb.rr:///m1/Robot1");

//
// Synchronous client example w/ ReplyTime QoS set.
// RobotControlSync interface is requested.
//
try {
  std::unique_ptr<mal::example::RobotControlSync> robot = factory.getClient
↪<mal::example::RobotControlSync>(
    uri,
```

(continues on next page)

(continued from previous page)

```
  {std::make_shared<mal::rr::qos::ReplyTime>(std::chrono::seconds(3))},
  {});

//
// Explicitly wait for connection to be established.
// (This is usually not needed.)
//
::elt::mal::future<void> connectionFuture = robot->asyncConnect();
std::future_status futureStatus =
  connectionFuture.wait_for(::boost::chrono::seconds(10));
bool connected = (futureStatus == std::future_status::ready);

//
// Invoke synchronous methods
// TimeoutException is throw if default timeout is exceeded.
//

robot->command(mal::example::RobotControl::Command::START_COMMAND);

try {
  float speed = robot->getSpeed();

  mal::shared_vector<const float> speedArray = {66.6, 41.0, 90.5};
  bool checker = robot->validSpeedValues(speedArray);
  if(checker) {
    speed += 30;
    }
  else {
    speed += 3;
  }

  robot->setSpeed(speed);
} catch (mal::example::RobotControl::TooFast& tf) {
  // handle too fast
}

//
// Multiple responses use-case using
// asynchronous method invocation (AMI).
//
std::shared_ptr<StrAmi> ami = robot->systemCheck();
for (auto future : *ami) {
    std::cout << future.get() << std::endl;
}

//
// Create a new client instance w/ specific timeout.
//
robot->timeout(std::chrono::seconds(10))
```

(continues on next page)

(continued from previous page)

```
    ->command(mal::example::RobotControl::Command::STOP_COMMAND);
} catch (std::exception& exc) {
  std::cerr << "EXCEPTION: " << exc.what() << std::endl;
}

//
// Asynchronous client example w/ reply timeout set.
// RobotControlAsync interface is requested.
//
try {
  std::unique_ptr<mal::example::RobotControlAsync> robot = factory.getClient
↪<mal::example::RobotControlAsync>(
      uri,
      {std::make_shared<mal::rr::qos::ReplyTime>
         (std::chrono::seconds(3))},
      {});

  //
  // Basic case, invoke and then wait, if necessary
  //
  mal::future<void> cfuture = robot->command
   (mal::example::RobotControl::Command::START_COMMAND);
  // ... do some work here ...,
  // and get response (void in this case), future will block if
  // the response is not yet received, or TimeoutException is throws in
  // reply-time is exceeded
  cfuture.get();

  // Chained future<> example:
  // async getSpeed() is called and on return setSpeed
  // with new value requested, exceptions are handled.
  // Wait for 2 seconds to complete.
  try {
    mal::future<float> future = robot->
     getSpeed().then([&](::mal::future<float> f)
       { return robot->setSpeed(f.get() + 0.1); }).get();

    if (future.wait_for(boost::chrono::seconds(2)) !=
        ::boost::future_status::ready) {
        // handle timeout
    }

  } catch (mal::example::RobotControl::TooFast& tf) {
    // handle too fast
  }

  //
  // Multiple responses use-case using
  // asynchronous method invocation (AMI).
```

(continues on next page)

(continued from previous page)

```
  //
  std::shared_ptr<StrAmi> ami = robot->systemCheck();
  for (auto future : *ami) {
      std::cout << future.get() << std::endl;
  }
} catch (std::exception& exc) {
  std::cerr << "EXCEPTION: " << exc.what() << std::endl;
}
```

Python

When using asynchronous client, result of the method call on the client reference is always a Future object. Actual result value must be obtained with get method on the result object.

To check state of the Future object, use its check method. Method takes no arguments and returns enumeration mal.rr.FutureStatus that can have the one of the following values:

- DEFERRED – requested method was not started yet.
- READY – requested method completed and result can be retrieved with call to get method.
- TIMEOUT – wait was performed on the Future, but timeout occurred.

```
import elt.pymal as mal
# Import synchronous client implementation
from ModManualExamples.Elt.Mal.Example.RobotControl import RobotControlSync
# Import asynchronous client implementation
from ModManualExamples.Elt.Mal.Example.RobotControl import RobotControlAsync
from ModManualExamples.Elt.Mal.Example import Command, TooFast

# 3 seconds.
THREE_SECONDS = datetime.timedelta(seconds=3)

# Minute
MINUTE = datetime.timedelta(seconds=60)

def _main():
    """Main function implementation"""

    # sync example
    try:
        # Load ZPB MAL mapping.
        uri = 'zpb.rr:///m1/Robot1'
        zpbMal = mal.loadMal('zpb', {})
        factory = mal.CiiFactory.getInstance()
        factory.registerMal('zpb', zpbMal)

        # Synchronous client example w/ ReplyTime QoS timeout set.
```

(continues on next page)

(continued from previous page)

```
        # RobotControlSync interface is requested.
        with factory.getClient(uri,
                               RobotControlSync,
                               qos=mal.rr.qos.ReplyTime(THREE_SECONDS)) as robot:

            # Explicitly wait for connection to be established.
            # (This is usually not needed).
            connectionFuture = robot.asyncConnect()
            connectionFuture.wait_for(THREE_SECONDS)

            # Invoke synchronous methods
            # TimeoutError is thrown if default timeout is exceeded.

            robot.command(Command.START_COMMAND)
            try:
                speed = robot.getSpeed()

                speedArray = mal.makeSharedVector('f', 3, 90.5)
                speedArray = speedArray.freeze()
                checker = robot.validSpeedValues(speedArray)
                if checker:
                  speed += 30
                else:
                  speed += 3

                robot.setSpeed(speed)
            except TooFast as e:
                # Handle too fast
                pass

            # Multiple responses use-case using
            # asynchronous method invocation (AMI).

            systemCheckAmi = robot.systemCheck()
            # We are not using context manager protocol here.
            # Once we are done with the AMI, it should be
            # closed().
            try:
                # Result set is iterable.
                for msg in systemCheckAmi:
                    print(msg)
            finally:
                systemCheckAmi.close()

        # Asynchronous example w/reply timeout set.
        # RobotControlAync interface is requested.
        with factory.getClient(uri,
                               RobotControlAsync,
                               qos=mal.rr.qos.ReplyTime(THREE_SECONDS)) as robot:
```

(continues on next page)

(continued from previous page)

```
            # Basic case, invoke and then wait, if necessary
            future = robot.command(Command.START_COMMAND)

            # ... do some work here ...
            # and get response (None in this case). Future will block
            # it the response is not yet received or TimeoutError is
            # thrown when reply time is exceeded.

            future.get()

            # Multiple responses use-case using
            # asynchronous method invocation (AMI).
            # Handling is the same as for sync. client.

        with robot.systemCheck() as systemCheckAmi:
            for future in systemCheckAmi:
                print('System Check: ', future.get())
except Exception as e:
    # Handle exceptions or
    raise
```

2.6.3 Server

Creating a server instance just creates a container where services can be registered. A service is a uniquely named instance reachable over the network. There can be multiple services of the same or different interface type registered to the server. Same service instance can also be registered multiple times using different names. The server will just propagate the request to a service by its name. Once a service is unregistered it is no longer reachable over the network. Any pending requests on unregistered service will complete normally.

When an interface is defined in ICD, a synchronous and asynchronous service variants of the interfaces are generated. A programmer needs to implement one of these interfaces. Only these interfaces can be registered as a service. An interface type dictates how requests are processed on the server side. Server processes requests of all the services using one thread. Once a request is received it decodes the request and calls a method on specified service. If the service implements a synchronous interface then invocation of its method will block. This includes processing of other requests of the same and other services. In case of asynchronous service the method will block just for a method to return a future object. The server will continue to process other pending requests. The server will send the response of asynchronous services back to the client immediately when the return value, or an exception, is given to the future object. This implies that asynchronous service responses might come back in a different order than their requests were issued.

When a C++ MAL service throws a known exception (i.e. declared in the MAL-ICD), it it is passed through network to the client, where it is re-raised. However, if the service throws other exceptions: a MAL exception of a type that is not declared for this particular method, a standard exception, or

throws a C++ object that is not derived from the std::exception, then MAL will behave as shown in Table 1.

ICD	Exc. type	Server behaviour	Client behaviour
throws	MAL excepti ion but other than declared	catch and propaga te UnhandledExcep tion	receive UnhandledRemoteExcep tion
throws	non-MAL C++ exception, e.g. std::ou t_of_range	terminate	MalException: Client disconnected
throws	non-MAL, non C++ exception (std:string)	terminate	MalException: Client disconnected
no throws	MalException although none was declared	catch and propaga te UnhandledExcep tion	receive UnhandledRemoteExcep tion
no throws	non-MAL, C++ exception, e.g. std::ou t_of_range	terminate	MalException: Client disconnected
no throws	non-MAL, non C++ exception (std:string)	terminate	MalException: Client disconnected

Table 1 MAL behaviour when an interface method throws an undeclared exception

Implementation of server-side AMI methods is the same for synchronous and asynchronous service interfaces. AMI methods declarations return the same Ami<returnType> as on the client side. However, the service needs to provide the implementation of the Ami<returnType>. The creation of it is done via ServerContextProvider singleton instance implemented as thread-safe-storage instance that provides access to server-side helper methods. A createAmi method will create a ServerAmi<returnType> instance (that implements Ami<returnType> interface). The implementation of the AMI method just needs to setup asynchronous execution of the method (the method should not block!) and return the ServerAmi<returnType> instance. Sending responses to the client is done by calling one of the three methods on the ServerAmi<returnType> instance: complete (non-last response), completeExceptionally (exception response, last response) and completed (last response). Once last response is sent the instance gets automatically released.

See the examples below on how to implement synchronous and asynchronous service.

The server has no responsibility over the lifecycle of registered services.

C++

```
namespace mal = ::elt::mal;

/**
 * Simulate some work
 */
static void simulateWork() {
  std::this_thread::sleep_for(std::chrono::seconds(2));
}
```

(continues on next page)

(continued from previous page)

```
/**
 * Synchronous RobotControl implementation example,
 * must implement RobotControl interface.
 */
class RobotControlImpl: public virtual mal::example::RobotControl {
 public:

  RobotControlImpl(float max, mal::shared_vector<const float> speeds)
    : maxSpeed(max), tempSpeeds(speeds) {
      allSpeeds = {0.0};
 }

  void command(RobotControl::Command command) override {
    switch (command) {
      case RobotControl::Command::START_COMMAND:
        allSpeeds = tempSpeeds;
        break;

      case RobotControl::Command::STOP_COMMAND:
        allSpeeds = {0.0};
        break;

      default:
        throw std::invalid_argument("RobotControlImpl: Unknown command");
    }
  }

  float setSpeed(float newSpeed) override {
    if (speed > maxSpeed) {
      throw RobotControl::TooFast(
       "RobotControlImpl::setSpeed, speed too high");
    }

    ::elt::mal::shared_vector<float> mutableSpeeds = thaw(allSpeeds);
    mutableSpeeds.push_back(newSpeed);
    allSpeeds = freeze(mutableSpeeds);

    return allSpeeds.back()
  }

  float getSpeed() const override { return allSpeeds.back(); }

  bool validSpeedValues(const ::elt::mal::shared_vector<const float>& values)␣
↪override {
    bool valid = true;
    for (const auto& val: values) {
      if(val < 0 || val > maxSpeed) {
        valid = false;
        break;
```

(continues on next page)

(continued from previous page)

```
      }
    }
    return valid;
  }

  std::shared_ptr<RobotControl::StrAmi> systemCheck() override {
    // Obtain ptr to ServerAmi
    std::shared_ptr<RobotControl::StrAmi> ami = mal::rr::ServerContextProvider
      <mal::example::ServerContextImpl<std::string>>::
      getInstance().createAmi();

    auto future = boost::async(boost::launch::async, [=]() {
      simulateWork();
      ami->complete("Battery OK");
      simulateWork();
      ami->complete("Drivertrain OK");
      simulateWork();
      ami->completed("Engine OK");
  });

  // Return ptr to Ami
  return ami;
}

 private:
  float maxSpeed;
  mal::shared_vector<const float> allSpeeds;
  mal::shared_vector<const float> tempSpeeds;
};

/**
 * Asynchronous RobotControl implementation example,
 * must implement AsyncRobotControl interface.
 */
class AsyncRobotControlImpl: public virtual mal::example::AsyncRobotControl {
public:

  AsyncRobotControlImpl(float max, mal::shared_vector<const float> speeds)
    : maxSpeed(max), tempSpeeds(speeds) {
      allSpeeds = {0.0}
  }

  mal::future<void> command(mal::example::RobotControl::Command command)
   override {
    std::function<void()> f = [this, command]() -> void {
      std::lock_guard<std::mutex> guard(speedMutex);
      switch (command) {
        case mal::example::RobotControl::Command::START_COMMAND:
          allSpeeds = tempSpeeds;
```

(continues on next page)

(continued from previous page)

```
          break;
        case mal::example::RobotControl::Command::STOP_COMMAND:
          allSpeeds = {0.0};
          break;

        default:
          throw std::invalid_argument("RobotControlImpl: Unknown command");
      }
    };
    return boost::async(boost::launch::async, f);
  }

  mal::future<float> setSpeed(float newSpeed) override {
    return boost::async(boost::launch::async, [this, newSpeed]() -> float {
      std::lock_guard<std::mutex> guard(speedMutex);
      if (newSpeed > maxSpeed) {
        throw mal::example::RobotControl::TooFast(
         "RobotControlImpl::setSpeed, speed too high");
      }

      ::elt::mal::shared_vector<float> mutableSpeeds = thaw(allSpeeds);
      mutableSpeeds.push_back(newSpeed);
      allSpeeds = freeze(mutableSpeeds);

      return allSpeeds.back()
    });
  }

  mal::future<float> getSpeed() const override {
   boost::promise<float> promise;
   promise.set_value(allSpeeds.back());
   return promise.get_future();
  }

  mal::future<float> validSpeedValues(const ::elt::mal::shared_vector<const␣
↪float>& values) override {
    boost::promise<bool> promise;

    bool valid = true;
    for (const auto& val: values) {
      if(val < 0 || val > maxSpeed) {
        valid = false;
        break;
      }
    }

    promise.set_value(valid);
    return promise.get_future();
  }
```

(continues on next page)

(continued from previous page)

```
  std::shared_ptr<AsyncRobotControl::StrAmi> systemCheck() override {
    // Obtain ptr to ServerAmi
    std::shared_ptr<AsyncRobotControl::StrAmi> ami =␣
↪mal::rr::ServerContextProvider
      <mal::example::ServerContextImpl<std::string>>::
      getInstance().createAmi();

    auto future = boost::async(::boost::launch::async, [=]() {
      simulateWork();
      ami->complete("Battery OK");
      simulateWork();
      ami->complete("Drivertrain OK");
      simulateWork();
      ami->completed("Engine OK");
  });

  // Return ptr to Ami
  return ami;
}

 private:
  float maxSpeed;
  std::mutex speedMutex;
  mal::shared_vector<const float> allSpeeds;
  mal::shared_vector<const float> tempSpeeds;
};

/**
 * Server example.
 *
 * Main entry point.
 * @param argc number of command line arguments
 * @param argv command line arguments
 */
int main(int argc, char** argv) {

  // Load ZPB MAL mapping
  std::shared_ptr<::elt::mal::Mal> zpbMal = mal::loadMal("zpb", {});
  // Register mapping with CiiFactory
  ::elt:: &factory = mal::CiiFactory::getInstance();
  factory.registerMal("zpb", zpbMal);

  mal::Uri uri("zpb.rr:///m1");

  try {
    //
    // Create server, default QoS.
    // Currently there are not MAL QoS relevant for the Server
```

(continues on next page)

(continued from previous page)

```
    //
    std::unique_ptr<rr::Server> server = factory.createServer(uri,␣
↪mal::rr::qos::QoS::DEFAULT, {});

    mal::shared_vector<const float> speedsArray = {0.1, 49.4, 74.4};

    // Register three Robot services.
    // They are all separate instances.
    // First two instance have synchronous implementation,
    // third has asynchronous.
    std::shared_ptr<mal::rr::RrEntity> robot1(new RobotControlImpl(120.0,␣
↪speedsArray));
    std::shared_ptr<mal::rr::RrEntity> robot2(new RobotControlImpl(80.0,␣
↪speedsArray));
    std::shared_ptr<mal::rr::RrEntity> robot3(new AsyncRobotControlImpl(80.0,␣
↪speedsArray));

    server->registerService<mal::example::RobotControl, true>
     ("Robot1", robot1);
    server->registerService<mal::example::RobotControl, true>
     ("Robot2", robot2);
    server->registerService<mal::example::RobotControl, false>
     ("Robot3", robot3);

    // Process commands until server->shutdown() is called.
    server->run();
  } catch (std::exception& exc) {
    std::cerr << "EXCEPTION: " << exc.what() << std::endl;
  }
  return 0;
}
```

Python

```
import datetime
import time
import threading
from concurrent.futures import ThreadPoolExecutor

# import top level Python MAL API module
import elt.pymal as mal
# import RobotControl binding
from ModManualExamples.Elt.Mal.Example import RobotControl
# import Command enumeration and TooFast exception
from ModManualExamples.Elt.Mal.Example import Command, TooFast
```

(continues on next page)

(continued from previous page)

```
def simulateWork():
  time.sleep(1.0)

class RobotControlSyncImpl:
    """Synchronous RobotControl implementation example, must
       implement methods defined in the ICD definition file:
       command, setSpeed, getSpeed, systemCheck
    """
    def __init__(self, maxSpeed, speeds):
        """constructor"""
        self._allSpeeds = [0.0]
        self._maxSpeed = maxSpeed
        self._tempSpeeds = speeds

    def command(self, com):
        """Handle command"""
        if com == Command.START_COMMAND:
            self._allSpeeds = self._tempSpeeds
        elif com == Command.END_COMMAND:
            self._allSpeeds = [0.0]
        else:
          raise ValueError('Invalid command value')

    def setSpeed(self, speed):
        """Set new speed.
           Throws TooFast when speed exceeds maximum speed.
        """
        if speed > self._maxSpeed:
            raise TooFast()
        self._allSpeeds.append(speed)
        return self._allSpeeds[-1]

    def getSpeed(self):
        """Return current speed"""
        return self._allSpeeds[-1]

    def validSpeedValues(self, values):
        """Return Validation of entry Values"""
        result = True
        for val in values:
         if val < 0 or val > self._maxSpeed:
             result = False
             break

        return result

    def systemCheck(self):
        """Perform system check"""
        # Obtain instance to the server context.
```

(continues on next page)

(continued from previous page)

```
        # This call uses ServerContextProvider.getInstance()
        # method internally.
        context = RobotControl.ServerContextString.getInstance()
        # Obtain server AMI
        serverAmi = context.createAmi()
        # Setup asynchronous execution
        systemCheckThread = threading.Thread(self._systemCheck,
          args=(serverAmi,))
        systemCheckThread.start()
        return serverAmi

    def _systemCheck(self, ami):
        """Actual systemCheck implementation"""
        simulateWork()
        # first response
        ami.complete('Battery OK')

        simulateWork()
        # second response
        ami.complete("Drivetrain OK")

        simulateWork()
        # final response, use completed method
        ami.completed("Engine OK")

class RobotControlAsyncImpl:
    """Asynchronous RobotControl implementation example
       all methods that don't return Ami, must return
       concurrent.futures.Future object
    """
    def __init__(self, maxSpeed, speeds):
      """constructor"""
      self._executor = ThreadPoolExecutor(max_workers=4)
      self._allSpeeds = [0.0]
      self._maxSpeed = maxSpeed
      self._tempSpeeds = speeds

    def command(self, com):
        """Handle command"""
        return self._executor.submit(self, _command, com)

    def _command(self, com):
        """Command implementation"""
        if com == Command.START_COMMAND:
            self._allSpeeds = self._tempSpeeds
        elif com == Command.END_COMMAND:
            self._allSpeeds = [0.0]
        else:
          raise ValueError('Invalid command value')
```

(continues on next page)

(continued from previous page)

```
    def setSpeed(self, speed):
        """Set speed"""
        self._executor.submit(self._setSpeed, speed)

    def _setSpeed(self, speed):
        """setSpeed implementation
           Throws TooFast when speed exceeds maximum speed.
        """
        if speed > self._maxSpeed:
            raise TooFast()
        self._allSpeeds.append(speed)
        return self._allSpeeds[-1]

    def getSpeed(self):
        """Get speed"""
        return self._executor.submit(lambda x: x, self._allSpeeds[-1])

    def validSpeedValues(self, values):
        """Check if values are inside the limitations"""
        return self._executor.submit(self._validSpeedValues, values)

    def _validSpeedValues(self, values):
        """validSpeedValues implementation
           Sanity check for list of values.
        """
        result = True
        for val in values:
         if val < 0 or val > self._maxSpeed:
           result = False
           break

        return result

    def systemCheck(self):
        """Perform system check"""
        # Obtain instance to the server context.
        # This call uses ServerContextProvider.getInstance()
        # method internally.
        context = RobotControl.ServerContextString.getInstance()
        # Obtain server AMI
        serverAmi = context.createAmi()
        # Setup asynchronous execution
        systemCheckThread = threading.Thread(self._systemCheck,
          args=(serverAmi,))
        systemCheckThread.start()
        return serverAmi

    def _systemCheck(self, ami):
```

(continues on next page)

(continued from previous page)

```
        """Actual systemCheck implementation"""
        simulateWork()
        # first response
        ami.complete('Battery OK')

        simulateWork()
        # second response
        ami.complete("Drivetrain OK")

        simulateWork()
        # final response, use completed method
        ami.completed("Engine OK")


def main():
    uri = 'zpb.rr:///m1'

    # Load ZPB MAL mapping
    zpbMal = mal.loadMal('zpb')
    factory = CiiFactory.getInstance()
    factory.registerMal('zpb', zpbMal)

    with factory.createServer(uri, mal.rr.qos.DEFAULT, {}) as server:

        speedsArray = [0.1, 49.4, 74.4]

        # Register three Robot services.
        # They are all separate instances.
        # First two have synchronous implementation,
        # the third has asynchronous implementation.
        server.registerService('Robot1',
                               RobotControl.RobotControlSyncService,
                               RobotControlSyncImpl(120.0, speedsArray))
        server.registerService('Robot2',
                               RobotControl.RobotControlSyncService,
                               RobotControlSyncImpl(80.0, speedsArray))
        server.registerService('Robot3',
                               RobotControl.RobotControlAsyncService,
                               RobotControlAsyncImpl(20.0, speedsArray))

        server.run()
```

2.7 Entities

As a general rule, entities must be instantiated through factory methods provided by the MAL (and likewise provided by MAL Publishers). If the entity is complex, the factory method deep-creates also the nested entities inside. To populate the complex entity, you have to get the nested entities from it, and populate them.

Caution: for some datatypes and middlewares, after getting the nested entity from the complex entity and populating it, you must (counter-intuitively) finally also call the setter on the complex entity. This is needed to trigger the marshalling of the data. We are seeing to improve this.

Caution: the C++ life-cycle of the child entities is bound to the life-cycle of its parent live-cycle. MAL implementation might use a middleware-specific structure to hold an entity data. The parent entity is the owner of the structure. Due to performance reasons (requirements), all the fields including child fields, are written directly to the structure without using any local storage. The middleware might not support ownership hierarchy, therefore the child entities are bound to the live-cycle of the specific structure owned by the parent. With the parent entity gone, the child entity would point to an invalid location of the previously located specific structure. If you need a child entity outside of the scope of a parent entity life-cycle, you need to clone the child entity, e.g. parent->getChild()->clone(). Do not assume that the smart pointers will take care of this limitation since the child entities do not hold a smart reference to the parent entities.

2.7.1 Array, Blob

To transfer ICD arrays, MAL uses the Shared Vector utility class, which is similar to the standard vector (or list) classes in the respective languages and additionally supports “freeze” and “thaw”. Only a “thawed” vector can be modified, only a “frozen” vector can be sent over MAL.

Python

Example: shared vector (used for sending/receiving arrays)

```
import elt.pymal as mal

# usage: makeSharedVector (<type>, <size>, <value>)
# <type>: float, double, string, uint8_t, int8_t (etc.)
vec = mal.makeSharedVector('int8_t', 10, 120)

print (vec)       # output: SharedVectorInt8[x, x, x, x, x, x, x, x, x, x]
print (vec[2:])   # output: SharedVectorInt8[x, x, x, x, x, x, x, x]

# before sending over MAL
vec_to_send = vec.freeze()

# vector to list
aslist = list (vec)
```

(continues on next page)

(continued from previous page)

```
print (aslist)   # output: [120, 120, 120, 120, 120, 120, 120, 120, 120, 120]

# list to vector
vec2 = mal.SharedVectorInt8 (aslist)
```

Example: blob

```
complex_entity = publisher.createDataEntity()
nested_blob = complex_entity.getNested()
vec = mal.makeSharedVector('int8_t', 512, 0x55)
vec = vec.freeze()
nested_blob.setBlob (vec)
```

C++

Example: directly map one-dimensional arrays into your preferred library objects, e.g. for Eigen:

```
using Eigen;

elt::mal::shared_vector<double> values = ...

// square 3x3 matrix
Matrix3d mat1(values.data());

// more advanced mapping of 3x3 matrix
Map<Matrix<double, 3, 3, RowMajor> mat2(values.data());
```

2.8 Appendix

2.8.1 Building CII application with WAF

A CII application must be built using waf/wtools that has ICD generator (icd-gen tool) and all other supported middleware generators are integrated. All the artifacts are built into property formed libraries that are needed to be dynamically loaded by MAL mappings. Steps to build a CII application is the following:

- Create a project directory with a project wscript file, e.g.:

```
from wtools.project import declare_project

wtools.project.declare_project('icd-demo-folder', '0.1-dev-version',        #␣
↪Project directory name and version
                               recurse='icd-app1 icd-app2',                 #␣
↪Name of all applications folders
                               requires='cxx boost java python protoc cii', #␣
↪Primary libraries to be linked
                               boost_libs='log log_setup thread system')    #␣
↪Secondary libraries to be linked
                                                                            # ...
```

Project directory will contain as many applications necessary with the correspondent ICD XML files. Following the above example code 2 application will be implemented. So, after creating the applications directories:

- Inside bought application directory, 2 subdirectories should be created: ‘icd’ (contains ICD files for appliciations) and ‘app’ (where the application implementation is located). Names of those two subdirectories can be different.
- Parallel to this folders a wscript must be added with the following sctruture:

```
from wtools import package

# Declare external libraries to be used specifically by the respective␣
↪application
def configure(conf):
    """System installed libraries"""
    conf.check_cfg(package='opentracing_api', uselib_store='OPENTRACE', args='--
↪cflags --libs')

    """Created WAF libraries based on other applications"""
    conf.check_wdep(wdep_name='cii.mal-common', uselib_store='MALCOMMON')
    conf.check_wdep(wdep_name="client-api.config", uselib_store="CLIENTCONF")

package.declare_package(recurse='app icd') # Folders inside application␣
↪directory
```

Jumping in too the declared packeges (‘app’ and ‘icd’), each one must have a wscript file. Starting with

the ‘app’ subdirectories, is worth noting that depending on the developer language the WAF method call to create the correspondent executable file will change (‘declare_cprogram’ ‘declare_pyprogram’ ‘declare_jar’):

```
from wtools.module import declare_cprogram

"""From pre-checked libraries in previous folder 'wscript', call the ones needed␣
↪for the respective application"""
declare_cprogram(target='executable-app-language', use='BOOST CLIENTCONF␣
↪MALCOMMON OPENTRACE')
```

And the ‘icd’ subdirectories with a wscript like:

```
from wtools.module import declare_malicd

declare_malicd() # Enables ICD artifacts generation
```

Remember that when building the project, the WAF Tool will only use files that exist inside a ‘src’ folder. So all necessary coding for the application to work as entended must be inside a source directory. In the end a developer should have a structure like this:

```
root
`-- icd-demo-folder
    |-- icd-app1
    |    |-- app
    |    |   |-- src
    |    |   |   `-- main-app1-file.xxx
    |    |   `-- wscript
    |    |-- icd
    |    |   |-- src
    |    |   |   `-- icd-app1-file.xml
    |    |   `-- wscript
    |    `-- wscript
    |
    |-- icd-app2
    |    |-- app
    |    |   |-- src
    |    |   |   `-- main-app2-file.xxx
    |    |   `-- wscript
    |    |-- icd
    |    |   |-- src
    |    |   |   `-- icd-app2-file.xml
    |    |   `-- wscript
    |    `-- wscript
    |
    `-- wscript
```

See icd-demo project as example.

CHAPTER

THREE

MAL ICD GENERATION

Document ID:	
Revision:	1.7
Last modification:	March 18, 2024
Status:	Release
Repository:	https://gitlab.eso.org/cii/info/cii-docs
File:	mal_icd.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
0.1	2019-02-21	dkumar	All	Created.
1.0	2019-02-26	dkumar	All	Review, updated missing sections.
1.1	2019-04-02	mvitorovic	6	Added description for topics.
1.2	2019-04-10	msekoranja, dkumar	All	Review, updated ICD limitations
1.3	2019-04-24	dkumar	6.3, 6.4	Updated topics namespace names.
1.4	2019-06-06	msekoranja	4.9	Referencing types from other packages note added.
1.5	2023-03-15	mschilli	3.5	Info about icd-gen splitter
1.6	2023-03-30	rleao	3.4.8	arrays in different middlewares
1.7	18.03.2024	mschilli	0	Public doc

Confidentiality

This document is classified as Public.

Scope

This document is a manual for the Interface Control Document language of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

Glossary of Terms

API	Application Programmers Interface
CII	Core Integration Infrastructure
MAL	Middleware Abstraction Layer
ICD	Interface Control Document

References

1. Cosylab, ELT CII MAL Transfer document, CSL-DOC-18-168015, version 1.8
2. Cosylab, ELT CII MAL API User's Manual, CSL-DOC-17-150198, version 2.1
3. Cosylab, Interface Control Document, Specification, CSL-DOC-17-147262, version 1.6

4. Cosylab, Data Addressing Specification, CSL-DOC-17-147264, version 1.2

3.1 Overview

This document is a user manual for developing Interface Control Documents, specifically Type Definition Documents holding ICD types.

3.2 Installation

See ELT CII MAL Transfer document [1], Chapter 1.2 on how to build and install elt-mal.

3.2.1 Prerequisites

In order to use icd generation the following prerequisites must be fulfilled:

- CII MAL prerequisites, see ELT CII MAL Transfer document [1], Chapter 1.1.
- Installed elt-mal.

3.3 Editing ICD file in Eclipse

Developing Interface Control Documents involves writing XML documents. Developer is encouraged to use a XML editor that has support for XML content assist and support for a schema validation. Eclipse that is part of ESO ELT Development Environment has support for editing XML files with XML content assist and schema validation.

See more at https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.wst.xmleditor.doc.user%2Ftopics%2FcworkXML.html.

For the purposes of this user guide, complete the following steps which will enable support for XML content assist and schema validation for **icdDemo.xml** within Eclipse:

1. Open Eclipse (/opt/eclipse/oxygen/eclipse/eclipse)
2. Click **File > New > Project**.
3. Select **General > Project** and click **Next**.
4. Enter **icd-demo** as 'Project name' and click **Finish**.
5. To import the ICD XML schemas into the workbench, click **File > Import**.
6. Select **General > File System** and click **Next**.
7. Click **Browse** on the next page of the wizard to select the directories from which you would like to add the XML schemas (i.e. $INTROOT/interface/schemas)
8. Select check box on 'schemas' which will import all schemas under the directory $PREFIX/interface/schemas.
9. Make sure that option 'Create top-level folder' is selected and then click **Finish**.
10. Next step will create a Waf package named **icd** containing **icdDemo.xml**. Open the terminal and execute the following commands:

```
mkdir -p ~/icd-demo/icd/src

echo '<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">
</types>' > ~/icd-demo/icd/src/icdDemo.xml

  echo 'from wtools.module import declare_malicd
declare_malicd()' > ~/icd-demo/icd/wscript
```

11. XML content assist and schema validation for **icdDemo.xml** will work within Eclipse only when the file icdDemo.xml is beside directory schemas/ in the Eclipse project. A soft link needs to be created in <user_eclipse_worskpace>/icd-demo, pointing to the icdDemo.xml that is part of Waf package. Open the terminal and execute the following commands (replace **user_eclipse_workspace** with a real path):

```
cd <user_eclipse_workspace>/icd-demo/

ln -s ~/icd-demo/icd/src/icdDemo.xml .
```

12. Refreshing the Eclipse project (pressing F5) should show the icdDemo.xml as part of the icd-demo Eclipse project.

Support for the XML content assist should work by placing the cursor into the type's element body and pressing **Ctrl + Space**. The schema validity for icdDemo.xml is being checked as you type or it can be activated manually by Right-clicking on your file in the Navigator view and selecting Validate. Figure 1 is showing the XML content assist and a schema validation error (invalid package name) when editing the icdDemo.xml.

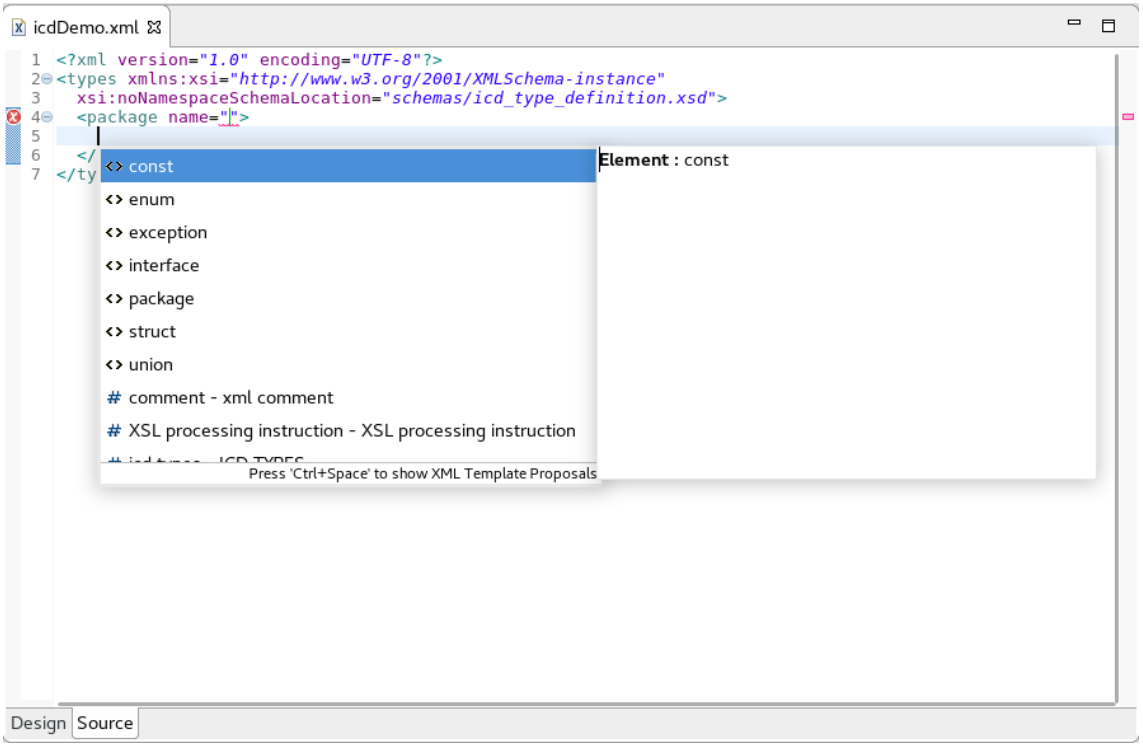


Figure 1: XML content assist and schema validation on icdDemo.xml

3.4 ICD Building blocks

Interface Control Document defines two types of XML documents [3], and this document is describes **Type definition document** ([3], 1.1) that contains structured and interface types in chapters 4 and 5. **Topic and interface definition document** ([3], 1.2) is described in chapter 6.

ICD Type definition document is composed of:

1. **Packages** that group related types and prevent naming conflicts on similarly named types.
2. **Structures** that group data elements together under one name. These data elements, known as members, can be a primitive type or other complex types such as structures and unions.
3. **Basic types** are primitive programming language types (including string) and standard ICD types (blob, timestamp).
4. **Enumerations** that are distinct types whose value is restricted to a range of values that include several explicitly named constants (“enumerators”).
5. **Unions** are a special types that can hold only one of its data members at a time.
6. **Interfaces** describe related methods belonging to the same interface.
7. **Exceptions** define exceptional events that occur during the execution of interface methods.
8. **Constants** which are values used in the elements and attributes where a primitive value would be used.

Within one package types with duplicate names are not allowed. In addition some keywords are reserved (see [3], section 1.1.8.1)

3.4.1 Package

All ICD types belong to a package. There are no package-less types. Package has a non-empty unqualified name (i.e. must not contain other package names delimited with scope resolution operator ::). Package can include other packages but only the most inner package can hold types. ICD XML schema makes sure that the package constraints are maintained in the generation phase but also while the developer is creating the ICD in IDE (i.e. Eclipse).

Listing 1 and Listing 2 contain an example of a single and a nested package.

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">
  <package name="elt">
    <!-- type definition content -->
  </package>
</types>
```

Listing 1: Single package

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

  <package name="elt">
    <package name="icdDemo">
      <!-- type definition content -->
    </package>
  </package>
</types>
```

Listing 2: Nested package

See Interface Control Document Specification for an example how package is mapped to Java and C++ ([3], 1.3.5).

3.4.2 Basic types

Basic types define:

1. Primitive types of members in structures and unions.
2. Types of union discriminators.
3. Define method arguments of primitive types or constant types.

Only basic types and enums can be keys in structures.

Basic types are:

- **boolean**, boolean primitive type holding true or false.
- **int8_t**, signed integer type with width of exactly 8 bits.
- **int16_t**, signed integer type with width of exactly 16 bits.
- **int32_t**, signed integer type with width of exactly 32 bits.
- **int64_t**, signed integer type with width of exactly 64 bits.
- **uint8_t**, unsigned integer type with width of exactly 8 bits.
- **uint16_t**. signed integer type with width of exactly 16 bits.
- **uint32_t**, unsigned integer type with width of exactly 32 bits.
- **uint64_t**, unsigned integer type with width of exactly 64 bits.
- **float**, single precision floating point type IEEE-754, 32 bit.
- **double**, double precision floating point type. IEEE-754 64 bit.
- **string,** string of characters. Length upper limit is 256 Bytes.
- **blob**, stream of bytes.

- **timestamp**, timestamp.

Listing 3 is showing a usage of basic types in different structured types and constants.

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">
  <package name="elt">

      <struct name="SimpleStruct">
     <member name="boolval" type="boolean"/>
     <member name="int8val" type="int8_t" key="true"/>
     <member name="uint8val" type="uint8_t" key="true"/>
   </struct>

   <const name="int8Const" type="int8_t" value="1"/>
   <const name="floatConst" type="float" value="6.4523424"/>

   <union name="SimpleUnion">
        <discriminator type="uint8_t"/>
    <case>
      <caseDiscriminator value="0"/>
      <member name="foo" type="string"/>
    </case>
    <case>
      <caseDiscriminator value="1"/>
      <member name="bar" type="int8_t"/>
    </case>
  </union>
 </package>
</types>
```

Listing 3: Usage of basic types for members in a structure, in constants and member of unions

3.4.3 Constants

Constants define values used in the elements and attributes where a primitive value would be used.

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

  <package name="elt">
     <const name="boolconst" type="boolean" value="true"/>
     <const name="int8const" type="int8_t" value="1"/>
     <const name="uint8const" type="uint8_t" value="2"/>
     <const name="int16const" type="int16_t" value="-3"/>
     <const name="uint16const" type="uint16_t" value="100"/>
     <const name="int32const" type="int32_t" value="-3200"/>
```

(continues on next page)

(continued from previous page)

```
    <const name="uint32const" type="uint32_t" value="100"/>
    <const name="floatconst" type="float" value="100.0"/>
  <const name="stringconst" type="string" value="batman"/>
</package>
</types>
```

Listing 4: Defining constants

3.4.4 Enumeration (Enum)

Enumeration is a distinct type whose value is restricted to a range of values explicitly named constants (enumerators). Enumeration has a non-empty unqualified name and at least one enumerator. The enumeration type is a building block for unions and can be included as a member of a structure and member of a union.

Listing 5 is showing an example of simple command modeled as an enumeration.

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

  <package name="elt">
      <enum name="Command">
    <enumerator name="START_COMMAND"/>
    <enumerator name="STOP_COMMAND"/>
   </enum>
 </package>
</types>
```

Listing 5: Enumerator Command

See Interface Control Document Specification for an example how enumeration is mapped to Java and C++ ([3], 1.3.1).

3.4.5 Union

Union is a special type that can hold only one of its data members at a time. The discriminator sets which member holds a value. Only numeric primitives and enums can be union discriminators. Union members can be: primitive types, unions, structures and enums. No arrays are allowed as union members.

Listing 6 and Listing 7 are showing examples of a union with a primitive type discriminator and a union with an enumerator as a union discriminator.

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

(continues on next page)

(continued from previous page)

```
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">
  <package name="elt">
   <union name="SimpleUnion">
        <discriminator type="uint8_t"/>
    <case>
      <caseDiscriminator value="0"/>
      <member name="foo" type="string"/>
    </case>
    <case>
      <caseDiscriminator value="1"/>
      <member name="bar" type="int8_t"/>
    </case>
  </union>
 </package>
</types>
```

Listing 6: Union with numeric primitive type discriminator

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

  <package name="elt">


      <enum name="Command">
     <enumerator name="START_COMMAND"/>
     <enumerator name="STOP_COMMAND"/>
   </enum>

   <union name="CommandData">
        <discriminator type="nonBasic" nonBasicTypeName="Command"/>
    <case>
      <caseDiscriminator value="START_COMMAND"/>
      <member name="startCommandData" type="string"/>
    </case>
    <case>
      <caseDiscriminator value="STOP_COMMAND"/>
      <member name="stopCommandData" type="int8_t"/>
    </case>
  </union>
 </package>
</types>
```

Listing 7: Union with enum discriminator

See Interface Control Document Specification for an example how union is mapped to Java and C++ ([3], 1.3.3).

3.4.6 Structure (struct)

Structure groups data elements (members) together under one name. Structure member can have a basic type or other structure type, union or enumeration type. Every structure can be assigned a key. A key is a set of fields (of primitive types) that uniquely identifies one instance. This allows programmers to publish/retrieve/filter data based on their instance. Only basic types and enums can be keys in structures. Structure can be extended. Extending structure means COPY (not inheritance!) from extended, including the keys.

Listing 8, Listing 9, Listing 10 and Listing 11 show typical type definitions of a structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

  <package name="elt">

      <struct name="SimpleStruct">
     <member name="boolval" type="boolean"/>
     <member name="int8val" type="int8_t" key="true"/>
     <member name="uint8val" type="uint8_t" key="true"/>
     <member name="int16val" type="int16_t"/>
     <member name="uint16val" type="uint16_t"/>
     <member name="int32val" type="int32_t"/>
     <member name="uint32val" type="uint32_t"/>
   </struct>
</package>
</types>
```

Listing 8: Structure with basic types as members

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

  <package name="elt">

   <enum name="Command">
     <enumerator name="START_COMMAND"/>
     <enumerator name="STOP_COMMAND"/>
   </enum>

      <struct name="ArrayStruct">
     <member name="boolvalArray" type="boolean" arrayDimensions="(16)"/>
     <member name="int8valArray" type="int8_t" arrayDimensions="(16)"/>
     <member name="CmdArray" type="nonBasic"
             nonBasicTypeName="Command" arrayDimensions="(20)"/>
   </struct>
</package>
</types>
```

Listing 9: Structure with arrays as members

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

  <package name="elt">

   <struct name="SimpleStruct">
     <member name="boolval" type="boolean"/>
     <member name="int8val" type="int8_t" key="true"/>
   </struct>

      <struct name="ComplexStruct">
     <member name="boolval" type="boolean"/>
     <member name="int8val" type="int8_t"/>
     <member name="nestedStruct" type="nonBasic"
             nonBasicTypeName="SimpleStruct"/>
   </struct>
</package>
</types>
```

Listing 10: Structure holding another structure as a member.

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

  <package name="elt">

   <struct name="BaseStruct">
     <member name="boolval" type="boolean"/>
     <member name="int8val" type="int8_t" key="true"/>
   </struct>

      <struct name="ExtendedStruct" baseType="BaseStruct">
     <member name="boolval" type="boolean"/>
     <member name="int8val" type="int8_t"/>
     <member name="nestedStruct" type="nonBasic"
             nonBasicTypeName="SimpleStruct"/>
   </struct>
</package>
</types>
```

Listing 11: Extending a structure.

See Interface Control Document Specification for an example how structure is mapped to Java and C++ ([3], 1.3.2).

3.4.7 Interface and Exception

A server exposes to the network one or more instances of ICD-defined interface (a service). A client connects to the server and invokes one or more methods (method element) on that service. One reserved method name is “serviceDescription” - that method is implemented by the ICD and cannot be declared in an ICD interface definition. Methods defined in ICD Interface can throw (throws attribute) exceptions raising exceptional events during the execution of methods. Exception can have optional members with additional exception data. Exception type cannot publicly extend other exception types, while the interface can be extended (extends element). Extending interface means copying from the extended interface. Methods that are part of the interface can have zero or more arguments (argument attribute). Methods return results (returnType attribute). When multiple responses (results) are needed, asynchronous method invocation (AMI) MAL support can be used (methodType attribute). Declaring the AMI method with a void return-type is banned and it will produce a compilation error.

Listing 12 shows a typical service definition with different types of methods.

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

  <package name="elt">

   <struct name="SpeedArray" trace="true">
     <member name="values" type="float" arrayDimensions="(10)"/>
   </struct>

   <enum name="Command">
     <enumerator name="START_COMMAND"/>
     <enumerator name="STOP_COMMAND"/>
   </enum>

   <exception name="TooFast">
     <member name="speed" type="float"/>
   </exception>

   <interface name="Logger">
  <method name="setLevel" returnType="void">
       <argument name="level" type="uint8_t"/>
     </method>
   </interface>

      <interface name="RobotControl">

     <extends interfaceName="Logger"/>

     <method name="command" returnType="void">
       <argument name="com" type="nonBasic" nonBasicTypeName="Command"/>
     </method>
```

(continues on next page)

(continued from previous page)

```
    <method name="setSpeed" returnType="float" throws="TooFast">
      <argument name="value" type="float" />
    </method>
    <method name="getSpeed" returnType="float" />
    <method name="validSpeedValues" returnType="boolean">
      <argument name="values" type="float" arrayDimensions="(10)" />
    </method>

    <method name="systemCheck" returnType="string" methodType="ami"/>

    <method name="setStrVec" returnType="void">
      <argument name="strvec" type="string" arrayDimensions="(5)"/>
    </method>

    <method name="getStrVec" returnType="string" arrayDimensions="(5)"/>

  </interface>
</package>
</types>
```

Listing 12: RobotControl service definition

See Interface Control Document Specification for an example how an exception and an interface are mapped to Java and C++ ([3], 1.3.4 and 1.3.6).

3.4.8 Arrays

Arrays are specified with an attribute **arrayDimensions** on:

- Member of structured type (struct, union, exception): member of a structured type is an array.
- Method return type: return type of method is an array.
- Method parameter: method parameter is an array

Arrays are one of:

- BOUNDED: arrayDimensions=”(dim1, dim2, . . . , dimN)” where dim1, dim2, . . . , dimN > 0
- FIXED: arrayDimensions=”[dim1, dim2, . . . , dimN]” where dim1, dim2, . . . , dimN > 0

Fixed array is sent over the wire in full capacity, regardless how many elements are actually stored the array. On the other hand, in a bounded array, only currently stored elements will be sent, however the middleware might reserve entire capacity to avoid reallocation at runtime. Setting boundaries too big might cause in high memory usage.

It is also relevant to warn that some types of communication will not comply with the Bounded/Fixed array types and it’s dimensional definitions. That is the case of ZPB communication method, because the actual array will be a repeated field in the protobuf. The following table shows how the array behave in a ZPD and DDS protocols:

Protocol/Pattern	4 Passed	2 Passed	1 Passed	Empty
ZPB RR Bounded (2)	4 Received	2 Received	1 Received	Empty
ZPB RR Fixed [2]	4 Received	2 Received	1 Received	Empty
ZPB PS Bounded (2)	4 Received	2 Received	1 Received	Empty
ZPB PS Fixed [2]	4 Received	2 Received	1 Received	Empty
DDS PS Bounded (2)	4 Received	2 Received	1 Received	Empty
DDS PS Fixed [2]	Exc on Publish	2 Received	Exc on Publish	Exc on Publish

Next is an ICD XML example file that shows how to declare an array:

```
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_type_definition.xsd">

  <package name="elt">

   <struct name="SpeedArray" trace="true">
     <member name="values" type="float" arrayDimensions="(10)"/>
   </struct>

   <struct name="SampleMultidimension">
     <member name="matrixValue" type="string" arrayDimensions="[3,2]"/>
   </struct>

  </package>
</types>
```

Listing 13: Example of a Structure Array Specification

3.5 ICD Generation

When the ICD Type definition documents are written, MAL agnostic and MAL middlelayer specific types need to be generated based on the types defined in them. The ICD generator (icd-gen tool installed in $PREFIX/bin) must never be used directly by the user. ICD generation has been integrated into waf/wtools.

User needs to create a waf package composed of ICD Type definition document stored in *src/* directory and use declare_malicd() wtools support in package wscript file. During installation process all ICDs gets installed in $PREFIX/interface/icd directory so that they can be included by other ICDs.

During the generation process, the icd generator splits ICD files into smaller chunks. This was added as an optimisation for memory allocation and compilation duration, and is fully transparent to the users. Should there ever be a need, you can enable debug-logs from the splitter, set the environment variable *ICD_PYTHON_CPP_SPLITTER_DEBUG* to non-empty. To prevent the splitting, set the environment variable *ICD_WITHOUT_PYTHON_CPP_SPLITTER* to non-empty.

See ELT CII MAL API User's Manual [2], Chapter 7.1 on how to create a CII application using waf/wtools named icd-demo (containing icdDemo.xml) that was created in Chapter 3.

3.6 Topic and interface definition document

The ICD generation tool can also be used to parse the **Topic and interface definition document** and generate language specific source files that expose various parameters of the definition to the developer as constant values that can be used application source code.

This type of document defines the following elements per each subsystem:

1. **Messaging pattern** that may be publish/subscribe topics or method definitions for interfaces.
2. **Quality of Service** parameters for both of the above.
3. **Performance characteristics** for both of the above.
4. **Type and interface names**; these must be defined in the type definition document.
5. **Middleware mappings**; the document lists only the names of the protocols that apply.
6. **Address URI** as defined in [4].

In order to enable topic and interface definition generation use –topic (-t) command-line option when calling icd-gen.

3.6.1 Subsystems

Each document lists on or multiple subsystems. At the moment the only subsystem type supported is **device**.

```
<?xml version="1.0" encoding="UTF-8"?>
<topics xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="schemas/icd_topic_definition.xsd">
  <device name="Dev1">
    <!-- subsystem definition content -->
  </device>
  <device name="Dev2">
    <!-- subsystem definition content -->
  </device>
</topics>
```

Listing 16: Subsystems (devices)

3.6.2 Messaging pattern

Each subsystem may contain one or multiple publish/subscribe topics or service interface definitions. The element for publish/subscribe topic is named **pubsub_topic**, and the element for the service interface definition is named **service**.

```
<?xml version="1.0" encoding="UTF-8"?>
<topics xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

(continues on next page)

(continued from previous page)

```
       xsi:noNamespaceSchemaLocation="schemas/icd_topic_definition.xsd">
  <device name="Dev1">
    <pubsub_topic>
      <!-- topic definition content -->
    </pubsub_topic>
  </device>
  <device name="Dev2">
    <service name="MSA1" interface="MSAzimuth">
      <!-- subsystem definition content -->
    </service>
  </device>
</topics>
```

Listing 17: Topic and service definition

3.6.3 Publish subscribe topic

The publish/subscribe topic definition contains the following required elements:

1. **topic_name** – this element defines the name of the topic.
2. **topic_type** – this element defines the type name of the structure defined in the type definition document (see chapter 4).
3. **address_uri** – this element contains the MAL address URI.
4. **qos** – this element defines the Quality of Service attributes.
5. **performance** - this element defines the performance attributes.
6. **mal** – this element defines provider specific elements, however it is not processed by the tool.

For a detailed description the elements and their attributes please see [3].

Based on the XML definition from Listing 18 the tool generates the code shown in Listing 19, Listing 20 and Listing 21.

```
<?xml version="1.0" encoding="UTF-8"?>
<topics xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_topic_definition.xsd">
  <device name="Dev1">
    <pubsub_topic>
      <topic_name>Az:Pos_Actual</topic_name>
      <topic_type>Pos_Actual</topic_type>
      <address_uri>dds.ps://m1/Pos_Actual?p1=v1&amp;p2=v2</address_uri>
      <qos latency_ms="0.1" deadline_ms="20" />
      <performance rate_hz="0.2" latency_ms="25" synchronous="false" />
      <mal>
        <dds />
      </mal>
```

(continues on next page)

(continued from previous page)

```
    </pubsub_topic>
  </device>
</topics>
```

Listing 18: Topic definition (full)

For all Java classes the package name is predefined. The device is a public class with static class members for all defined topics (and service interfaces). The name of the topic class is derived from the **topic_name** element with the **PubSub** suffix. The class member constants are derived from the other elements listed above.

```
package elt.icd.topicdefinition;

public class Dev1 {

    public static class AzPosActualPubSub {
        public static String TYPE = "Pos_Actual";
        public static String ADDRESS_URI = "dds.ps://m1/Pos_Actual?p1=v1&p2=v2";
        public static double QOS_LATENCY_MS = 0.1;
        public static double QOS_DEADLINE_MS = 20.0;
        public static double PERFORMANCE_RATE_HZ = 0.2;
        public static double PERFORMANCE_LATENCY_MS = 25;
        public static boolean PERFORMANCE_SYNCHRONOUS = false;
    } // AzPosActualPubSub

} // Dev1
```

Listing 19: Topic Java code

For C++ the tool generates only the header file. The top 3 namespaces are predefined. The device is a namespace with namespace members for all defined topics (and service interfaces). The name of the topic namespace is derived from the **topic_name** element with the **PubSub** suffix. The constants are derived from the other elements listed above.

```
#ifndef __DEV_1_HPP
#define __DEV_1_HPP


namespace elt
{
namespace icd
{
namespace topicdefinition
{

namespace Dev1
{

  namespace AzPosActualPubSub
```

(continues on next page)

(continued from previous page)

```
  {
    constexpr char[] type = "Pos_Actual";
    constexpr char[] addressUri = "dds.ps://m1/Pos_Actual?p1=v1&p2=v2";
    constexpr double qosLatencyMs = 0.1;
    constexpr double qosDeadlineMs = 20.0;
    constexpr double performanceRateHz = 0.2;
    constexpr double performanceLatencyMs = 25;
    constexpr bool performanceSynchronous = false;
  } // namespace AzPosActualPubSub

} // namespace Dev1

} // namespace topicdefinition
} // namespace icd
} // namespace elt

#endif // __DEV_1_HPP
```

Listing 20: Topic C++ code

For Python the tool generates a class for each device with class members for all defined topics (and service interfaces). The name of the topic class is derived from the **topic_name** element with the **PubSub** suffix. The members are derived from the other elements listed above.

```
class Dev1:

  class AzPosActualPubSub:
    type = "Pos_Actual";
    addressUri = "dds.ps://m1/Pos_Actual?p1=v1&p2=v2";
    qosLatencyMs = 0.1;
    qosDeadlineMs = 20.0;
    performanceRateHz = 0.2;
    performanceLatencyMs = 25;
    performanceSynchronous = False;

  # end AzPosActualPubSub

# end Dev1
```

Listing 21: Topic Python code

3.6.4 Service interface

The attributes of each **service** definition are the service instance name and the interface type name. It contains the following required elements:

1. **address_uri** – this element contains the MAL address URI.
2. **performance** – this element defines the performance attributes.
3. **qos** – this element defines the Quality of Service attributes.
4. **is_configuration** – the configuration has no defined performance constraints and represents aspects of the Local Control System software which must be controllable externally from the CCS. This element is optional.
5. **mal** – this element defines provider specific elements, however it is not processed by the tool.
6. **methods** – this element contains all the method definitions for this service.

The **methods** element contains one or many **method** elements. Each element has a **name** attribute specifying the method name. Each method contains the following optional elements which override the values listed above:

1. **performance** – this element overrides the performance attributes.
2. **qos** – this element overrides the Quality of Service attributes.
3. **is_configuration** – this element overrides the configuration setting.

```
<?xml version="1.0" encoding="UTF-8"?>
<topics xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schemas/icd_topic_definition.xsd">
  <device name="Dev2">
    <service name="MSA1" interface="MSAzimuth">
      <address_uri>zpb.rr://m4lsv.pl.eso.org:5002</address_uri>
      <performance rate_hz="0.2" latency_ms="25" synchronous="false" />
      <qos connection_timeout_sec="0.1" reply_timeout_sec="20" />
      <mal>
        <zpb />
      </mal>
      <methods>
        <method name="SomeMethodThatIsPartOfMSAzimuth">
          <performance rate_hz="10" latency_ms="2" synchronous="false" />
          <qos connection_timeout_sec="0.1" reply_timeout_sec="20" />
        </method>
        <method name="GetLogLevel">
          <is_configuration>true</is_configuration>
        </method>
      </methods>
    </service>
  </device>
</topics>
```

Listing 22: Service definition (full)

For all Java classes the package name is predefined. The device is a public class with static class members for all defined service interfaces (and topics). The name of the service class is derived from the service **name** attribute with the **Service** suffix. The **Methods** class name is also predefined. The class member constants are derived from the other elements listed above.

```
package elt.icd.topicdefinition;


public class Dev2 {

    public static class MSA1Service {
        public static String INTERFACE = "MSAzimuth";
        public static String ADDRESS_URI = "zpb.rr://m4lsv.pl.eso.org:5002";
        public static double PERFORMANCE_RATE_HZ = 0.2;
        public static double PERFORMANCE_LATENCY_MS = 25;
        public static boolean PERFORMANCE_SYNCHRONOUS = false;
        public static double QOS_CONNECTION_TIMEOUT_SEC = 0.1;
        public static double QOS_REPLY_TIMEOUT_SEC = 20.0;
        public static class Methods {
            public static class SomeMethodThatIsPartOfMSAzimuth {
                public static double QOS_CONNECTION_TIMEOUT_SEC = 0.1;
                public static double QOS_REPLY_TIMEOUT_SEC = 20.0;
                public static double PERFORMANCE_RATE_HZ = 10.0;
                public static double PERFORMANCE_LATENCY_MS = 2;
                public static boolean PERFORMANCE_SYNCHRONOUS = false;
            } // SomeMethodThatIsPartOfMSAzimuth
            public static class GetLogLevel {
                public static boolean IS_CONFIGURATION = true;
            } // GetLogLevel
        } // Methods
    } // MSA1Service

} // Dev2
```

Listing 23: Service Java code

For C++ the tool generates only the header file. The top 3 namespaces are predefined. The device is a namespace with namespace members for all defined service interfaces (and topics). The name of the service namespace is derived from the service **name** attribute with the **Service** suffix. The **Methods** namespace is also predefined. The constants are derived from the other elements listed above.

```
#ifndef __DEV_2_HPP
#define __DEV_2_HPP


namespace elt
{
```

(continues on next page)

(continued from previous page)

```
namespace icd
{
namespace topicdefinition
{

namespace Dev2
{

  namespace MSA1Service
  {
    constexpr char[] interface = "MSAzimuth";
    constexpr char[] addressUri = "zpb.rr://m4lsv.pl.eso.org:5002";
    constexpr double performanceRateHz = 0.2;
    constexpr double performanceLatencyMs = 25;
    constexpr bool performanceSynchronous = false;
    constexpr double qosConnectionTimeoutSec = 0.1;
    constexpr double qosReplyTimeoutSec = 20.0;

    namespace Methods
    {
      namespace SomeMethodThatIsPartOfMSAzimuth
      {
        constexpr double performanceRateHz = 10.0;
        constexpr double performanceLatencyMs = 2;
        constexpr bool performanceSynchronous = false;
        constexpr double qosConnectionTimeoutSec = 0.1;
        constexpr double qosReplyTimeoutSec = 20.0;
      } // namespace SomeMethodThatIsPartOfMSAzimuth
      namespace GetLogLevel
      {
        constexpr bool isConfiguration = true;
      } // namespace GetLogLevel
    } // namespace Methods
  } // namespace MSA1Service

} // namespace Dev2

} // namespace topicdefinition
} // namespace icd
} // namespace elt

#endif // __DEV_2_HPP
```

Listing 24: Service C++ code

For Python the tool generates a class for each device with class members for all defined service interfaces (and topics). The name of the service class is derived from the service **name** attribute with the **Service** suffix. The **Methods** class name is also predefined. The members are derived from the other elements listed above.

```
class Dev2:

  class MSA1Service:
    interface = "MSAzimuth";
    addressUri = "zpb.rr://m4lsv.pl.eso.org:5002";
    performanceRateHz = 0.2;
    performanceLatencyMs = 25;
    performanceSynchronous = False;
    qosConnectionTimeoutSec = 0.1;
    qosReplyTimeoutSec = 20.0;

    class Methods:
      class SomeMethodThatIsPartOfMSAzimuth:
        performanceRateHz = 10.0;
        performanceLatencyMs = 2;
        performanceSynchronous = False;
        qosConnectionTimeoutSec = 0.1;
        qosReplyTimeoutSec = 20.0;

      class GetLogLevel:
        isConfiguration = True;

    # end Methods
  # end MSA1Service

# end Dev2
```

Listing 25: Topic Python code

CHAPTER

FOUR

MAL MAPPINGS

Document ID:	
Revision:	2.10
Last modification:	March 18, 2024
Status:	Released
Repository:	https://gitlab.eso.org/cii/info/cii-docs
File:	mal_mappings.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
0.1	2018-07-21	mseko-ranja	All	Created.
0.9	2018-08-03	mseko-ranja	3	Added Python sections, properties.
1.0	2018-08-10	mseko-ranja	All	Minor updates, release.
2.0	2019-02-02	mseko-ranja	All	Updated property names, removed duplicated parts from MAL API manual and Transfer document.
2.1	2019-04-16	mseko-ranja	3 References	Updated OPC UA C++ properties. References update
2.2	2020-03-24	mseko-ranja	3	Added ZPB C++/Java send queue size property.
2.3	2021-09-07	dkumar	3	Added OPCUA logging description for open62541
2.4	2021-09-24	dkumar	3	Added ZPB C++/Java disable type check property.
2.5	2022-04-26	dkumar	3	Added ZPB C++ enable ephemeral port check
2.6	2022-06-23	dkumar	3	Added MudPi C++ broadcast interface property
2.7	2022-06-30	mschilli	3	Added URI subsections
2.8	2023-07-04	nkornwei	4.3.1	Updated URI subsection
2.9	2024-03-14	mschilli	4.3.1.2	participant name - new property
2.10	18.03.2024	mschilli	0	Public doc

Confidentiality

This document is classified as Public.

Scope

This document is a manual for the Middleware mappings of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

Glossary of Terms

API	Application Programmers Interface
CII	Core Integration Infrastructure
DDS	Data Distribution Service
MAL	Middleware Abstraction Layer
MUDPI	Multicast UDP Interface
URI	Uniform Resource Identifier
ZPB	ZeroMQ and Protocol Buffers
OPC UA	OPC (Object Linking and Embedding for Process Control) Unified Architecture

References

1. Cosylab, MAL Design document, CSL-DOC-17-147260 v1.5
2. Cosylab, ELT CII MAL Transfer document, CSL-DOC-18-168015, version 1.8
3. Cosylab, Python MAL Bindings Design document, CSL-DOC-18-152412, version 1.3
4. Cosylab, MAL API User Manual document, CSL-MAN-17-150198, version 2.1

4.1 Overview

This document describes configuration options for the following MAL mappings: DDS, ZPB and OPC UA. All the details on the MAL API and its functionality are covered by the MAL design document [1] and MAL User Manual [4].

4.2 Installation

See MAL Mappings Transfer document [2] on how to install MAL mappings.

4.3 C++

4.3.1 DDS

URIs

The underlying middleware is multicast. Discovery is by domain id, and topic-oriented. Publisher and Subscriber use the same URI to indicate which topic they both join. Peers (publishers or subscribers) in different domains will not discover each other. Request-Reply is not supported (any more).

- Publisher: *dds.ps://<DOMAIN_ID>/DDS-TOPIC-NAME*
- Subscriber: same as Publisher
- Requester: Not supported

When IPv6 is enabled (through native configuration of the middleware), no changes to the URIs are needed.

Details

CII URI syntax is of the following form: *<scheme>:[//<authority>]</path>[?<query>][#fragment]*. The Authority is used to indicate the DDS domain, while Query and fragment URI components are not used in DDS MAL URIs, therefore the DDS MAL URI is simplified: *<scheme>://<domain_id>/path*

Schema component of DDS MAL URI uses dds.ps string for specifying a DDS middleware publish subscribe profile. The string dds.rr for a DDS middleware request reply profile is no longer supported. A mandatory URI path component needs to hold the name of the topic (for publish subscribe profile). The domain_id is optional, if no specified it is set by the DDS QoS XML, or defaults to domain id 0.

Examples of MAL DDS URIs:

- *dds.ps://100/Sample* - publisher for topicName = Sample in domain id 100
- *dds.ps://100/Sample* - subscriber connected to topicName = Sample in domain id 100

Mal Specific Properties

The following table enumerates mapping specific properties:

Name	Type	Description	Used by	Default
dds.qos.profile.library	string	DDS QoS profile library.	MAL	(none)
dds.qos.profile.name	string	DDS QoS profile name.	MAL	(none)
dds.qos.profile.name. participant	string	DDS QoS profile name for participant.	MAL	(none)
dds.qos.profile.name. topic	string	DDS QoS profile name for topic.	Pub	(none)
dds.qos.profile.name. writer	string	DDS QoS profile name for writer.	Pub	(none)
dds.qos.profile.name. reader	string	DDS QoS profile name for reader.	Sub	(none)
dds.qos.profile.name. subscriber	string	DDS QoS profile name for subscriber.	Sub	(none)
dds.qos.profile.name. publisher	string	DDS QoS profile name for publisher.	Pub	(none)
dds.wait_for_ack. timeout_ms	integer	DDS wait for ack. before tearing down writer and publisher.	MAL	5000ms
dds.qos.participant. name	integer	QoS override: name to assign to participant	MAL	(none)
dds.qos.participant. interfaceWhiteList	integer	QoS override: list of NICs for participant	MAL	(none)

Table 1 C++ DDS MAL mapping specific properties

DDS QoS profile name selection

Custom QoS profiles are enabled only when a user sets the “dds.qos.profile.library” mal property pointing to the file containing the QoS profiles. If the user sets only the “dds.qos.profile.name” mal property, then the same QoS profile name will be used for the creation of the following DDS entities: participant, topic, writer, publisher, reader, and subscriber. User has the option to override the default QoS profile name when creating a participant (“dds.qos.profile.name.participant”), a topic(“dds.qos.profile.name.topic”), a writer (“dds.qos.profile.name.writer”), a reader(“dds.qos.profile.name.reader”), a subscriber(“dds.qos.profile.name.subscriber”), and a publisher(“dds.qos.profile.name.publisher”). If the user does not override the profile name for a specific DDS entity, the default profile name is used.

DDS Wait for acknowledgment timeout property

If this property is not provided, a default timeout of 5000 ms will be assumed as the timeout value. If the timeout value is larger than 0 then writer->wait_for_acknowledgements() and publisher->wait_for_acknowledgements() will be called before tearing down the writers and the publisher. This means that developers can still suppress calling wait_for_acknowledgements if they specify the new property with a timeout value of 0, otherwise the call to wait_for_acks will always happen from now on.

QoS overrides

The MAL DDS offers to modify a few QoS attributes after they were read from an external QoS file.

- *Participant name*: The participant name is displayed in tools like dds-monitor.
- *Participant interface whitelist*: Network interfaces the participant uses (whitespace-separated list).

Note: MAL holds one participant for every domain-id. Accordingly, all entitites (publishers, subscribers) in one domain share the same name and the same interfaces.

Note: These attributes only take effect on creation of a participant, i.e. the moment when the first entity (publisher, subscriber) for any given domain-id gets instantiated.

4.3.2 ZPB

URIs

The underlying middleware is peer-to-peer. Discovery is based on host/port, plus name-filtering per port. A publisher exposes a topic-name on a local port, a subscriber needs to know where the publisher is. A server exposes one or more server-interfaces on a local port, a requester needs to know where the server is.

- Publisher: *zpb.ps://*:PORT/ZPB-TOPIC-NAME*
- Subscriber: *zpb.ps://PUBLISHER-HOST:PUBLISHER-PORT/TOPIC-NAME*
- Requester: *zpb.rr://PUBLISHER-HOST:PUBLISHER-PORT/SERVER-NAME/SERVER-IF-NAME*

ZMQ supports IPv6, but it needs to be enabled on the ZMQ socket as an option (currently not done by the ZPB-MAL). Once it is enabled on the sockets, IPv6 addresses need to be used in the URIs.

Details

ZPB MAL uses zpb.ps for publish-subscribe and zpb.rr for request-reply string for its URI schema component. A server-based authority component must be specified in the URI, without unsupported user-info element. When binding to the socket, i.e. in case of a publisher and request-reply server, an asterisk wildcard character is allowed to specify ANY network interface to bind to. A mandatory URI path component needs to hold the name of the topic or service. A path can be hierarchical. An empty URI path is allowed when creating a server: a path component is used as a service name prefix for all services registered to that server. Query and fragment URI components are not used.

There are helper methods available to create ZPB URIs, e.g. *ZpbUtil.createPsUri(String host, int port, String topicName)* and *ZpbUtil.createRrUri(String host, int port, String instanceName)* .

Examples of ZPB URIs:

- *zpb.ps://*:12345/Sample* - publisher bound on ANY interface on port 12345, topicName = Sample:
- *zpb.ps://eltmgr01:12345/Sample* - subscriber connected to eltmgr01, port = 12345, topicName = Sample:

- *zpb.rr://localhost:5555/RobotControl* - server bound to localhost interface, port = 5555, service name prefix = RobotControl:
- *zpb.rr://10.0.0.1:5555/RobotControl/Robot1* - client connected to 10.0.0.1, port = 5555, serviceName = RobotControl/Robot1:

Mal Specific Properties

The following table enumerates mapping specific properties:

Name	Type	Description	Used by	Default
zpb.ps.slowJoinerDelay Ms	integer	Slow joiner delay. Time in millisecond s to wait after publisher has been instantiate d. This mitigates slow joiner delay problem but does NOT guarantee that first N messages will not be lost.	Publisher	0 (no delay)
zpb.schedulEdExecutorT hreads	integer	Number of thread pool threads to handle internal timer based tasks (e.g. QoS deadline timeouts).	MAL	3
zpb.rr.callTimeout-Sec (C++ only)	integer	Client asynchronou s method call timeout in seconds.	Client	0 (no timeout)
zpb.ps.zmq.sndhwm	integer	Publisher send queue size. Use to limit memory usage when dealing with large messages. 0 stands for unlimited.	Publisher	1000 (ZMQ timeout)
zpb.ps. disableType HashCheck	boolean	Disable subscriber's check for type hash correctness	Subscribers	false
zpb.rr. zmqReconnect IvlMs	integer	Set reconnection interval in ms	Client	100 ms
zpb.rr. zmqReconnect IvlMaxMs	integer	Set maximum reconnection interval in ms	Client	0 ms
zpb.rr. enableEpheme ralPortCheck	boolean	Throw an ex exception if ephemeral po rt is used, otherwise log warning	Server	false

Table 2 C++ ZPB MAL mapping specific properties

4.3.3 OPC UA

URIs

The underlying middleware is connection-oriented. Discovery via host-port. The URI encodes a mapping between ICD-entity and OPC-UA node, expressed as a semicolon-separated list of tuples (namespaceId, objectId), where objectId is either a nodeId or a methodId, e.g. “2,HelloWorld/ScalarTypes/Int64;2,HelloWorld/ScalarTypes/Double”. The order of the tuples must match the order of the fields in the ICD struct.

- Publisher: not applicable
- Subscriber: same as Requester
- Requester: *opcua.rr://SERVER-HOST:SERVER-PORT/NAMESPACE#MAPPING*

OPC-UA supports IPv6. To use it, the SERVER-HOST must be specified according to Format for Literal IPv6 Addresses.

Mal Specific Properties

The following table enumerates mapping specific properties:

Name	Type	Description	Used by	Default
opc.asyncLoopExecutionPeriodMs	integer	The sleep period of the asynchronou s process loop.	MAL	50
opc.asyncCallSubmitTimeoutMs	integer	Timeout for submitting an asynchronou s call.	Client	1000
opc.asyncCallRetryPeri odMs	integer	The retry period for failed asynchronou s calls.	MAL	250
opc.log4cplus.filename	string	log4cplus file name.	MAL	undefined
opc.ps.pollingPeriodMs	integer	Multi-node subscriber polling period in millisecond s.	Subscriber, MrvSubscrib er	1000
opc.ps.outstandingPublishRequests	integer	Number of outstanding publish requests.	Publisher, Subscriber, MrvSubscrib er	10
opc.ps. usePollSubsc riber	boolean	Force subscriber to poll for value updates instead of a monitor subscription	Subscriber, MrvSubscrib er	false
opc.ps.requestedPublishingIntervalMs	integer	Server-side sampling interval of a subscription (i.e. an interval to sample/poll/ monitor an item)	Subscriber, MrvSubscrib er	5000

Table 3 C++ OPC UA MAL mapping specific properties

Data Access

The DA profile is implemented using special dedicated MAL request-response DataAccess interface. The interface is defined as:

```
public interface DataAccess extends RrEntity {
   <T extends DataEntity<T>> T read(Class<T> dataClass, SampleInfo sample_info);
   <T extends DataEntity<T>> void write(T value);
}
```

Synchronous (DataAccessSync) and asynchronous (DataAccessAsync) versions are available that directly issue OPC-UA DA read and write requests to the OPC-UA server. OPC UA nodes need to be specified at client instantiation time using URI fragment component using the following format: *opcua.rr://<host>:<port>/namespace#nsId1,nodeId1;nsId2,nodeId2;. . .;nsIdN,nodeIdN;*

IDs can be either a string or an integer ID, where nsId stands for namespace ID and nodeId for node ID. Node values are then mapped to the structure of type T at the read method call, or structure of type

T mapped to the nodes values at the write method call. Note that order and type of the node values and structure fields must match. In case of mismatch, an exception is thrown. In case of an error or timeout (specified as QoS), an exception is thrown. Connection management is handled by MAL transparently, as for every request-reply interface: connection process is initiated asynchronously and automatically and in case of disconnection a connection is restored automatically.

The SampleInfo class passed with read() provides the source timestamp, one for each node in the request. An example of use is below:

```
with factory.getClient(uri,DataAccessSync,mal.rr.qos.ReplyTime(THREE_SECONDS),
↪mal.MalProperties()) as da:
   try:
         sample = da.getMal().createDataEntity(dataEntity)
         sample_info=SampleInfo()
         da.read(sample, sample_info)
         log.info('READ %s info nodes' % (sample_info.GetNodeCount()))
         x=0
         while x < sample_info.GetNodeCount():
            log.info('Timestamp[%s]=%s' % (x,sample_info.GetTimestampForNode(x)))
            x += 1
   except:
         raise
```

The OPC UA MAL library provides a predefined set of structures that hold a value of the supported types (see design document). User defined structures must be defined using ICD and their OPC UA specific structures generated by the icd-gen tooling.

In addition, MAL Publisher interface is also implemented as DA write operation, i.e. publish call actually calls DA write and is instantiated with the same URI semantic.

Logging

Additional info for opcua logging:

Open62541 library (client part) will log events via log4cplus malOpcuaOpen62541 logger. If a user does not provide a log4cplus configuration with the malOpcuaOpen62541 logger or a root logger, no log output will be generated from the open62541 library. Bellow is an example configuration pertaining to the open62541 library logging: malOpcuaOpen62541 logger configured for only ERROR and FATAL logging with additivity set to false (events won't be propagated to the root logger). With this configuration all other mal loggers will log with TRACE level. User needs to provide the location of the log4cplus configuration for mal-opcua using mal property opc.log4cplus.filename (see table 3).

log4cplus.rootLogger=TRACE, stdout
log4cplus.logger.malOpcuaOpen62541=ERROR, stdout
log4cplus.additivity.malOpcuaOpen62541=false

log4cplus.appender.stdout=log4cplus::ConsoleAppender

log4cplus.appender.stdout.layout=log4cplus::PatternLayout
log4cplus.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

4.3.4 MUDPI

URIs

The underlying middleware is packet-oriented, packets are addressed via host-port. The Publisher needs to know the address of the subscriber. The Publisher binds to a random local port. The publisher sends unicast or multicast depending on the address encoded in the URI (multicast typically: 224.0.0.0 - 239.255.255.255). In case of unicast, the subscriber must (obviously) run on the host described the URI. In case of multicast, the publisher can choose the outgoing_interface by passing a property to *getPublisher(. . .)*, see section Mal Specific Properties below.

- Publisher: *mudpi.ps://SUBSCRIBER-HOST:SUBSCRIBER-PORT/MUDPI-TOPIC-NAME*
- Subscriber: same as Publisher
- Requester: *mudpi.rr://SERVER-HOST:SERVER-PORT/MUDPI-SERVER-IF-NAME*

Examples of URIs

- *mudpi.ps:///127.0.0.1:12781/TELIF* - unicast on loopback interface
- *mudpi.ps://224.0.0.1:12000/TELIF* - multicast in subnet
- *mudpi.ps://134.171.3.160:12000/TELIF* - unicast to host 134.171.3.160

Mal Specific Properties

The following table enumerates mapping specific properties:

Name	Type	Description	Used by	Default
mudpi.ps. interfaceName	string	Broadcast interface name for mudpi sender Calling setInterface	Publisher	not set (empty string)

4.4 Java

4.4.1 DDS

Mal Specific Properties

For mapping specific properties refer to Table 1 C++ DDS MAL mapping specific properties.

4.4.2 ZPB

Mal Specific Properties

For mapping specific properties refer to Table 2 C++ ZPB MAL mapping specific properties.

4.4.3 OPC UA

Mal Specific Properties

The following table enumerates mapping specific properties:

Name	Type	Description	Used by	Default
opc.schedulEdExecutorThreads	integer	Number of thread pool threads to handle internal timer based tasks (e.g. QoS deadline timeouts).	MAL	3
opc.ps.pollingPeriodMs	integer	Multi-node subscriber polling period in milliseconds.	Subscriber, MrvSubscriber	1000
opc.ps.reqPubRateMs	integer	OPC UA stack subscriptio n requested publishing interval in millisecond s.	MAL	1000
opc.ps.queu eSize	integer	OPC UA stack subscriptio n queue size.	MAL	3

Table 4 Java OPC UA MAL mapping specific properties

4.5 Python

Python MAL API builds on MAL C++ API foundation. There is no native Python MAL mapping implementation, everything is provided via language mapping to C++.

Python applications use Python MAL API only and are not aware of the actual MAL implementation details.

Refer to Python Bindings Design [3] and MAL API User Manual [4] documents for detailed information.

CHAPTER

FIVE

MAL PYTHON MAPPINGS

Document ID:	
Revision:	1.1
Last modification:	March 18, 2024
Status:	Released
Repository:	https://gitlab.eso.org/cii/srv/cii-srv
File:	mal_pythonmappings.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
0.1	2018-07-25	msekoranja		Created.
1.0	2018-08-10	msekoranja		Minor reviewer changes, released.
1.1	18.03.2024	mschilli	0	Public doc

Confidentiality

This document is classified as Public.

Scope

This document is a manual for the Middleware python bindings of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

Glossary of Terms

AMI	Asynchronous Method Invocation
API	Application Programmers Interface
CII	Core Integration Infrastructure
DDS	Data Distribution Service
GC	Garbage Collector
GIL	Global Interpreter Lock
MAL	Middleware Abstraction Layer
QoS	Quality Of Service
STL	Standard Template Library
URI	Uniform Resource Identifier
ZPB	ZeroMQ and Protocol Buffers

References

1. ESO, E-ELT Control System Architecture Concepts, ESO-283831_1
1. Cosylab, MAL ZPB Design Document, CSL-DOC-18-150609, version 1.0
2. Cosylab, MAL DDS Design Document, CSL-DOC-18-150607, version 1.0
3. https://pybind11.readthedocs.io/

5.1 Introduction

This document provides information on use of Python MAL mappings interface also referred to as Python MAL API.

Python MAL API builds on MAL C++ API foundation. It provides programming interface that is very similar to the programming interface that is exposed by MAL C++ API.

Python MAL API is compatible with Python 3.5 as provided with ESO E-ELT development environment v2.1.8.

Python applications use Python MAL API and are not aware of the actual MAL mappings details. The **CiiFactory** interface is the primary means to access MAL functionality.

5.2 Prerequisites

In order to use Python MAL API the following prerequisites must be fulfilled:

- ESO E-ELT Development Environment v2.1.8 must be properly set up.
- Directory hierarchy with installed MAL products must be established (commonly referred to as DATAROOT).
- MAL and MAL COMMON products must be installed under DATAROOT.
- Additional C++ MAL mappings must be installed (for example MAL DDS) under DATAROOT.
- Interface modules for specific data types, generated by ICD GENERATOR or created manually must be installed under DATAROOT.
- Linker path for dynamic loading of libraries must be properly set. On Linux this means that LD_LIBRARY_PATH environment variable should contain reference to the **lib64** subdirectory under DATAROOT. In more general terms, linker path must be setup in such a way that shared libraries required by Python modules and MAL mapping in use are within that path.
- PYTHONPATH environment variable must point to directory **lib/python3.5/site-packages** directory within DATAROOT hierarchy, for example on Linux:

```
# when using bash
```

```
::
```

Define DATAROOT

```
export DATAROOT=~/INTROOT
```

```
::
```

Do other stuff ...

```
::
```

Define LD_LIBRARY_PATH

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$DATAROOT/lib64
```

```
::
```

Define PYTHONPATH

```
export PYTHONPATH=$PYTHONPATH:$DATAROOT/lib/python3.5/site-packages
```

5.3 Using Python MAL API

This chapter provides usage information for Python MAL API.

Python MAL API is implemented as language binding to C++ MAL mapping therefore it closely follows C++ API in its resemblance and operation.

5.3.1 Importing top level Python module, obtaining CiiFactory reference

Top level Python MAL API module is **elt.pymal**. It is recommended to be imported with alias import to mal.

Once module is imported, reference to CiiFactory can be obtained with call to **mal.CiiFactory.getInstance**. This call returns reference to the shared CiiFactory instance.

Alternatively, **malCiiFactory.createInstance** call creates new instance of the CiiFactory.

The following example provides basic steps:

```
#!/usr/bin/env python

# import top level Python MAL API module
import elt.pymal as mal

# use mal interface to obtain regerence to shared CiiFactory instance
ciiFactory = mal.CiiFactory.getInstance()
```

elt.pymal module provides the following interfaces (classes or methods):

- CiiFactory, CII Factory class interface
- Exceptions, MalException, TimeoutException, SchemeNotSupportedException, IllegalUriException, IllegalStateExceptions
- Uri, wrapper for C++ ::elt::mal::Uri class implementation. Not needed by general applications, as methods within Python API that require uri as one of the parameters take a string as input.
- loadMal, method for loading MAL mappings
- PsQoSList, python class helper for creating list of Publish-Subscribe QoS options required by Publisher-Subscriber interface.
- RrQoSList, python helper for creating list of Request-Reply QoS options, required by Request-Reply interface.

5.3.2 Loading and registering MAL mappings

When imported, Python MAL API does not load any MAL mapping by default. Application must load required MAL mappings dynamically, by using **loadMal** method. **loadMal** method takes two required arguments:

- **malName**, string, name of the MAL mapping. Supported values are:
  - dds, for DDS MAL
  - zpb, for ZPB MAL
  - opcua, for OPCUA MAL
- **malProperties**, python dictionary with properties, used to initialize MAL mapping.

loadMal method returns opaque handle to the MAL mapping. In case MAL mapping could not be loaded, exception will be raised.

Opaque handle to the loaded MAL mapping must then be passed to the **CiiFactory.registerMal** class method as second argument. First argument to this method is name of the URI scheme that this particular MAL mapping covers.

Example that follows, illustrates loading and registering two separate MAL mappings:

```
# !/usr/bin/env python

# import top level python MAL API module
import elt.pymal as mal

# obtain reference to CiiFactory
ciiFactory = mal.CiiFactory.getInstance()

# Prepare MAL properties. Note that different MAL mappings
# require specific properties to be initialy set. Refer to
# documentation for specific MAL mapping for details.
malProperties = {}

# Dynamically load DDS MAL mapping
ddsmal = mal.loadMal('dds',  malProperties)

# Register DDS MAL for two schemes with CiiFactory
ciiFactory.registerMal('dds.ps', ddsmal)
ciiFactory.registerMal('dds.rr', ddsmal)

# Dynamically load ZPB MAL mapping
zpbmal = mal.loadMal('zpb', malProperties)

# Register ZPB MAL with CiiFactory
ciiFactory.registerMAL('zpb.ps', zpbmal)
```

5.3.3 Python MAL API relative time parameters (timeouts/durations)

Many API calls require relative time parameters, mostly for timeouts. Relative time parameters are also used within constructors of QoS parameters.

Python MAL API expects instance of **datetime.timedelta** class when relative time parameter is required. For example:

```
import datetime
# ...
publishTimeout = datetime.timedelta(milliseconds=500)
publisher.publish(sample, publishTimeout)
```

5.3.4 Using Publisher-Subscriber interface

Publisher-Subscriber interface relies on generated or handcrafted mapping module that provides Python interface for specific data entity type. Data entity type is typically represented as Python class within mapping module that must be imported. This Python class is a key to obtaining correct Subscriber or Publisher instances from the CiiFactory. Both **CiiFactory.getSubscriber** and **CiiFactory.getPublisher** methods expect reference to the data entity class (not instance) as second parameter. Both methods take URI as the first argument, properties dictionary as third and list of QoS parameters as fourth argument. References obtained with these methods should be checked to be valid (not None).

Code to obtain reference to the subscriber generally looks like:

```
#!/usr/bin/env python
# import top level python MAL API module
import elt.pymal as mal

# Import data entity class. This usually resides in the interface module that
# is either handcrafted or generated.
# In this case TestSample class is our data entity class. Data entity classes
# must provide specific interface to the rest of the Python bindings,
# so they are usually generated automatically.
# Public interface provided by data entity class is specified by data entity
# definition for generator.

from TestSampleIf import TestSample

# obtain reference to CiiFactory
ciiFactory = mal.CiiFactory.getInstance()

# Prepare MAL properties
malProperties = {}

# load and register MAL with CiiFactory
zpbmal = mal.loadMal('zpb', malProperties)
```

(continues on next page)

(continued from previous page)

```
ciiFactory.registerMal('zpb.ps', malProperties)

# prepare subscriber specific MAL properties
subscriberMalProperties = {}

# obtain reference to Subscriber

subscriber = factory.getSubscriber(
 'zpb.ps://localhost:13455/test', TestSample, mal.ps.qos.DEFAULT,
 subscriberMalProperties)
if subscriber is None:
     throw RuntimeError('could not obtain subscriber')

# Read all available samples
for sample in subscriber.read(0):
     # Use interface specific to data entity type, to get values
     print(sample.getSampleId())
# Close the subscriber
subscriber.close()
```

Code to obtain reference to the publisher is similar:

```
#!/usr/bin/env python

import datetime
# import top level python MAL API module
import elt.pymal as mal

# Import data entity class. This usually resides in the interface module that
# is either handcrafted or generated.
# In this case TestSample class is our data entity class. Data entity classes
# must provide specific interface to the rest of the Python bindings,
# so they are usually generated automatically.
# Public interface provided by data entity class is specified by data entity
# definition for generator.

from TestSampleIf import TestSample

# obtain reference to CiiFactory
ciiFactory = mal.CiiFactory.getInstance()

# Prepare MAL properties
malProperties = {}

# load and register mAL with CiiFactory
zpbmal = mal.loadMal('zpb', malProperties)
ciiFactory.registerMal('zpb.ps', malProperties)

# prepare publisher specific MAL properties
```

(continues on next page)

(continued from previous page)

```
publisherMalProperties = {}

# obtain reference to Publisher

publisher = factory.getPublisher(
 'zpb.ps://localhost:13455/test', TestSample, mal.ps.qos.DEFAULT,
 publisherMalProperties)
if publisher is None:
     throw RuntimeError('Could not obtain publisher')

# Use the publisher to create new data entity
dataEntity = publisher.createDataEntity()

# Use interface specific to data entity type, to set values
dataEntity.setSampleId(100)
dataEntity.setText('test')
# Publish sample
publisher.publish(dataEntity, datetime.timedelta(seconds=1))
# Close the publisher
publisher.close()
```

Using Subscriber/Publisher as a Context Manager

Subscriber and Publisher can be closed explicitly by calling **close** method or implicitly when they are destroyed by Python garbage collector. Alternative is to use them as Context Managers. In this case, underlying objects are automatically closed as soon as containing context is destroyed.

See the following example for illustration:

```
# ...setup...

uri = 'zpb.ps:/localhost:13455/test'
properties = {}
qos = mal.ps.qos.DEFAULT
# Use subscriber as Context Maanager
with ciiFactory.getSubscriber(uri, TestSample, qos, properties) as subscriber:
     samples = subscriber.read(0)
# subscriber was automatically closed at the end of previous context
```

Constructing list of QoS parameters

Construction of Subscriber or Publisher object requires list of QoS parameters. Due to underlying implementation this list cannot be simple Python list.

In case list contains no elements, predefined symbol **mal.ps.qos.DEFAULT** should be used.

To construct non empty list, special helper class **mal.PsQoSList** has to be used. Constructor to this class takes either individual QoS parameters or Python list with individual QoS parameters:

```
import elt.pymal as mal

# ...

# Individual QoS parameters
deadline = mal.ps.qos.Deadline(datetime.timedelta(0, 30))
latency = mal.ps.qos.Latency(datetime.timedelta(0, 20))

# Construct QoS list from individual elements
qosList1 = mal.PsQoSlist(deadline, latency)

# Construct QoS list from Python list
lst = [deadline, latency]
qosList2 = mal.PsQoSlist(lst)
```

Constructing Data Event filters

Data Event filters are related to type of data entity that is to be filtered. Filter must be constructed for specific data entity class (not instance). To construct specific event filter, class methods **mal.ps.DataEventFilter.new** or **mal.ps.DataEventFilter.all** must be used with data entity class as argument as in following example:

```
import elt.pymal as mal

# Import data entity class definition
from TestSampleIf import TestSample

# ...
# Obtain reference to CiiFactory
ciiFactory = mal.CiiFactory.getInstance()
# Load and register MAL implementation to be used
# ...

# Create a subscriber
subscriber = factory.getSubscriber(
 'zpb.ps://localhost:13455/test', TestSample, mal.ps.qos.DEFAULT, {})

# create instance for filtering
```

(continues on next page)

(continued from previous page)

```
filterInstance = subscriber.createDataEntity()

# set parameters in filter instance, use API supported by specific data entity

# Create Data event filter. Default filter performs no filtering.
# Parameter is data entity class for which the filter is created.

eventFilter = mal.ps.DataEventFilter.new(TestSample)

# Set filter instance
eventFilter.setFilterInstance(filterInstance)

# Set filter events, input parameter is Python set of events
eventFilter.setInstanceEvents(set(mal.ps.InstanceEvent.UPDATED,
  mal.ps.InstanceEvent.REMOVED))

# use event filter...
```

Both methods return default filter for that data entity.

5.3.5 Using Request-Reply interface

As with Publisher-Subscriber, Request-Reply interface also relies on generated or handcrafted binding module that provides Python interface for specific Request-Reply client. Client can be implemented as either synchronous or asynchronous. Request-Reply client is typically represented as Python class within binding module that must be imported. This python class is key to obtaining correct Client instance from the CiiFactory. Method **CiiFactory.getClient** expects reference to the data Request-Reply client class (not instance) as second parameter. Method takes URI as the first argument, list of QoS parameters as third argument and dictionary of properties as the fourth argument.

The following example illustrates the concept:

```
#!/usr/bin/env python

# Import top level Python API module
import elt.pymal as mal

# from mapping module implementing specific Request-Reply interface,
# import client class
from MathModule import MathSyncClient

# obtain reference to CiiFactory, load MAL implementaton and register MAL with
# CiiFactory
ciiFactory = mal.CiiFactory.getInstance()
# ...

# Obtain synchronous client for Math RR interface
```

(continues on next page)

(continued from previous page)

```
client = factory.getClient(uri, MathSyncClient, mal.rr.qos.DEFAULT, {})

# issue explicit connect request. This is asynchronous, therefore
# we must wait on future returned from connect() method.

client.connect().get()

if client:
    # issue explicit connect request. This is asynchronous, therefore
    # we must wait on future returned from connect() method.

    client.connect().get()

    # This is synchronous client, client will wait for result to arrive
    sum = client.sum(10.1, 15.10)
    print(sum)
    # once done, close the client
    client.close()
```

Using Request-Reply client as Context Manager

Request-Reply clients also support Context Manager protocol to allow for closing client automatically on context exit.

```
# setup...
#
#
qos = mal.rr.qos.DEFAULT

with ciiFactory.getClient(uri, MathSyncClient, qos, {}) as client:
     client.connect().get()
     sum = client.sum(10.1, 15.10)

# Client was automatically closed by now
```

Constructing list of QoS parameters

Construction of Client object requires list of QoS parameters. Due to underlying implementation this list cannot be simple Python list.

In case list contains no elements, predefined symbol **mal.rr.qos.DEFAULT** should be used.

To construct non empty list, special helper class **mal.RrQoSList** has to be used. Constructor to this class takes either individual QoS parameters or Python list with individual QoS parameters:

```python
import elt.pymal as mal

# ...

# Individual QoS parameters
ct = mal.rr.qos.ConnectionTime(datetime.timedelta(0, 5))
rt = mal.rr.qos.ReplyTime(datetime.timedelta(0, 1))

# Construct QoS list from individual elements
qosList1 = mal.RsQoSlist(ct, rt)

# Construct QoS list from Python list
lst = [ct, rt]
qosList2 = mal.RrQoSlist(lst)
```

CHAPTER

SIX

CONFIGURATION

Document ID:	
Revision:	1.1
Last modification:	March 18, 2024
Status:	Released
Repository:	https://gitlab.eso.org/cii/info/cii-docs
File:	config-ng.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document history

Revision	Date	Changed/ Reviewed	Section(s)	Modification
0.8	15.03.2022	Jure Repinc Marcus Schilling	All	Document created / reviewed
0.9	30.10.2023	Jure Repinc	6.4.2	Document cach ing
1.0	06.12.2023	Jure Repinc Marcus Schilling	6.4.2 6.8 6.10	root file provider, list entries
1.1	18.03.2024	Marcus Schilling	0	Public doc

Confidentiality

This document is classified as Public.

Scope

This document is a manual for the Configuration system of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

Glossary of Terms

API	Application Programming Interface
CII	Core Integration Infrastructure
MDI	Metadata instance
YAML	YAML Ain't Markup Language

6.1 Overview

This is the user manual for the CII Configuration API (Config-ng API). All examples in this manual are also present in the cii-demo repository.

The Config-ng API is provided for C++ and Python. It provides support for parsing YAML documents (with support for additional information or directives via tags) into configuration documents. A Configuration document contains set of named instances with optional data type and metadata information.

A Configuration document allows access to its instances and supports additional operations like saving, merging with another configuration document and updating with programmatically prepared instance map.

6.2 Includes and imports

6.2.1 C++

To use the config-ng API from C++, the following directives are needed:

wscript (project)

```
cnf.check_wdep(wdep_name='config-ng.cpp.config-ng', uselib_store='config-ng')
# Transitive Dependencies:
cnf.check_cfg(package='log4cplus', uselib_store='log4cplus', args='--cflags --
↪libs')
```

wscript (module)

```
use=[...,
 'config-ng'
 # Transitive Dependencies:
 'log4cplus'
```

source

```
#include <config-ng/ciiConfigApi.hpp>
```

6.2.2 Python

To use the config-ng API from Python, the following directives are needed:

wscript (project)

```
cnf.check_wdep(wdep_name='config-ng.python.config-ng', uselib_store='config-ng')
```

wscript (module)

```
use=[..., 'config-ng' ]
```

source

```
import elt.configng
```

6.3 API Components

API provides the following basic components:

- Client
- Document
- Node
- Data type
- Metadata

Client is the entry point to the API. It provides basic operations like loading document from provided filename/URI, retrieval or modification of **config-ng** search path.

Document represents a configuration object that was either loaded from file/stream or created in memory. Operations like loading produce a new document instance. Document itself offers additional operations (like merging, check, update and save).

Node represents an item (also known as instance) within a configuration object. Generally, it can have a name and can represent a scalar (single value), a list of items or a map (dictionary) of items. Access to nodes is available through the document instance interface desribed later.

Data type describes a data type assigned to the configuration instance (Node) if any. Data type of instance is optional. There is a set of built-in data types. Defining custom data types is supported by using a predefined yaml syntax. A data type definition can contain its own metadata attributes. Some predefined attributes are understood by the **config-ng** (when running the Check() operation on a document):

default Provides default value of the configuration item when its value is not explicitly provided.

min Allowed minimum value of the scalar configuration item.

max Allowed maximum value of the scalar configuration item.

allowed_values Provides list of allowed scalar values for scalar configuration item.

Metadata is associated with Node and can be seen as a map of attributes assigned to item. These attributes have a name and can be scalars, lists, or maps. There is no data type assigned to the attributes of the item. Some attributes are automatically generated by the **config-ng** at the time of the parsing of the initial yaml document. There are **length** for lists and **rows** and **columns** for matrices.

6.4 Creation of Documents

Documents can be created by parsing YAML source either from input/output stream or provided URI. Additionally documents can also be constructed programatically.

6.4.1 Parsing YAML source from input/output stream

Note that parsing document from YAML can cause exceptions either within **config-ng** or within the underlying YAML parser, therefore proper exception handling is needed (refer to **config-ng** examples).

C++

In C++, YAML source can be provided through a **std::istream** object as demonstrated below:

```
#include <iostream>
#include <sstream>
#include <config-ng/ciiConfigApi.hpp>

int main() {
  const char *document_source = R"DOC(
   a: 10
   b: this is a string
  )DOC";
  std::stringstream document_source_stream;
  document_source_stream << document_source;
  ::elt::configng::CiiConfigDocument document =␣
↪::elt::configng::CiiConfigClient::Load(document_source_stream);
  std::cout << document.Instances()["a"].As<std::int>() << ", "
            << document.Instances()["b"].As<std::string>()
            << std::endl;

  return 0;
}
```

Python

Loading from Python's standard **io** objects is supported:

```
import io
import elt.configng
document_source = """
 a: 10
 b: this is a string
"""
document = elt.configng.CiiConfigClient.load_from_stream(io.StringIO(document_
↪source))
```

(continues on next page)

(continued from previous page)

```
print(document.instances.a.as_value())
print(document.instances.b.as_value())
```

6.4.2 Loading YAML source from file/URI

Documents can be loaded from a local file by specifying a URI. The supported URI syntax is *cii.config://<authority>/<path>* where:

<authority> can be either **local**, or **root** (since CII v4).

<path> is the path to the file.

The "<authority>" component determines the file search strategy, ie. how the "<path>" component gets interpreted. The available file search strategies are implemented by so-called file providers. They are described next.

File-provider: local

The "local" file provider will interpret the path as relative to the search path currently set in the client.

The search path consists of a list of directories to be searched for a specific file. Directories are searched in the order they appear in the list. First match of the filename wins. From a user point of view, the search path is a string containing zero or more directories delimited by : (i.e. `"/path/to/dir1:/path/to/dir2:path/to/dir3"`.

The search path can be set to the empty string. In this case, search for files is limited to only relative to current working directory of the process.

Client initialises the search path from "CFGPATH" environment variable, if defined. To obtain the current search path, it provides method `GetSearchPath()` in C++ and `get_search_path()` in Python. When the search path was initialized from "CFGPATH", the current working directory is prepended to the search path defined in "CFGPATH".

To modify the search path `SetSearchPath(new_path)` is provided in C++ and `set_search_path(new_path)` in Python. The following examples demonstrate manipulation of the search path.

C++ example

```
// save current search path
std::string saved_search_path =␣
↪::elt::configng::CiiConfigClient::GetSearchPath();
std::string my_search_path = "/usr/local/conf:/local/conf";
// establish own search path
::elt::configng::CiiConfigClient::SetSearchPath(my_search_path);
// do something
...
```

(continues on next page)

(continued from previous page)

```
// restore search path
::elt::configng::CiiConfigClient::SetSearchPath(saved_search_path);
```

Python example

```
# save current search path
saved_search_path = elt.configng.CiiConfigClient.get_search_path()
# establish own search path
my_search_path = '/usr/local/conf:/local/conf'
elt.configng.CiiConfigClient.set_search_path(my_search_path)
# do something
...
# restore search path
elt.configng.CiiConfigClient.set_search_path(saved_search_path)
```

File-provider: root

The “root” file provider accesses all files relative to its “document_root” option. Default value of “document_root” option is path ‘/’ (i.e. root of file system). The URI “cii.config://root/etc/passwd” therefore accesses the file “/etc/password”. The “root” file provider does not use the search path (CFGPATH env. variable), its mapping of path to file is 1 to 1, there is no additional steps to determine the actual filename.

To set the “document_root” for the “root” file provider:

```
// C++
::elt::configng::CiiConfigClient::GetFileProviderByName("cii.config://root")->
↪SetOption("document_root", "/absolute/path/to/new/document_root");

# Python
elt.configng.CiiConfigClient.get_file_provider_by_name('cii.config://root').set_
↪option('document_root', '/absolute/path/to/new/document_root')
```

Filenames and relative URIs

As a short-hand alternative to a full URI, it is possible to specify just a path, e.g. “my/path/filename.yaml”.

To convert the path into a URI, the **default file provider** is used, e.g. “my/path/filename.yaml” –> “cii.config://local/my/path/filename.yaml”.

Default file provider

The default file provider is automatically used whenever only a path part of a URI is passed to one of the `::elt::configng::CiiConfigClient::Load` or `::elt::configng::CiiConfigDocument::Save` methods.

The default setting for the default file provider is **local**.

To change the default file provider:

```
// C++
::elt::configng::CiiConfigClient::SetDefaultFileProviderName("cii.config://root
↪");

# Python
elt.configng.CiiConfigClient.set_default_file_provider_name('cii.config://root')
```

The analogous getter methods are not shown here.

Cache setup in C++

Config-ng can cache loaded documents, so that big document reloads can be avoided. The cache is **disabled by default** and must be enabled explicitly trough CiiConfigClient.

```
#include <iostream>
#include <config-ng/ciiConfigApi.hpp>
int main() {
  // Obtain cache state
  bool cache_state = ::elt::configng::CiiConfigClient::GetCacheState();
  if (cache_state == false) {
    // Enable cache
    ::elt::configng::CiiConfigClient::SetCacheState(true);
  }
  ...
}
```

Cache setup in Python

```
import elt.configng

# Obtain cache state
cache_state = elt.configng.CiiConfigClient.get_cache_state()
if not cache_state:
    # Enable cache
    elt.configng.CiiConfigClient.set_cache_state(True)
```

6.4.3 Actual document loading

C++ Document is loaded by calling method `::elt::configng::CiiConfigClient::Load()` as demonstrated below. Note that parsing YAML and loading document can produce exceptions therefore exception handling is needed (refer to examples for more information):

```
// Load path/file.yaml using search path
::elt::configng::CiiConfigDocument document =␣
↪::elt::configng::CiiConfigClient::Load("cii.config://local/path/file.yaml")

// Load path/file.yaml using document root
::elt::configng::CiiConfigDocument document =␣
↪::elt::configng::CiiConfigClient::Load("cii.config://root/path/file.yaml")
```

In the same manner as C++, Python version provides method `elt.configng.CiiConfigClient.load()`:

```
# Load path/file.yaml usign current search path
::elt::configng::CiiConfigDocument document = elt.configng.CiiConfigClient.load(
↪'cii.config://local/path/file.yaml')

# Load path/file.yaml usign document root
::elt::configng::CiiConfigDocument document = elt.configng.CiiConfigClient.load(
↪'cii.config://root/path/file.yaml')
```

6.4.4 Building document programmatically

To construct a document programatically, a special helper class **CiiConfigMapBuilder** must be used to construct the 'map' of the instances to be added to document. **CiiConfigMapBuilder** provides a simple interface to add elements to the map. The general workflow is like this:

- Create empty document instance
- Use map builder to prepare an instance map of instances to be added to the document
- Update document with prepared map. With C++, use method `` `SetInstancesFromMap() `` on the document instance. With Python, use method `set_instances_from_map()` on the document instance.

Note that on the API level there are slight differences between Python and C++ API (due of language differences), but the general workflow is the same.

C++ Example

```cpp
// See programmatic example for more comprehensive information
#include <vector>
#include <string>
#include <iostream>
#include <config-ng/ciiConfigApi.hpp>
// NOTE: additional includes needed
#include <config-ng/ciiConfigMapBuilder.hpp>
#include <config-ng/ciiConfigValueConverter.hpp>

int main() {
   // prepare empty document
   ::elt::configng::CiiConfigDocument document =␣
↪::elt::configng::CiiConfigDocument();
   // Create CiiConfigMapBuilder
   ::elt::configng::CiiConfigMapBuilder builder;
   // Builder provides operators to simplify construction of node hirearchy
   // Note that values cannot directly be assigned to the builder items,
   // factory method ::elt::config::CiiConfigInstanceNode::From() must be used
   // to turn C++ value into correct node to be later assigned to document

   // Create untyped instance a and assign value 3 to it
   builder["a"] = ::elt::configng::CiiConfigInstanceNode::From(3);
   // Create untyped instance b and assign list 1,2,3 to it
   builder["b"] = ::elt::configng::CiiConfigInstanceNode::From(std::vector<int>
↪{1,2,3});
   // Create type instance c, using builtin type vector_int32
   builder["c"] = ::elt::configng::CiiConfigInstanceNode::From(std::vector<int>
↪{1,2,3},
     "!cfg.type:vector_int32");
   // Create map with untyped items a and b
   builder["map"]["a"] = ::elt::configng::CiiConfigInstanceNode::From(3);
   builder["map"]["b"] = ::elt::configng::CiiConfigInstanceNode::From("a string
↪");

   // Once map is constructed, document can be updated
   document.SetInstancesFromMap(builder);
   // Display instance a
   std::cout << document.Instances()["a"].as<int>();
   return 0;
}
```

Python Example

```python
# See programmatic example for more comprehensive information
import elt.configng
# Prepare empty document
```

(continues on next page)

(continued from previous page)

```
document = elt.configng.CiiConfigDocument()
# Create CiiConfigMapBuilder
builder = elt.configng.CiiConfigMapBuilder()
# Create untyped instance a and assign value 3 to it
builder['a'] = 3
# Create untyped instance b and assign list [1,2,3] to it
builder['b'] = [1,2,3]
# Create typed instance c and assign list [1,2,3] to it.
# Whenever typed instance is required, use of elt.confing.CiiConfigNodeFacory.
↪make_from
# method must be used, as it takes additinal parameters optional tag denoting␣
↪data type
# to be used when creating instance and optional converter method for custom
# value conversion when required
builder['c'] = elt.configng.CiiConfigNodeFactory.make_from([1,2,3], '!cfg.
↪type:vector_int32')
# Crreate map with untyped items a and b
builder['map']['a'] = 3
builder['map']['b'] = 'a string'
# Once map is constructed, document can be updated
document.set_instances_from_map(builder)
# Display instance a
print(document.instances.a.as_value())
```

6.5 Access to YAML source tree of a document

Once a document is loaded from YAML stream or file, the original YAML node tree is available to the user trough `CiiConfigDocument::GetYamlSourceRootNode()` in C++ and `CiiConfigDocument.get_yaml_source_root_node()` in Python.

To operate on the returned YAML root node in C++, **yaml-cpp** API must be used.

To operate on the returned YAML root node in Python, **PyYAML** API must be used.

Note that subsequent changes on the returned YAML node tree do not reflect on the config document, and vice versa, subsequent changes on the config document do not reflect on the YAML node tree. It is recommended that when dealing directly with the YAML node tree, the best course is to save the modified tree as YAML into file or stream, and then load it into another config document. The sample, analogously, applies to changes on the config document: save it to YAML stream/file, and then load it directly with a YAML parser.

6.6 Document instance interface

Once the document is created or loaded, access to its instances is available via the document instance

name Name of the instance (only valid with top level and Map member instances)

type Three user visible types are predefined: Scalar, Sequence, Map.

metadata Metadata atrributes of the instance.

data type Optional config ng data type associated with the instance

origin Source file location (line, column) of the YAML element from which the instance has been constructed.

Sequence and Map instances provide non recursive (by default) and recursive iterators and methods to access the instance value.

6.6.1 Accessing document instance interface

C++

The document instance interface is available through method ::elt::configng::CiiConfigDocument::Instances() method. i.e.:

```
::elt::configng::CiiConfigDocument document = ::elt::configng.
↪CiiConfigClient::Load("cii.config://local/file.yaml");
// For reading only
const ::elt::configng::CiiConfigInstanceNamespace &instances = document.
↪Instances();
```

Access to a specific instance is done through operator[]:

```
// Target type is known, instance value will be automatically
// converted to target type (note exception is thrown if conversion not possible)
double d = instances["theDouble"];
// Target type is not known, conversion method As() must be used
std::cout << instances["theDouble"].As<double>() << std::endl;
// Assign new value to the instance
instances["theDouble"].Assign(6.7);
```

Obtaining additional information:

```
::elt::configng::CiiConfigInstanceNamespace &theDouble = document.Instances()[
↪"theDouble"];
// Check the type
if (theDouble.IsScalar()) {
   ...
```

(continues on next page)

(continued from previous page)

```
} else if (theDouble.IsSequence()) {
  // Tterate non recursively over instance
  for (const auto &item: instance) {
    ...
  }
  ...
} else if (theDouble.IsMap()) {
  ...
}
// Obtain the origin
::elt::configng::CiiConfigItemOrigin origin = theDouble.GetOrigin();
// Display the origin
std::cout << "Filename: " << origin.source << " line: " << origin.line << "␣
↪column: "
          << origin.column << std::endl;
// obtain config-ng data type of the instance
::elt::configng::CiiConfigDataType data_type = theDouble.GetDataType();
// access metadata of the instance
::elt::configng::CiiConfigMetadata &metadata = theDouble.GetMetadata();
// and metadata properties
if (metadata.Has("max")) {
   std::cout << "theDouble has max property: " << metadata["max"].As<double>()
             << std::endl;
}
```

Python

The document instance interface is available through method elt.config.CiiConfigDocument.get_instances(), or through the attribute accessor **instances** on the document object, i.e.:

```
document = elt.config.CiiConfigClient.load('cii.config://local/file.yaml')
# Use attribute accessor instances to access instance 'theDouble'
theDouble = document.instances.theDouble
# Or the classic way
instances = document.get_instances()
theDouble = instances['theDouble']
# Print value of the instance
print(theDouble.get_value())
# Alternative to obtain the value
# is to use as_value() method which takes optional converter argument
# that is actually a function to call when instance value needs to be
# converted to python value
print(theDouble.as_value(lambda x: x.get_value() + 1))
# Assign new value to the instance
theDouble.assign(6.7)
```

Obtaining additional information:

```
theDouble = document.instances.theDouble
# Check the type
if theDouble.is_scalar():
   ...
elif theDouble.is_sequence():
  # Iterate non recursively over instance
  for item in theDouble:
      print(item.get_value())
elif theDouble.is_map():
  ...
# Obtain origin of the instance
origin = theDouble.get_origin()
# Display the origin, get_info() method returns tuple (filename, line, column)
# indicating origin of the instance
print('Origin filename %s, line: %s, column: %s' % origin.get_info())
# Obtain config-ng data type of the instance
data_type = theDouble.get_data_type()
# When data type is not assigned to the instance, None is returned
if data_type is None:
   print('Instance theDouble is untyped')
else:
   print('Data type of the theDouble instance is ', data_type.get_name())
# Access metadata of the instance
metadata = theDouble.get_metadata()
# And metadata properties ...
if metadata.has('max'):
   print('theDouble has max property: ', metadata['max'].get_value())
```

6.6.2 Iteration over document instances

The object returned by the document instance interface offers a recursive iterator over all document instances by default.

C++

```
const ::elt::configng::CiiConfigInstanceNamespace &instances = document.
↪Instances();
for (const auto &[name, node]: instances) {
    std::cout << "Name: " << name << ", Origin: " << node.GetOrigin() <<␣
↪std::cerr;
    // Obtain associated metadata
    ::elt::configng::CiiConfigMetadata &metadata = node.GetMetadata();
    // Obtain data type of the instance node from metadata
    ::elt::configng::CiiConfigDataType &data_type = metadata.GetDataType();
    std::cout << " ... data type: " << data_type.GetName()
              << " originating: " << data_type.GetOrigin() << std::endl;
```

(continues on next page)

(continued from previous page)

```
    std::cout << " ... basic data type: "
              << elt::configng::util::ToString(data_type.GetBasicDataType()) <<␣
↪std::endl;
   if (data_type.IsVector() || data_type.IsMatrix()) {
      std::cout <<  ".... element data type: " << data_type.GetElementDataType().
↪GetName();
   }
 }
```

Python

```
document = elt.configng.CiiConfigLoad('filename.yaml')
for name, item in document.instances:
    # Data type can be None, indicating that config-ng data type is not assigned
    # to the instance
    data_type = item.get_data_type()
    print(name, ' Data type: ', data_type)
    print(name, ' Originating: ', item.get_origin()

    if data_type is not None:
        print('Basic data type: ', data_type.get_basic_data_type())
        if data_type.is_matrix() or item.data_type.is_sequence():
            print('..... element data type: ', data_type.get_element_data_type())
```

6.7 Document Validation

Metadata that is produced for each instance node is initially not validated, therefore it might not reflect the actual state of document.

To check the document validity against the schema (defined by elements with CII custom tags) and validity of the metadata, one must invoke check operation on the document. This validates the schema and all values of the document instances.

The Check operation returns an object describing the issues detected during document validation. In case this object does not contain any issues with error severity, the document is valid. This implies that the document is valid when only issues with warning severity were detected.

It is recommended to perform document validation after document loading or any operation that updates document like merge with another document or update with a programmaticaly built map of instances.

6.7.1 C++

The document validation operation is invoked by calling `::elt::configng::CiiConfigDocument::Check` on the document instance.

```
// Load document
::elt::configng::CiiConfigDocument document =␣
↪::elt::configng::CiiConfigClient::Load("cii.config://local/file.yaml");
// Perform document validation
::elt::configng::CiiConfigDocumentIssues issues = document.Check();
// The following test returns true in case there was at least one issue with␣
↪error severity
// was detected
if (issues) {
  std::cerr << "Document is not valid! The following issues were detected: " <<␣
↪std::endl;
  // One can iterate over issues. Each issue is instance␣
↪of::elt::configng::CiiConfigDocumentIssue
  // class.
  for (auto &issue: issues) {
     // Print out the issue
     std::cerr << " " << issue << std::endl;
     // Or examine it
     if (issue.IsError()) {
        std::cerr << "This is issue with code " << static_cast<int>(issue.
↪GetCode())
                  << " and message: " << issue.GetMessage() << std::endl;
     }

} else {
  // Document is valid at this point, but issues with warning severity can still␣
↪be present
```

(continues on next page)

(continued from previous page)

```
  std::cout << "Document is valid." << std::endl;
  if (issues.HasWarnings()) {
      std::cout << "The following warnings were issued during validation: " <<␣
↪std::endl;
      for (auto &issue: issues) {
        std::cout << " " << issue << std::endl;
   }
}
```

6.7.2 Python

Calling method check() on an elt.configng.CiiConfigDocument instance invokes the document validation operation.

```
# Load document
document = elt.configng.CiiConfigClient.load('cii.config://local/file.yaml')
# Perform document validation
issues = document.check()
if issues.has_errors():
    print('Document is not valid! The following issues were detected:')
    for issue in issues:
        # Print the issue
        print(' ', issue)
        # Or examine it
        if issue.is_error():
        print('This is issue with code ', issue.get_code(), ' with message: ',
            issue.get_message())
else:
    # Document is valid, but issues with warning severity can still be present
    print('Document is valid.')
    if issues.has_warnings():
        print('The following warnings were issued during validation:')
        for issue in issues:
            print(' ', issue)
```

6.8 Saving documents to YAML

A document can be emitted in YAML format to local files or streams. It is recommended that the documennt is validated before emitting it in YAML format. The Save operation supports additional options that can prevent overwriting existing file or inlining includes (meaning not generating !cfg.include directives in the emitted YAML document).

As of CII v4, `::elt::configng::CiiConfigDocument::Save (elt.configng.CiiConfigDocument.save)` method also supports a URI as an argument. If a filename is specified, the default file provider is used to save the file (RootFileProvider in path relative to its "document_root", LocalFileProvider relative to current working directory (absolute paths are forbidden with LocalFileProvider)).

6.8.1 C++

```
// Load document
::elt::configng::CiiConfigDocument document =␣
↪::elt::configng::CiiConfigClient::Load('cii.config://local/file.yaml');
// Validate document
::elt::configng::CiiConfigDocumentIssues issues = document.Check();
// Only save valid document
if (!issues) {
    // Save to file. Existing files are overwriten by default. Includes are not␣
↪inlined
    // Note that argument here is acutally file name not URI.
    document.Save('file1.yaml');
    // Save to file but prevent overwriting existing file.
    // In case file already exists, ::elt::configng::CiiConfigExistError is␣
↪thrown
    try {
        document.Save('file1.yaml,
            {::elt::configng::CiiConfigDocument::SaveOptions::NO_OVERWRITE});
    } catch (const ::elt::configng::CiiConfigExistsError &e) {
        std::cerr << "Could not overwrite the file (" << e.what() << ")" <<␣
↪std::endl;
    }
    // Save to stream, inline includes
    std::ostringstream stream;
    document.Save(stream,
        {::elt::configng::CiiConfigDocument::SaveOptions::INLINE_INCLUDES});
    // Save to stream, generate !cfg.include directives for included files
    std::ostringstream another_stream;
    document.Save(another_stream);
}
```

6.8.2 Python

```
# Load document
document = elt.configng.CiiConfigClient.load('cii.config://local/file.yaml');
# Validate document
issues = document.check()
# Only save valid document
if not issues.has_errors():
    # Save to file. Existing files are overwritten by default. Includes are not␣
↪inlined.
    # Note that argument here is actually file name not URI.
    document.save('file1.yaml')
    # Save to file but prevent overwriting existing file.
    # In case file already exists, elt.configng.CiiConfigExistsError is raised.
    try:
        document.save('file1.yaml',
                      (elt.configng.CiiConfigDocument.SaveOptions.NO_OVERWRITE,))
    except elt.configng.CiiConfigEistsError as e:
       print('Could not overwrite the file %s' % e)
    # Save to stream, inline includes
    stream = io.StringIO()
    document.Save(stream,
                  (elt.configng.CiiConfigDocument.SaveOptions.INLINE_INCLUDES,))
    # Save to stream, generate !cfg.include directives for included files
    another_stream = io.StringIO()
    document.Save(another_stream)
```

6.9 Merging documents

One can request a merge of two documents via the merge operation. This operation modifies the target document in place.

The Merge operation does not modify target document in case it detects any issue that would prevent the merge from fully succeeding, unless explicitly permitted by the user.

It is recommended to validate both documents before attempting to execute merge operation. For clarity, validations are not performed in the following examples.

6.9.1 C++

```
// Load target document.
::elt::configng::CiiConfigDocument document =␣
↪::elt::configng::CiiConfigClient::Load("cii.config://local/document.yaml");
// Load document to merge
::elt::configng::CiiConfigDocument document_to_merge =␣
↪::elt::configng::CiiConfigClient::Load(
  "cii.config://local/document_to_merge.yaml");
// Try merge operation. List of issues that prevented merge is returned. In case␣
↪returned
// list is empty, merge succedded.
std::vector<::elt::configng::CiiConfigDocument::MergeIssue> merge_issues =␣
↪document.Merge(document_to_merge);
if (merge_issues.empty()) {
   std::cout << "Merge succeeded" << std::endl;
} else {
   std::cerr << "Merge failed, the following issues were detected:" << std::endl;
   for (const auto &issue: merge_issues) {
      std::cerr << " " << issue << std::endl;
   }
   // Target document was not updated.
   // Perform partial merge, only update those instances that can be safely␣
↪merged.
   // Partial merge is requested by supplying additinal boolean argument with␣
↪true value.
   // Merge issues for instances that could not be merged are still returned
   merge_issues = document.Merge(document_to_merge, true)
}
```

6.9.2 Python

```
# Load target document
document = elt.configng.CiiConfigClient.load('cii.config://local/document.yaml')
# Load document to merge
document_to_merge = elt.configng.CiiConfigClient.load(
    'cii.config://local/document_to_merge.yaml')
# Try merge operation. List of issues that prevented merge is returned. In case␣
↪returned
# list is empty, merge succeeded.
merge_issues = document.merge(document_to_merge)
if not merge_issues:
    print('Merge succeeded')
else:
    print('Merge failed, the following issues were detected:')
    for issue in merge_issues:
        print(' ', issue)
    # Target document was not updated.
    # Perform partial merge, only update thos unstances that can be safely␣
↪merged.
    # Partial merge is requested by supplying additional bool argument with True␣
↪value.
    # Merge issues for instances that could not be merged are still returned
    merge_issues = document.merge(document_to_merge, True)
```

6.10 Listing documents

As of CII v4, Config client provides a method to list existing documents.

Note: this method is only supported with RootFileProvider as there is no meaningful implementation for LocalFileProvider.

```
// C++
::elt::configng::CiiConfigClient::ListDirectoryEntries(const std::string &uri)

# Python
elt.configng.CiiConfigClient.list_directory_entries(uri: str)
```

The method returns list of pairs (python: tuple) each containing URI of the entry and boolean indicator whether that entry is a directory (true/false, python: True/False).

6.11 Exceptions

Config-ng API calls can produce a number of exceptions. For details please refer to the API docs.

All exceptions generated by the config-ng API have common basic class (`::elt::configng::CiiConfigError` in C++ and `elt.configng.CiiConfigError` in Python).

List of config-ng exceptions that are defined in Python and C++:

CiiConfigError Base clas of config-ng exceptions.

CiiConfigBuildError Generated during process of building configuration document when major inconsistency is detected.

CiiConfigTypeError Indicates that argument with wrong data type was used.

CiiConfigValueError Indicates that argument with wrong value was used.

CiiConfigNotFoundError Item was not found

CiiConfigExistsError Item or file already exists.

CiiConfigNotImplementedError Feature was not implemented yet.

CiiConfigIterationError Inconsistency detected during iteration.

CiiConfigIllegalUriError Illegal URI was used.

An additional exception is defined in C++ implementation:

CiiConfigDocumentNotFromFileError Attempted to save document that was not loaded from file without providing file name.

6.12 YAML tags recognized by config-ng

Config-ng uses several specific tags in different contest. All tags recognized by config-ng have prefix **cfg**.

6.12.1 cfg.include

Request inclusion of another YAML file.

Syntax: !cfg.include <URI>: [NOERROR]

Where <URI> is the URI or the path of the file to include in the document. If <path> is used, the file is looked up according to the rules of the default file provider.

Optional NOERROR flag indicates that in case the file is not found, the operation should continue. Without this flag, an exception is thrown when the requested file could not be found.

Example:

```
# Include basic definitions
!cfg.include cii.config://local/basic_definitions.yaml:
# Include additional extensions, but do not fail if the file was not found
!cfg.include cii.config://local/extensions: NOERROR
```

6.12.2 cfg.type

Reference data type.

Syntax: !cfg.type:<TYPENAME>

<TYPENAME> is the name of a built-in data type or user-defined data type.

Example:

```
# Untyped instance
a: 10
# Typed instance, in this case b is of built in type int32
b: !cfg.type:int32 10
```

6.12.3 cfg.typedef

Define type alias or custom data type.

Syntax: !cfg.typedef <TYPENAME>[(<BASE>)]

<TYPENAME> is the name of the new data type. <BASE> is optional and must be the name of an existing built-in or user-defined data type. When specified, config-ng derives a new data type from the provided base data type.

Example:

```
# Type alias
!cfg.typedef myint(int32):

# Type alias with additional metadata
!cfg.typedef extint(int32):
    min: 10
    max: 100
    default: 25

# User defined data type (like struct)
!cfg.typedef Point:
    x: !cfg.type:double
    y: !cfg.type:double


# Derived, inherits the members of the basic data type,
# in this case x and y. Adds new member z.
!cfg.typedef Point3D(Point):
    z: !cfg.type:double
```

6.12.4 cfg.optional, cfg.required

Modifiers that can be used within user-defined data type member definition.

cfg.optional indicates that the value of this member needs not be present when initializing an instance of that data type.

cfg.required indicates that the value of this member must always be present when initializing an instance of this data type.

When neither of these modifiers are used, the value of the member is initialized from the default value, if not explicitly specified.

Example:

```
!cfg.typedef Point:
    # x must be always specified
    !cfg.required x: !cfg.type:double
    # y is initialized from default not specified
    y: !cfg.type:double
    # z is optional
    !cfg.optional z: !cfg.type:double

valid_point_1: !cfg.type:Point { x: 10 }
valid_point_2: !cfg.type:Point { x: 10, y: 55 }
valid_point_3: !cfg.type:Point { x: 10, z: 33 }
# The following instance is invalid, x is missing
invalid_point: !cfg.type:Point { y: 3, z: 3 }
```

6.13 List of built-in data types

List of built-in data types recognized by config-ng:

- int8
- uint8
- int16
- uint16
- int32
- uint32
- int64
- uint64
- single
- double
- boolean
- string
- binary
- vector_uint8
- vector_int8
- vector_uint16
- vector_int16
- vector_uint32
- vector_int32
- vector_uint64
- vector_int64
- vector_single
- vector_double
- vector_boolean
- vector_string
- matrix2d_uint8
- matrix2d_int8
- matrix2d_uint16

- matrix2d_int16
- matrix2d_uint32
- matrix2d_int32
- matrix2d_uint64
- matrix2d_int64
- matrix2d_single
- matrix2d_double

CHAPTER

SEVEN

ONLINE DATABASE

Document ID:	
Revision:	1.14
Last modification:	March 18, 2024
Status:	Released
Repository:	https://gitlab.eso.org/cii/info/cii-docs
File:	oldb.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
0.1	05.07.2019	jpri bosek, manovak	5	Document creation. GUI Section.
0.2	19.07.2019	jpri bosek, jstrnisa	1-4 Appendix A Appendix B	Added client API sections.
0.3	05.08.2019	torovic bter-pinc	All	Document review and fixes.
1.0	24.09.2019	jpri bosek	All	Final updates, release.
1.1	20.1.2020	jstrnisa	All	Major update of all sections after ESO topical review
1.2	13.7.2020	bter-pinc	3.1 4.1 5.6	Statement about pub/sub queue adjustment added. Note about the synchronous methods added. Additional External Redis serves explanation added.
1.3	16.7.2020	bter-pinc	7	CLI tools update
1.4	04.4.2022	mschilli	7	Updated cpp py examples
1.5	09.5.2022	dkumar	All	HDFS to Remo te FS, disab le formula c heck when wr iting
1.6	05.07.2023	mschilli	7.6.2.2	Details on timestamp resolution
1.7	21.08.2023	mschilli	7.3 - 7.6	Updates for CII v4 OLDB (LFS,CE)
1.8	24.08.2023	jcapinc	7	Added chapter describing YAML data point type.
1.9	25.08.2023	jcapinc	7.4.1.6	Described Python subscription caveat with GIL (ECII-77 9)
1.10	28.08.2023	mschilli	7.6.2.3.2	Syntax for references in formulas has changed
1.11	25.10.2023	jcapinc	7.5.5 7.5.8 7.6.2.3.4	Data point configuratio n, data poin t aging, isSubscripti on-Notificati only,creatin g data point s with isSub scriptionNot ificationOnl flag
1.12	17.11.2023	dkumar		Subscribe method NOTE added
1.13	28.02.2024	mschilli	4, 7	datapoint aliases ; oldb-cli: short uris, create, delete
1.14	18.03.2024	mschilli	0	Public doc

Confidentiality

This document is classified as Public.

Scope

This document is manual for the Online Database system of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

Glossary of Terms

API	Application Programming Interface
CII	Core Integration Infrastructure
CLI	Command Line Interface
DP	Data Point
GUI	Graphical User Interface
ES	ElasticSearch
JSON	Javascript object notation
OLDB	Online Database
SVN	Subversion
YAML	YAML Ain't Markup Language

References

1. ELT Software Development Homepage https://www.eso.org/projects/elt/develop
2. CII User Manuals - Service Management: https://www.eso.org/~eltmgr/CII/latest/manuals/html/docs/services.html
3. CII User Manuals - Internal Configuration System: https://www.eso.org/~eltmgr/CII/latest/manuals/html/docs/config.html
4. Redis main page: https://redis.io/
5. Redis INFO command https://redis.io/commands/info

7.1 Overview

This document is a user manual for usage of CII OLDB system. It explains how to use the OLDB Client API through Client API library and GUI Application to interact with the CII Online Database. All examples in this manual will be also presented in SVN code (project oldb-examples).

7.2 Introduction

The Core Integration Infrastructure (CII) Online Database (OLDB) provides distributed data publishing and access to actual or live data for user interface and control applications that do not have low-latency or real-time performance requirements. The term “online” refers to the fact that the database provides current and live values for the data points of the control system.

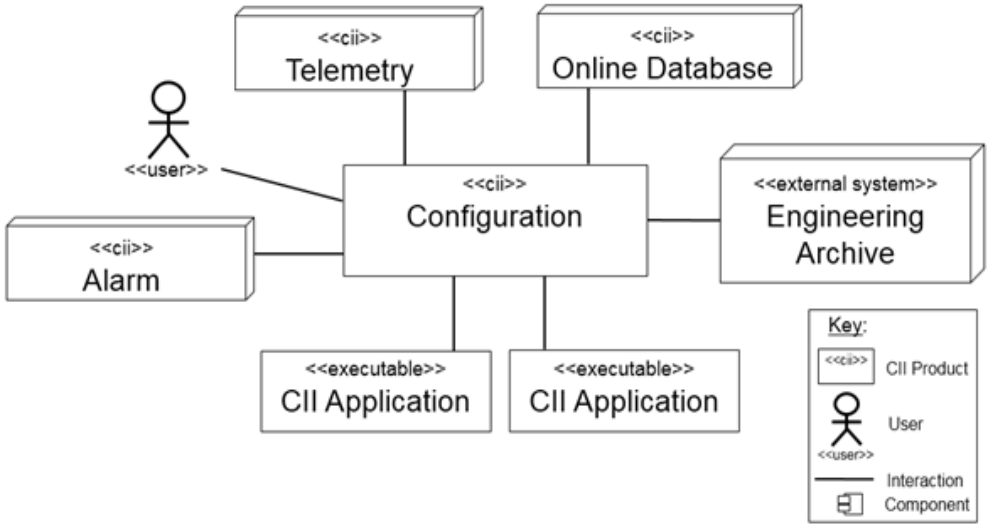


Figure 2-1: OLDB API interactions

A **data point** represents a single value in the OLDB and consists of a value, timestamp, quality, and metadata. Data points are identified by a unique URI (section 6.2.1).

Data point **value** is the actual data the data point is holding. It can be any of the predefined types (Described in Appendix A). The type of the data point can be a primitive type (e.g. integer, double, string) or a more complex type (vector or 2D matrix).

The **timestamp** is the timestamp of the last data point value modification.

The **quality** represents the state of the data point. It can be OK, SUSPECT and BAD. Only OK data points can be read and written to (except when correcting the quality to OK). The meaning of each quality state is as follows:

OK: the value of the Property is valid - this is the default quality for any new data point that is created with a provided value.

BAD: the value is not OK and should not be used - this is the default quality for any new data point that is created without a value (i.e. when a default value is used).

SUSPECT: there is a value, but it is not clear if it is reliable. For example, the readout of a sensor is inconsistent with other readings and the sensor might be faulty.

Metadata is the meta information about the data point and consists of information like DP data type, formula, quality expression and other fields specific to DP data type (for a detailed description of metadata content see section 6.2.3). Metadata is maintained by CII Configuration system [3], but can

also be manipulated through the OLDB Client.

Data points can define a calculation **formula** (included in metadata) that determines the value of the data point. The formula can include URIs of other data points and so the value of the data point can depend on values of other data points. A data point with formula is called **calculated data point** and cannot be written to by users, since its value is determined by the formula.

OLDB **Client API** provides a means to manipulate data points (create, read, write and delete) through the OLDB client. Users can access and manipulate OLDB through the Client API library (srv-oldb) or OLDB GUI and CLI applications. The srv-oldb library is written in C++ and Python. For examples, see Chapter 4 where complete examples are listed in both languages.

OLDB uses Redis key-value store [4] for storing the data. Small data points are stored in Redis as keys, while large DP that exceed the limit defined in the OLDB configuration (section 3.3.1) are split automatically before being saved. The Redis storage is further divided into two parts:

main storage: This is the default location for data points. All small sized data points are stored here.

extra storage: This is a special storage where only predetermined data points (defined in data point configuration) are stored. It is intended for larger values that would affect the performance of the main storage.

Both default and in-memory-only storages are horizontally scalable and can consist of any number of Redis servers either single instance or clusters.

The lifetime of the data points is not bounded by the lifetime of the OLDB client instance because their values and configurations are saved in the OLDB storage and Configuration service respectably. If the Redis storage is configured to be persistent, the data points also persist after Redis server shutdown.

7.3 Prerequisites

This section describes the prerequisites for using the OLDB library and applications. Note that the preparation of the environment (i.e. installation of required modules, configuration) is not in the scope of this document. For information on how to run the services, refer to [2].

7.3.1 WAF Modules

For using the OLDB library and CLI application the following modules must be built and installed with WAF in the specified order:

1. elt-mal
2. client-api
3. srv-config
4. oldb-client
5. srv-oldb

For using the OLDB GUI application in addition to the above two more modules must be built and installed:

6. elt-qt-widgets
7. oldb-gui

7.3.2 Services

Redis

OLDB uses Redis [4] key-value store as its main storage of data point values. At a minimum the OLDB needs at least one main storage server, at least one extra storage (External Redis Server - 5.6) storage and exactly one pub/sub server. One Redis instance can serve as all three servers so minimally one Redis instance must be running for OLDB to work.

The number of default and in-memory-only storage servers is unbounded. Only one Redis server can be designated to be a pub/sub server and can be either a single instance or cluster.

Calculation Engine

For the OLDB framework to work as intended, the **calculation service** (OLDB service) must be running. The calculation service is responsible for calculating the value and quality of calculated data points. The calculation service consists of any number of **calculation node** processes and a **scheduler process**. Each calculation node is responsible for the calculation of a subset of all existing data points and the scheduler process is responsible for distributing the calculated data points among the calculation nodes.

All the calculation nodes must be running before the scheduler process. The scheduler must be configured with addresses of existing calculation nodes so it can connect to them.

To add a calculation node, the scheduler configuration must be updated with the new node's address and the scheduler process must be restarted. The old nodes will retain their calculated data points and continue to recalculate their values. Note that during scheduler downtime newly created calculated data points will not be distributed and calculated. **They will be distributed however when the scheduler process restarts.**

If a calculation node shuts down the subset of calculated data points assigned to it will not be re-assigned to another calculation node and will not be calculated. For this to happen the scheduler process must be restarted.

The proper way to shut down the calculation service is to first stop the scheduler process and then the calculation nodes.

Configuration Service

OLDB uses the internal CII configuration storage [3] to store its configuration. The configuration must be accessible for the OLDB Client and Calculation Service to successfully initialize.

7.3.3 Configuration

Both the OLDB Client and the calculation service must be configured to work properly. Both OLDB and Calculation Service configurations are stored in a versioned way.

OLDB Client Configuration

The OLDB client configuration is stored in the CiiOldbConfigClass object. The configuration is fetched from the CII Configuration Service by the OLDB client at start-up.

The configuration class contains the following fields:

1. redisServers: a MdStringArray class containing a list of strings that contains parameters for **default** storage servers. Each string corresponds to one server and is formatted as <sequential-number>:<hostname>:<port>:<server-type> where server-type is either SINGLE or CLUSTER and describes the type of the server (either single instance or cluster). The <sequential-number> should be the sequential number of servers in the list starting at 0.

2. redisServersExternal: a MdStringArray class containing a list of strings that contains parameters for **in-memory-only** storage servers. Each string corresponds to one server and is formatted as <alias>:<hostname>:<port>:<server-type> where server-type is either SINGLE or CLUSTER and describes the type of the server (either single instance or cluster).
3. pubSubServerInfo: a MdString class containing a string that contains parameters for the OLDB pub/sub server. The string is formatted in the same manner as strings in the redisServers field, except that the <sequential-number> has no significance since there is always one server.
4. **pubSubHandlerProccesingInterval: a MdInt32 containing an integer** that represents the interval in milliseconds at which a pub/sub message handler should handle messages. To prevent the incorrect ordering of command execution, the incoming pub/sub messages are ordered by timestamp in unbounded queues and processed every pubSubHandler-ProccesingInterval milliseconds. The performance of the pub-sub system will degrade with larger pubSubHandlerProccesingInterval value, be but the correct order of message execution will be more guaranteed. The recommended value for this field is 10. The size of the pub/sub event queue is unbounded, and thou it is not adjustable. **Note: This reason for this setting is the proper timestamp ordering of the incoming messages.**
5. **valueSizeLimitRedisExt: a MdInt32 containing an integer that** represents the value size limit in bytes for in-memory-only servers. A data point value written to these servers is sliced and distributed among several key-value pairs so that each chunk does not exceed the valueSizeLimitRedisExt. If this value is too small write operations on large data points will be slower. It should not be greater than 512 Mb (512 * 1024 * 1024 bytes) since this is the limit of one key-value pair on Redis. The recommended value for this field is 128 Mb (128 * 1024 * 1024 bytes). The value must be set in bytes (e.g., 134217728).
6. **extRedisKeyExpireTime: a MdInt32 containing an integer that** represents the expire time in milliseconds of the keys on the in-memory-only servers. Whenever a data point is written to in-memory-only storage, the old value keys are not rewritten (some client could at that time be reading them) but only set to expire (are deleted) after the extRedisKeyExpireTime. This value should always be bigger than a time of a write operation on any data point, else a read operation could return an error or malformed value. The recommended value is 5000.

Note that the OLDB client configuration cannot be changed dynamically. If the configuration has been changed, the OLDB client must be restarted for the changes to take effect.

The OLDB configuration is stored in CII config storage under the URI:

```
cii.config://remote/oldb/configurations/oldbclientconfig
```

For testing purposes, a default remote configuration can be deployed on the Configuration Service using the *oldb-initRedis* script. This script will initialize testing OLDB configuration data. Every time the script is run, it will create a new version. The OLDB client always reads the last version of the configuration at start up.

```
oldb-initRedis
```

Calculation Service Configuration

The scheduler process must be configured before it can be run. The configuration is done through the CiiOldbSchedulerConfig in the CII configuration storage [3]. The only field this configuration needs is a list of string URIs of calculation nodes that the scheduler should handle.

The calculation service configuration is stored in CII config storage under the URI:

```
cii.config://remote/oldb/calculationnodes
```

7.4 OLDB Library Usage

This section explains through examples of how to use the OLDB API library in an application. The OLDB API library provides the user the functionality to create, read, write, query, subscribe to and delete data points in the CII OLDB. The API is distributed among two classes. The CiiOldb and CiiOldbDataPoint.

7.4.1 Includes/Imports

For basic usage of the OLDB library, the user needs the CiiOldb client class, the CiiOldbDataPoint and the CiiOldbDpValue class. The first two classes contain the API for manipulating data points and the last one is a container for the data point value triplet (i.e. value, quality, timestamp). Next the language specific URI class is needed. If data points are created with provided metadata, respective metadata classes need to be imported. See section 6.2.3 for the list of metadata classes.

C++

```
#include <ciiOldbDpValue.hpp>
#include <ciiOldbFactory.hpp>
//Metadata classes
#include <meta/MdOldbNumber.hpp>
```

Python

```
import elt.oldb
import elt.config.Uri
```

For subscribing to data point changes, the CiiOldbSubscription interface or a class implementing this interface must be imported.

C++

```
#include <ciiOldbSubscription.hpp>
```

Python

Included in elt.oldb module.

Since OLDB API also throws exceptions, these need to be imported to. For C++ these exceptions can be found in the <ciiOldbExceptions.hpp> header. For Python these exceptions are included in elt.oldb module.

Linked library includes can be seen in code examples of 4.1.2, 4.1.3 and 4.1.4

7.4.2 Basic Example

The examples in this section can be found in the oldb-examples project (sample-app.cpp, cii-oldb-examples-sample-app-py.py).

In basic example (Listing 4-1, Listing 4-2) four constant (without formulas) data points of type DOUBLE, STRING, MATRIX2D_DOUBLE and VECTOR_INT32 are created. Multiple different creation methods are used to create these data points (see appendix C.1 on creation methods). These data points are then read to confirm their initial values. New values are then written to the data points.

Then the data points are queried using a glob expression and again with additional filter arguments (query for the specific type and value range).

Then the children of specific directory parent in the URI hierarchy are queried.

At last, the data points are deleted and OLDB client is closed.

Note: All OLDB client methods are synchronous towards the Redis server.

Listing 4-1: C++ Basic Example

```
/**
 * @copyright (c) Copyright ESO 2019 All Rights Reserved
 * ESO (eso.org) is an Intergovernmental Organisation, and therefore special␣
↪legal conditions apply.
 * @ingroup oldb-examples
 */
#include <mal/utility/Uri.hpp>

#include <ciiOldbFactory.hpp>
#include <ciiOldbDpValue.hpp>
#include <ciiOldbExceptions.hpp>


int main(int ac, char *av[]) {
  try {
    // Initialize OLDB

    std::shared_ptr<elt::oldb::CiiOldb> oldb_client =␣
↪::elt::oldb::CiiOldbFactory::GetInstance();
    ::elt::oldb::CiiOldbGlobal::SetWriteEnabled(true);
```

(continues on next page)

(continued from previous page)

```
    std::string RND_PREFIX = ::elt::oldb::CiiOldbUtil::NewUUID();

    // Define URIs

    ::elt::mal::Uri double_dp_uri{"cii.oldb:///" +RND_PREFIX+ "/sampleroot/child/
↪device/doubledp"};
    ::elt::mal::Uri matrix_dp_uri{"cii.oldb:///" +RND_PREFIX+ "/sampleroot/child/
↪matrixdp"};
    ::elt::mal::Uri string_dp_uri{"cii.oldb:///" +RND_PREFIX+ "/sampleroot/child/
↪stringdp"};
    ::elt::mal::Uri vector_dp_uri{"cii.oldb:///" +RND_PREFIX+ "/sampleroot/child/
↪vectordp"};

    // CREATE DATA POINTS

    // Create DOUBLE data point with provided metadata instance.
    // Metadata must first be created and saved to Config Service.

    const std::string double_dp_meta_instance_name = RND_PREFIX +
↪"customDoubleDpMeta";
    std::shared_ptr<elt::config::CiiDataPointMetadataBase> double_dp_meta =␣
↪CiiDataPointMetadataFactory::getNewMetadataInstance<
      ::elt::config::classes::meta::MdOldb<double>>(double_dp_meta_instance_
↪name);

    double_dp_meta->setComment("metadata of a constant DP");
    double_dp_meta->set_default_value(0.0);
    double_dp_meta->set_min_limit(0.0);
    double_dp_meta->set_max_limit(10.0);

    // meta data needs to be created before calling CreateDataPoint methods

    ::elt::oldb::CiiOldbUtil::SaveOrUpdateMetadata(*double_dp_meta);

    std::shared_ptr<::elt::oldb::CiiOldbDataPoint<double>> double_dp = oldb_
↪client->CreateDataPoint<double>(
      double_dp_uri, double_dp_meta_instance_name);

    // Create MATRIX2D_DOUBLE data point with default matrix metadata␣
↪OldbDoubleMatrixStd
    // and provided initial value. The default metadata must exist on Config␣
↪Service.
    // Run oldb-initRedis script to generate default metadata instances.

    std::vector<double> matrix_double =
      {1.0, 20.0, 30.0, 40.0, 50.0, 4.4, 45.3, 34.4, 445.3, 301.3,
       2.0, 33.3, 34.3, 33.3, 33.3, 5.5, 32.2, 33.4, 222.3, 203.1};

    std::shared_ptr<::elt::config::CiiConfigClient::CiiDataPointMetadataBase>␣
↪mdi_matrix_double =
```

(continues on next page)

(continued from previous page)

```
      ::elt::config::CiiConfigClient::RetrieveMetadata(
        ::elt::oldb::CiiOldbGlobal::GetTargetConfigStorage(),
        ::elt::oldb::CiiOldbGlobal::MDI_MATRIX_DOUBLE,
        ::elt::oldb::CiiOldbGlobal::CONFIG_VERSION);

   bool is_matrix = true;
   std::shared_ptr<::elt:oldb::CiiOldbDataPoint<std::vector<double>>> matrix_dp␣
↪=
      oldb_client->CreateDataPointByValue(matrix_dp_uri, matrix_double, is_
↪matrix);

   // Create STRING data point with default string metadata OldbStringStd and
   // provided initial value.

   std::shared_ptr<::elt::config::CiiConfigClient::CiiDataPointMetadataBase>␣
↪mdi_string =
      ::elt::config::CiiConfigClient::RetrieveMetadata(
        ::elt::oldb::CiiOldbGlobal::GetTargetConfigStorage(),
        ::elt::oldb::CiiOldbGlobal::MDI_STRING,
        ::elt::oldb::CiiOldbGlobal::CONFIG_VERSION);

   std::shared_ptr<::elt:oldb::CiiOldbDataPoint<std::string>> string_dp = oldb_
↪client->CreateDataPointByValue(
     string_dp_uri, std::string("ABCDEF"));

   // Create VECTOR_INT32 data point with provided metadata instance name.
   // Metadata with this instance name must exist in Config Service.

   std::shared_ptr<::elt::config::CiiConfigClient::CiiDataPointMetadataBase>␣
↪mdi_vector_int32 =
      ::elt::config::CiiConfigClient::RetrieveMetadata(
        ::elt::oldb::CiiOldbGlobal::GetTargetConfigStorage(),
        ::elt::oldb::CiiOldbGlobal::MDI_VECTOR_INT32,
        ::elt::oldb::CiiOldbGlobal::CONFIG_VERSION);

   std::cout << "Creating datapoints\n";

   std::shared_ptr<::elt::oldb::CiiOldbTypedDataBase> vector_dp_base =
      oldb_client->CreateDataPoint(vector_dp_uri,␣
↪::elt::oldb::CiiOldbGlobal::MDI_VECTOR_INT32);
   ::elt::oldb::CiiOldbDataPoint<std::vector<std::int32_t>> vector_dp =
     std::dynamic_pointer_cast<::elt::oldb::CiiOldbDataPoint<std::vector
↪<std::int32_t>>>(
       vector_dp_base);
   if (!vector_dp) {
     throw ::elt::oldb::CiiOldbException("Casting to data point of vector int32␣
↪type failed");
   }
```

(continues on next page)

(continued from previous page)

```
    // Read data point values.
    // Unline matrix and string data points, the double_dp was created wihout a␣
↪value.
    // That means a default value of BAD quality was used and since we are␣
↪calling ReadValue before
    // WriteValue, we need to request that the quality check should be skipped.

    std::shared_ptr<::elt::oldb::CiiOldbDpValue<double>> const_dp_value = double_
↪dp->ReadValue(false);
    std::shared_ptr<::elt::oldb::CiiOldbDpValue<std::vector<double>>> matrix_dp_
↪value = matrix_dp->ReadValue();
    std::shared_ptr<::elt::oldb::CiiOldbDpValue<std::string>> string_dp_value =␣
↪string_dp->ReadValue();

    // vectors do not have default value when created, needs a write before read
    vector_dp->WriteValue(std::vector<std::int32_t>{2, 4, 6});
    std::shared_ptr<::elt::oldb::CiiOldbDpValue<std::vector<std::int32_t>>>␣
↪vector_dp_value = vector_dp->ReadValue();

    // Write to data points

    std::cout << "Writing datapoints\n";

    std::vector<double> new_matrix_value(2500);
    for (std::int32_t i = 0; i < 2500; ++i) {
      new_matrix_value.push_back(static_cast<double>(i));
    }

    double_dp->WriteValue(2.0);
    matrix_dp->WriteValue(new_matrix_value);
    string_dp->WriteValue("12345");
    vector_dp->WriteValue(std::vector<std::int32_t>{2, 4, 6});

    // Read the data point values.

    std::cout << "Reading datapoints\n";

    const_dp_value = double_dp->ReadValue();
    matrix_dp_value = matrix_dp->ReadValue();
    string_dp_value = string_dp->ReadValue();
    vector_dp_value = vector_dp->ReadValue();

    // Get multiple data points satisfying an URI glob expression

    std::cout << "Multi-retrieval of datapoints\n";

    std::vector<::elt::mal::Uri> uris = {::elt::mal::Uri("cii.oldb:///sampleroot/
↪**")};
    std::vector<std::shared_ptr<::elt::oldb::CiiOldbTypedDataBase>> search_
↪result = oldb_client->GetDataPoints(uris);
```

(continues on next page)

(continued from previous page)

```
    // Get multiple DOUBLE data points with value between 1.0 and 3.0


    std::cout << "Filtered Search\n";

    std::vector<std::shared_ptr<CiiOldbDataPoint<double>>> filtered_search_
↪result = oldb_client->GetDataPoints<double>(
      uris, ::elt::common::CiiBasicDataType::DOUBLE, 1.0, 3.0);

    // Get the children of directory URI cii.oldb:///sampleroot. Returns a map␣
↪specifying whether
    // a child is a data point or just a directory

    std::map<std::string, bool> children = oldb_client->
↪GetChildren(::elt::mal::Uri("cii.oldb:///sampleroot"));

    // Delete data points

    std::cout << "Deleting datapoints\n";

    oldb_client->DeleteDataPoint(double_dp_uri);
    oldb_client->DeleteDataPoint(matrix_dp_uri);
    oldb_client->DeleteDataPoint(string_dp_uri);
    oldb_client->DeleteDataPoint(vector_dp_uri);

    return 0;
  } catch (const ::elt::oldb::CiiOldbException& ex) {
    std::cerr << "CiiOldbException occured while executing sample code. What: "
      << ex.what() << '\n';
  }
  return -1;
}
```

Listing 4-2: Python Basic Example

```
#!/usr/bin/env python
"""
@copyright (c) Copyright ESO 2019 All Rights Reserved
ESO (eso.org) is an Intergovernmental organisation,
and therefore special legal conditions apply.
@ingroup client-apis-python
@author Cosylab
"""
#pylint: disable=E1101,C0103,R0914,C0330,W0612
#This script shows basic usage of OLDB API.

import sys
import traceback
```

(continues on next page)

(continued from previous page)

```
import uuid
import elt.config
import elt.oldb

# Define URIs

RND_PREFIX = str(uuid.uuid4())

double_dp_uri = elt.config.Uri("cii.oldb:///%s/sampleroot/child/device/doubledp"
↪% RND_PREFIX)
string_dp_uri = elt.config.Uri("cii.oldb:///%s/sampleroot/child/device/stringdp"
↪% RND_PREFIX)
vector_dp_uri = elt.config.Uri("cii.oldb:///%s/sampleroot/child/device/vectordp"
↪% RND_PREFIX)
matrix_dp_uri = elt.config.Uri("cii.oldb:///%s/sampleroot/child/device/matrixdp"
↪% RND_PREFIX)

# Initialize OLDB client
oldb_client = elt.oldb.CiiOldbFactory.get_instance()

def _main():
    """main method implementation"""
    # enable writing
    elt.oldb.CiiOldbGlobal.set_write_enabled(True)

    # create DOUBLE data point with provided metadata instance
    # Metadata must first be created and saved to Config Service.

    double_dp_meta = \
        elt.oldb.typesupport.DOUBLE.\
        get_new_number_metadata_instance("%scustomDoubleDpMeta" % RND_PREFIX)
    double_dp_meta.set_comment("metadata of a constant DP")
    double_dp_meta.set_min_limit(0.0)
    double_dp_meta.set_max_limit(10.0)
    double_dp_meta.set_default_value(0.0)

    # Create & Save Metadata
    elt.oldb.CiiOldbUtil.save_or_update_metadata(double_dp_meta)

    # Create DOUBLE data point
    double_dp = oldb_client.create_data_point(double_dp_uri, double_dp_meta.get_
↪instance_name())

    # Create MATRIX2D_DOUBLE data point with default matrix metadata␣
↪OldbDoubleMatrixStd
    # and provided initial value. The default metadata must exist on Config␣
↪Service.

    matrix_double = elt.oldb.VectorDOUBLE([
```

(continues on next page)

(continued from previous page)

```
      1.0, 20.0, 30.0, 40.0, 50.0, 4.4, 45.3, 34.4, 445.3, 301.3,
      2.0, 33.3, 34.3, 33.3, 33.3, 5.5, 32.2, 33.4, 222.3, 203.1])

    mdi_matrix_double = elt.config.CiiConfigClient.retrieve_metadata(
        elt.oldb.CiiOldbGlobal.get_target_config_storage(),
        elt.oldb.CiiOldbGlobal.MDI_MATRIX_DOUBLE,
        elt.oldb.CiiOldbGlobal.CONFIG_VERSION)

    is_matrix = True

    matrix_dp = oldb_client.create_data_point_by_value(matrix_dp_uri, matrix_
↪double, is_matrix)

    # Create STRING data point with default string metadata OldbStringStd anf␣
↪provided initial value

    mdi_string = elt.config.CiiConfigClient.retrieve_metadata(
        elt.oldb.CiiOldbGlobal.get_target_config_storage(),
        elt.oldb.CiiOldbGlobal.MDI_STRING,
        elt.oldb.CiiOldbGlobal.CONFIG_VERSION)

    string_dp = oldb_client.create_data_point_by_value(string_dp_uri, "ABCDEF")

    # Create VECTOR_INT32 data point with default vector metadata␣
↪OldbDoubleVectorStd

    mdi_vector32 = elt.config.CiiConfigClient.retrieve_metadata(
        elt.oldb.CiiOldbGlobal.get_target_config_storage(),
        elt.oldb.CiiOldbGlobal.MDI_VECTOR_INT32,
        elt.oldb.CiiOldbGlobal.CONFIG_VERSION)

    vector_dp = oldb_client.create_data_point(vector_dp_uri,
                                              elt.oldb.CiiOldbGlobal.MDI_VECTOR_
↪INT32)

    # Read data point values.
    # Unline matrix and string data points, the double_dp was created wihout a␣
↪value.
    # That means a default value of BAD quality was used and since we are␣
↪calling read_value
    # before write_value, we need to request that the quality check should be␣
↪skipped.

    # double_dp was created wihout a value which means a default value
    # of BAD quality was used, therefore when reading we need to request
    # that the quality check should be skipped

    const_dp_value = double_dp.read_value(False)
    matrix_dp_value = matrix_dp.read_value()
```

(continues on next page)

(continued from previous page)

```
    string_dp_value = string_dp.read_value()

    # vectors do not have default value when created, needs a write before read

    vector_dp.write_value([2, 4, 6])
    vector_dp_value = vector_dp.read_value()

    print('Initial Value of double_dp: ', const_dp_value.get_value())
    print('Initial Value of string_dp: ', string_dp_value.get_value())
    print('First Value of vector_dp: ', vector_dp_value.get_value())

    # Write new values to the Data points

    new_matrix_value = elt.oldb.VectorDOUBLE()
    for x in range(250):
        new_matrix_value.append(x)

    double_dp.write_value(2.7)
    matrix_dp.write_value(new_matrix_value)
    string_dp.write_value('New string value')
    vector_dp.write_value([7, 8, 9, 10])

    # Read values
    const_dp_value = double_dp.read_value()
    matrix_dp_value = matrix_dp.read_value()
    string_dp_value = string_dp.read_value()
    vector_dp_value = vector_dp.read_value()

    print('New Value of double_dp: ', const_dp_value.get_value())
    print('New value of matrix_dp: ', matrix_dp_value.get_value())
    print('New Value of string_dp: ', string_dp_value.get_value())
    print('New Value of vector_dp: ', vector_dp_value.get_value())

    # Get Multiple data points satisfying URI glob expression

    uris = [elt.config.Uri('cii.oldb:///%s/sampleroot/**' % RND_PREFIX)]

    search_result = oldb_client.get_data_points(uris)
    print('Search Result: ', search_result)

    # Get Multiple DOUBLE data points with value between 1.0 and 3.0

    filtered_search_result = oldb_client.get_data_points_double(
                        uris, elt.config.CiiBasicDataType.DOUBLE, 1.0, 3.0)
    print('Filtered search result: ', filtered_search_result)

    # Get the children of directory URI cii.oldb:///sampleroot. Returns a map␣
↪specifying whether
    # a child is a data point or just a directory
```

(continues on next page)

(continued from previous page)

```
    children = oldb_client.get_children(elt.config.Uri('cii.oldb:///sampleroot'))

    # Delete data points
    oldb_client.delete_data_point(double_dp_uri)
    oldb_client.delete_data_point(matrix_dp_uri)
    oldb_client.delete_data_point(string_dp_uri)
    oldb_client.delete_data_point(vector_dp_uri)

    # Delete meta data
    elt.config.CiiConfigClient.delete_metadata(
            elt.oldb.CiiOldbGlobal.get_target_config_storage(),
            double_dp_meta.get_instance_name())
    return 0

def main():
    """main method wrapper"""
    result = 0
    try:
        result = _main()
    #pylint: disable=W0703
    except Exception as e:
        print(e)
        traceback.print_tb(sys.exc_info()[2])
        result = 5
    return result

if __name__ == '__main__':
    sys.exit(main())
```

7.4.3 Subscribing Example

The examples in this section can be found in the oldb-examples project (subscription-sample-app.cpp, cii-oldb-examples-subscription-sample-app-py.py).

In the subscribing example (Listing 4-3, Listing 4-4) a data point of type DOUBLE is created and a subscription is added to it to detect value changes and deletion.

Listing 4-3: C++ Subscribing Example

```
/**
 * @copyright (c) Copyright ESO 2019 All Rights Reserved
 * ESO (eso.org) is an Intergovernmental Organisation, and therefore special␣
↪legal conditions apply.
 * @ingroup oldb-examples
 */
#include <mal/utility/Uri.hpp>
```

(continues on next page)

(continued from previous page)

```
#include <ciiOldbFactory.hpp>
#include <ciiOldbSubscription.hpp>
#include <ciiOldbDpValue.hpp>
#include <ciiOldbExceptions.hpp>


namespace elt {
namespace oldb {
namespace app {

class AppOldbDpSubscription : public CiiOldbDpSubscription<double> {
 public:
  AppOldbDpSubscription():
    CiiOldbDpSubscription<double>(::elt::common::CiiBasicDataType::DOUBLE) {}

  void DpRemoved(::elt::mal::Uri uri) override  {
    std::cout << "dpRemoved:" << uri.string() << '\n';
  }

  void NewValue(std::shared_ptr<CiiOldbDpValue<double>> value, ::elt::mal::Uri␣
↪uri) override  {
    std::cout << "newValue:" << uri.string() << " " << value->GetValue() << '\n';
  }
};

}  // namespace app
}  // namespace oldb
}  // namespace elt


int main(int ac, char *av[]) {
  try {
    // Initialize OLDB
    std::shared_ptr<elt::oldb::CiiOldb> oldb_client =␣
↪::elt::oldb::CiiOldbFactory::GetInstance();
    ::elt::oldb::CiiOldbGlobal::SetWriteEnabled(true);

    // Define URIs
    std::string RND_PREFIX = ::elt::oldb::CiiOldbUtil::NewUUID();
    ::elt::mal::Uri double_dp_uri{"cii.oldb:///" +RND_PREFIX+ "sampleroot/child/
↪device/doubledp"};

    // CREATE DATA POINTS

    // meta data needs to be created before calling CreateDataPoint methods

    std::shared_ptr<::elt::config::CiiConfigClient::CiiDataPointMetadataBase>␣
↪mdi_double =
        ::elt::config::CiiConfigClient::RetrieveMetadata(
```

(continues on next page)

(continued from previous page)

```
        ::elt::oldb::CiiOldbGlobal::GetTargetConfigStorage(),
        ::elt::oldb::CiiOldbGlobal::MDI_DOUBLE,
        ::elt::oldb::CiiOldbGlobal::CONFIG_VERSION);

    std::shared_ptr<::elt:oldb::CiiOldbDataPoint<double>> double_dp = oldb_
↪client->CreateDataPointByValue(double_dp_uri, 1.0);

    // Subscribe to a data point value changes.

    std::shared_ptr<::elt::oldb::app::AppOldbDpSubscription> subscription =␣
↪std::make_shared<::elt::oldb::app::AppOldbDpSubscription>();

    double_dp->Subscribe(subscription);

    // Subscription takes affect after some period after Subscribe

    std::this_thread::sleep_for(std::chrono::seconds(2));

    // Write to data points

    double_dp->WriteValue(2.0);

    // Takes time for subscription listener to be called

    std::this_thread::sleep_for(std::chrono::seconds(2));

    // Delete data points

    oldb_client->DeleteDataPoint(double_dp_uri);

    std::this_thread::sleep_for(std::chrono::seconds(3));

    return 0;
  } catch (const ::elt::oldb::CiiOldbException& ex) {
    std::cerr << "CiiOldbException occured while executing sample code. What: "
              << ex.what() << '\n';
  }
  return -1;
}
```

Listing 4-4: Python Subscribing Example

```
#!/usr/bin/env python
"""
@copyright (c) Copyright ESO 2019 All Rights Reserved
ESO (eso.org) is an Intergovernmental organisation,
and therefore special legal conditions apply.
@author Cosylab
"""
```

(continues on next page)

(continued from previous page)

```
#pylint: disable=E1101,C0103,R0914,C0330,W0612

#This script presents DP subscription example.

import sys
import traceback
import time
import threading
import uuid
import elt.config
import elt.oldb

RND_PREFIX = str(uuid.uuid4())

# Initialize OLDB client
oldb_client = elt.oldb.CiiOldbFactory.get_instance()

# Define URIs
double_dp_uri = elt.config.Uri("cii.oldb:///%s/sampleroot/child/device/double_dp
↪" % RND_PREFIX)

class AppOldbDpSubscription:
    """Subscription listener implementation, must implement
       new_value and dp_removed methods
    """
    #pylint: disable=R0201
    def new_value(self, value, uri):
        """Handle DP value change event"""
        print('new_value, value=%s, uri=%s' % (value.get_value(), uri.string()))

    def dp_removed(self, uri):
        """Handle DP remove event"""
        print('dp_removed, uri=%s' % (uri.string(),))

class WorkThread(threading.Thread):
    """Work thread implementation"""

    def run(self):
        """Modify DP then delete it"""
        dp = oldb_client.get_data_point(double_dp_uri)
        for x in range(1, 3):
            dp.write_value(x*.5)
            time.sleep(0.2)
        del dp
        oldb_client.delete_data_point(double_dp_uri)
        # Give subscription system time to pass the message around
        time.sleep(1.0)

def _main():
```

(continues on next page)

(continued from previous page)

```
    """main method implementation"""
    # enable writing
    elt.oldb.CiiOldbGlobal.set_write_enabled(True)

    # create data points
    # metadata needs to be created before calling cereate_data_point method

    mdi_double = elt.config.CiiConfigClient.retrieve_metadata(
        elt.oldb.CiiOldbGlobal.get_target_config_storage(),
        elt.oldb.CiiOldbGlobal.MDI_DOUBLE,
        elt.oldb.CiiOldbGlobal.CONFIG_VERSION)

    double_dp = oldb_client.create_data_point_by_value(double_dp_uri, 1.0)

    # Subscribe to data point. Pass instance of SubscriptionListener() to the␣
↪subscription.
    listener = elt.oldb.typesupport.DOUBLE.get_new_subscription_
↪instance(AppOldbDpSubscription())
    double_dp.subscribe(listener)

    # Release GIL for sufficient time to allow other threads to execute
    # See notes below this listing
    time.sleep(1.0)

    # Do not write to the data points from the same thread. Deadlock because of␣
↪GIL.
    # launch work thread that manipulates data
    workThread = WorkThread()
    workThread.start()
    workThread.join()
    return 0

def main():
    """main method wrapper"""
    result = 0
    try:
        result = _main()
    #pylint: disable=W0703
    except Exception as e:
        print(e)
        traceback.print_tb(sys.exc_info()[2])
        result = 5
    return result

if __name__ == '__main__':
    sys.exit(main())
```

There is important caveat when using subscriptions and writing data point values in the same script. Python interpreter is GIL based meaning that only one thread of execution is active at one time.

When the script creates subscription and then immediatelly begins to write new values to the data point, subscription listener will not get new_value notifications.

This happens because main thread did not release the GIL to allow the whole mechanism to set up (subscriptions are handled in separate thread).

There is no clean solution to this. The suggested workaround is to release GIL in the main thread briefly after the subscription was created. Most handy method is by calling time.sleep() function. For more information please consult ticket ECII-779.

7.4.4 Advanced Example

The source code of the example is located in the CII Demo Repository (https://gitlab.eso.org/cii/info/cii-demo):

1. oldb-examples/cpp/advanced-sample-app/src/advanced-sample-app.cpp
2. oldb-examples/python/app/advanced-sample-app/src/cii-oldb-examples-advanced-sample-app-py.py

The advanced example demonstrates:

1. Creation and use of a calculated datapoint
2. Creation and use of a quality expression
3. Usage of the OLDB write protection mechanism
4. as well as other advanced aspects of OLDB usage.

Note that the calculation service must be running for this example to produce expected results.

7.4.5 Alias Example

The source code of the example is located in the CII Demo Repository (https://gitlab.eso.org/cii/info/cii-demo):

- oldb-examples/cpp/aliases-sample-app/src/aliases-sample-app.cpp

The example demonstrates:

1. Basic Operations: Creation, Read, Update, Delete operations on aliases

```
// Data point needs not exist at the time of alias creation.
// Aliases are type agnostic, but one needs to know the type
// of the data point at the time of retrieval.

// create
client->CreateDataPointAlias (dp1_alias_uri, dp1_uri);

// read
```

(continues on next page)

(continued from previous page)

```
std::shared_ptr<::elt::oldb::CiiOldbDataPoint<std::string>>
   alias_data_point = client->GetDataPoint<std::string> (dp1_alias_uri);

dp1_uri_again = alias_data_point->GetTargetUri();

// update
alias_data_point->WriteValue ("Text Value");

// delete
client->DeleteDataPoint (dp1_alias_uri);
```

2. Deleting underlying datapoints
3. Subscriptions on aliases and their underlying datapoints

7.5 Advanced Topics

This section describes in detail the advanced functionalities of OLDB API library.

7.5.1 OLDB Statistics

Basic traffic statistics can be obtained for the OLDB Client and Calculation service.

OLDB Client Statistics

For OLDB client the basic operation statistics can be obtained directly from Redis by using the redis-cli command line client application. Note that Redis must be installed on the machine so that the redis-cli application is available. Redis client app can be run in the terminal. It can be given options -h (for hostname) and -p (for the port) to connect to specific Redis server. By default it will connect to localhost:6379.

```
redis-cli -h 10.71.0.247
10.71.0.247:6379>
```

Do get basic statistics about the read and write operations the INFO command with commandstats option should be run (Listing 5-1).

Listing 5-1: INFO commandstats result

```
10.71.0.247:6379> INFO commandstats
# Commandstats
cmdstat_expire:calls=54,usec=754,usec_per_call=13.96
cmdstat_subscribe:calls=690,usec=9138,usec_per_call=13.24
cmdstat_exists:calls=296,usec=2683,usec_per_call=9.06
cmdstat_get:calls=948,usec=8600,usec_per_call=9.07
cmdstat_mget:calls=11,usec=88,usec_per_call=8.00
cmdstat_mset:calls=59,usec=662,usec_per_call=11.22
cmdstat_del:calls=190,usec=1708,usec_per_call=8.99
cmdstat_command:calls=3,usec=3138,usec_per_call=1046.00
cmdstat_set:calls=29601,usec=411882,usec_per_call=13.91
cmdstat_publish:calls=29992,usec=318727,usec_per_call=10.63
cmdstat_info:calls=3,usec=273,usec_per_call=91.00
```

The most important fields in the (Listing 5-1) are:

cmdstat_get:calls – number of read operations.

cmdstat_set:calls – number of write operations.

cmdstat_del:calls – number of delete operations.

See [5] for full explanation of the INFO command and how to get other information about the Redis server (e.g. master – slave statistics for cluster servers).

Calculation Service Statistics

As of CII v4, the calculation service does not provide statistics. This may be re-added in a future version.

The logs are located at /var/log/elt/cii-oldb-calc-XYZ.

7.5.2 Disabling Writes to Data Points

Write operations on data points can be disabled or enabled by calling the setWriteEnabled method on the OLDB client. If disabled, all calls of writeValue on data points will throw a CiiOldbWriteDisabledException. Note that the write disabling only effects the same process environment in which it was done. Two OLDB clients on different machines (or in different processes on the same machine) can have different settings regarding the permission of write operations. Write operations are by default enabled.

```
 //Disable write operations on data points
oldbClient.setWriteEnabled(false);
  try {
    inputDubleDp.writeValue(0.2);
  } catch (CiiOldbWriteDisabledException e) {
    System.out.println("Write on DP not permited");
  }
oldbClient.setWriteEnabled(true);
```

When writes to the data points are enabled, a user can also enable an additional check for a formula existance when writing. If the data point formula is not empty and the write is not done by a calculation engine, write will fail with an exception CiiOldbIllegalOperationException. User can enable this check by calling the method SetCheckFormulaIfWriteEnabled in the OLDB client API.

7.5.3 Custom functions in Data Point Formulas

As of CII v4, the Calculation engine does not support custom functions. This may be re-added in a future version.

7.5.4 Manipulating Data Point Metadata

Metadata of the data point can be updated dynamically. Certain metadata attributes (e.g. limits or formula) can be changed or the data point's metadata instance can be replaced altogether (change of the metadata instance name). This can be done by calling the setMetadata method on the data point which accepts a string metadata instance name. The data point metadata can be changed in two ways:

1. The attributes of the metadata of this data point have been changed (the provided metadata instance name is the same as the one the data point already has). These changes must be saved to Configuration Service before calling this method for it to have any effect.

2. The metadata instance of the data point has been replaced (the provided metadata instance name is different than the one the data point already has). The new metadata must already exist and must be of the same type as the old one. This change will be propagated to the data point configuration (Section 5.5).

The change will be propagated to the calculation service so that if the change is relevant to it (e.g. a data point has become a calculated data point), it can take proper actions (e.g. add it to a calculation node).

```
//Update a constant data point metadata to change it into calculated data point
//(add a formula to metadata)
squareMd.setFormula(String.format("DP_VALUE('%s')^2", inputDoubleUri.
↪toString()));


configClient.updateMetadata(
CiiOldbUriUtils.createMetaDataConfigUri(squareMd.getMetadataInstanceName()),
squareMd.getMetadataInstanceName(), -1, squareMd);

squareDoubleDp.setMetadata(squareMd);
```

Note that the type of data point cannot be changed. Trying to update metadata with a different type of metadata will result in CiiOldbInvalidTypeException. That is, the class of the metadata cannot change once it has been defined in the data point at creation. If there is a need for data point type change the data point must be deleted and created again with the updated metadata.

The change will however not be propagated to other instances of the data point. For the change to take effect on these data points must be fetched again from the OLDB.

7.5.5 Data Point Configuration

Each data point has a configuration in Configuration Service which is created automatically at data point creation. It is primarily used for internal management so that the user needs not to concern themselves with it except in a case where the storage location needs to be changed.

The data point configuration is stored in the CiiOldbDataPointInfo object. This configuration class contains the following fields.

1. uri: the MdString class containing the URI of the data point as the string
2. metadataInstanceName: the MdString class containing the data point's metadata instance name as a string.
3. metadataClassName: the MdString class containing the data point's metadata class name as a string (See Table 6-1 for a list of OLDB metadata classes).
4. serverAlias: The MdString class containing the alias for external Redis server (in-memory-only data storage) on which this data point value is stored. All newly created data points have this field *null* which means that their value is stored in main storage (default Redis server).

5. dataTypeGroup: unused MdString class.
6. isSubscriptionNotificationOnly: the MdBoolean class defining behaviour of data point subscriptions on data point value changes. Default is false, which causes subscriptions to receive new data point value via NewValue() subscription callback. When set to true, ValueChanged() subscription callback is invoked providing only data point uri without new value. This is useful for large data points (i.e vectors, matrices) where transferring large quantities of data can hinder application performance.

Fields relevant to the user is serverAlias and isSubscriptionNotificationOnly. Field serverAlias can be used to change the storage location of the data point (Section 5.7). Field isSubscriptionNotificationOnly modifies behaviour of data point subscriptions as described above.

7.5.6 External Redis Servers

To provide the ability for low latency high throughput, the OLDB client has an additional setting that enables the usage of multiple independent Redis servers to run in parallel.

By default, all newly created data points are stored in the default storage location (default Redis servers). This location is defined in the data point configuration. For low latency high throughput configuration, it is possible to also define in-memory-only storage locations (external Redis servers) and transfer data point values there.

External Redis servers are used for data points with larger values that would negatively affect the responsiveness of the main storage. They are regular Redis servers (can be single instances or clusters), but data point values sliced and saved under multiple key-value pairs. The number of slices depends on the data size limit of each slice which is defined with the valueSizeLimitRedisExt field in the OLDB configuration (Section 3.3.1). The list of the external Redis servers that OLDB client will connect to is written in the OLDB configuration. This setting is read at OLDB client start up (See redisServersExternal filed in Section 3.3.1). There each external server is formatted as <alias>:<hostname>:<port>. This alias, which is defined by the administrator can be used to assign a data point to the external server. A single specific data point can be assigned to specific Redis External Server. The data point setting serverAlias is used to control this behavior (5.5).

If extra storage servers are intended to be used for high throughput of large data, properly tuning Redis will be needed. The setting can be set to single or cluster Redis, but when low latency is required a cluster can cause additional delay. By design, Redis is single-threaded, which means that every write or read to the server, will block it. In case a cluster is used, the time to copy the data between the cluster parts need to be considered. If large blocks of data are used, these times can be very long.

The data points that have value written to the external Redis server have also a key-value field on the default Redis storage. This key-value doesn't represent the actual data point value, but it is only the address of the external server (alias of the external Redis).

7.5.7 Defining the Storage Location of a Data Point

By default datapoints are hosted on the main storage. To create a datapoint on an extra storage server (external redis), users set the creation mode before creating the data point.

```
# datapoint will live on main redis
oldb_client->CreateDataPointByValue (data_point_uri1, someValueX);

# reconfigure creation mode
elt::oldb::CiiOldbDpCreateContext custom_context;
std::string extredis = "testExtRedisServer";
custom_context.server_alias = extredis;
oldb_client->SetDataPointCreateContext(custom_context);

# datapoints will live on extredis
oldb_client->CreateDataPointByValue (data_point_uri2, largeValueX);
oldb_client->CreateDataPointByValue (data_point_uri3, largeValueY);

# reset creation mode
::elt::oldb::CiiOldbCreateContext default_context;
oldb_client->SetDataPointCreateContext(default_context);

# datapoint will live on the main redis
oldb_client->CreateDataPointByValue (data_point_uri4, someValueY);
```

Moving an already existing datapoint to a different storage is a highly involved operation, which we discourage. The needed steps include: retrieving and updating the data point configuration using the internal config API, and changing the serverAlias field. If this field is set to null the data point value will be moved to the main storage servers, otherwise it will be moved to external Redis server with this alias. For the configuration change to take effect in the OLDB storages, the data point must be fetched again from the OLDB and a value needs to be written to it. Also note that clients currently using the data point are not notified of the change, so they may be need to be restarted, in order to re-read the oldb.

7.5.8 Enabling notification-only subscriptions on datapoints

To allow oldb client to create data points with notification only subscriptions, instance of structure ::elt::oldb::CiiOldbDpCreateContext must be initialized with is_subscription_notification_only member set to true and then applied to the client with SetDataPointCreateContext() method.

To restore initial client behaviour when creating data points, call client's SetDataPointCreateContext() method with unmodified instance of ::elt::oldb::CiiOldbDpCreateContext structure.

Example:

```
// C++
::elt::oldb::CiiOldbDpCreateContext context;
// All data points created with this context will have their configuration␣
↪setting
```

(continues on next page)

(continued from previous page)

```
// isSubscriptionNotificationOnly set to true, causing their subscriptions to
// receive notifications of the value change trough ValueChanged() callback.
context.is_subscription_notification_only = true;
auto client = ::elt::oldb::CiiOldbFactory.GetInstance();
client->SetDataPointCreateContext(context);
// Create data points ...
client->CreateDataPoint(...)

// Once we are done
::elt::oldb::CiiOldbCreateContext default_context;
client->SetDataPointCreateContext(default_context);
// Data points created after this point, will have their subscriptions receiving
// notifications of the value change through NewValue() callback.
```

```
# Python
import elt.oldb
context = elt.oldb.CiiOldbDpCreateContext()
# All data points created with this context will have their configuration␣
↪setting
# isSubscriptionNotificationOnly set to true, causing their subscriptions to
# receive notifications of the value change trough ValueChanged() callback.
context.is_subscription_notification_only = True
client = elt.oldb.CiiOldbFactory.get_instance()
client.set_data_point_create_context(context)
# Create data points ...
client.create_data_point(...)

# Once we are done
default_context = elt.oldb.CiiOldbDpCreateContext()
client.set_data_point_create_context(default_context)
# Data points created after this point, will have their subscriptions receiving
# notifications of the value change through NewValue() callback.
```

7.6 OLDB API LIBRARY

This section explains the details of the srv-oldb library. The OLDB API is split between two classes. The CiiOldb and CiiOldbDataPoint.

7.6.1 CiiOldb

The core OLDB Client API is the singleton CiiOldb class. The CiiOldb class contains methods for creating retrieving and deleting data points. See Appendix C for a description of CiiOldb API.

7.6.2 CiiOldbDataPoint

The other part of the OLDB Client API is the CiiOldbDataPoint class (Figure 3). It represents the data point in the OLDB and allows reading from and writing to it. The CiiOldbDataPoint class includes three main fields:

URI: the unique data point identifier.

CiiOldbDpValue: the value class containing the data point data value, timestamp and quality. It is parameterized with the type of the data value.

CiiOldbMetaData: the metadata class that contains data point type, formula, quality expression and other data type specific information.

The CiiOldbDataPoint is a generic class parameterized with the programming language type of the data value (for C++).

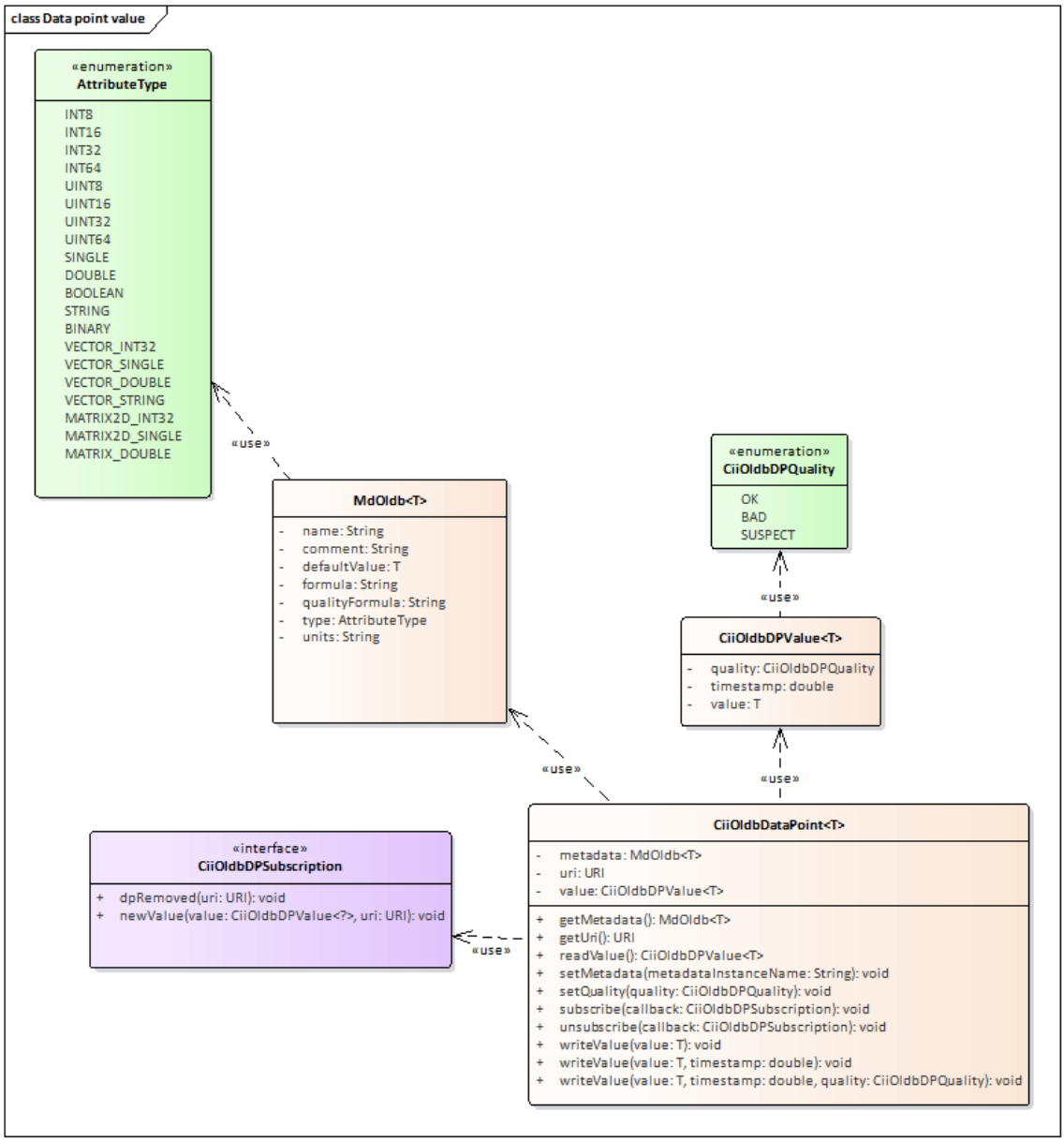


Figure 3: CiiOldbDataPoint class diagram

Data Point URI

Data point URI is the unique hierarchy identifier of the data point and contains OLDB schema cii.oldb:/// and the path to the data point (e.g. cii.oldb:///root/child/dp1).

Note: To make OLDB hierarchy working as expected, the data point URI should not contain any triple underscore (“___”) character. Because internally all “/” characters are converted to triple underscores, a triple underscores in a level name would result in an additional level.

Note: For the same reason, do not use path components that end with an underscore. For example cii.oldb:///a/b_/c will translate to a sequence of 4 underscores which would wrongly translate back to cii.oldb:///a/b/_c.

Note: Due to a limitation of the config service, the path component of the data point URI should not contain a sole number, i.e. cii.oldb:///a/12/dp1 is not allowed. Creating a data point with a number in the URI path will throw an exception clearly stating that a number in URI path is not allowed.

```
URI uri = point.getUri();
```

Note: The OLDB is case insensitive so two URIs differing only in case are considered the same. All URIs are internally stored in lower case.

Note: It is valid that a data point URI path is a sub-path of another data point URI path since the Configuration Service does not hold the URIs in a hierarchy structure. So for example, if there a data point with URI cii.oldb:///some/path/a, a data point with cii.oldb:///some/path/a/b can be created.

Data Point Value

The CiiOldbDpValue is a generic class parameterized with the programming language type of the data value (for C++, but not Python because it is not strongly typed).

```
CiiOldbDpValue<T> value = datapoint->ReadValue(false);
```

The CiiOldbDpValue contains three pieces of information that dynamically change during the data point lifetime:

Value

The actual data value of the data point. Data types of values are listed in Appendix A.

Quality

The CiiOldbDpQuality enumeration that represents the quality state of the data point. Possible values are OK, SUSPECT or BAD.

The meanings of the quality state are as follows:

OK: the value of the Property is valid

BAD: the value is not OK and should not be used

SUSPECT: there is a value, but it is not clear if it is reliable. For example, the readout of a sensor is inconsistent with other readings and the sensor might be faulty.

If the data point quality is not OK, data point value cannot be read and written to. In order to read the datapoint with a BAD/SUSPECT quality is necessary to call ‘ReadValue’ method with a ‘False’

boolean value. Note that the difference between BAD and SUSPECT is justsemantic. The OLDB library threats these two states the same. If anything goes wrong with the data point internally (e.g. error in the data point formula) the quality is set to BAD. It is up to the user to set the quality to SUSPECT if appropriate.

Timestamp

The moment in time when the data point was last modified.

Timestamps are in UTC, with nanosecond resolution, since the epoch.

Example: This code would produce a valid timestamp in C++ (ignoring differences between Unix Time and UTC time):

```
auto now = std::chrono::system_clock::now();
auto nanos = std::chrono::time_point_cast<std::chrono::nanoseconds>(now);
auto since = nanos.time_since_epoch();
int64_t timestamp = since.count();
```

Metadata

The MdOldb is the class containing all the metadata information about the data point. It is not expected that this information would change often but it is possible to manipulate it through the OLDB API (Section 5.4). It is a subtype of MdBase from the Cii Config Service [3] so it contains all the fields contained in the MdBase. The OLDB relevant fields of the MdBase are genType (the type of the data point value) and defaultValue (the default value of the data point). The OLDB specific fields contained in the base class MdOldb are data point formula and qualityExpression.

For other fields in MdBase class consult the [3] document.

The CiiOldbDataPoint class contains a subtype of MdOldb specific to the data point type (Table 6-1). The user of the CiiOldbDataPoint must know the type of the data point and the metadata class corresponding to it so they can retrieve the metadata object, as seen in the following example:

```
MdOldbString metadata = point.getMetadata(MdOldbString.class);
```

If at runtime the type of the data point is not known, the user can determine it by retrieving the metadata as the MdOldb base type, retrieving the generic type from it and casting the metadata in the right class.

```
MdOldb metadata = point.getMetadata(MdOldb.class);
AttributeType type = metadata.getGenType();
if (type == AttributeType.STRING) {
  metadata = (MOldbString) metdata;
}
```

All the built-in OLDB metadata classes are listed in Table 6-1. All classes derive from MdOldb (which

derives from config MdBase) class which defines basic OLDB fields (quality and formula). The metadata class associated with the data type can be read in Appendix B.

Note that it is not meant to extend the existing metadata classes and define new “types” of data points since the data types defined by Core Integration Infrastructure are covered with existing metadata. It is in theory possible to extend OLDB metadata classes, but it means extending the OLDB design and coding of new extended classes.

Table 6-1: Built-in metadata classes

Class	Description	Specific Fields
MdOldbNumber	Metadata for all scalar number types. Parameterized with the language mapping of the type	minLimit (data point value type): the lower limit of the data point value. maxLimit (data point value type): the upper limit of the data point value. units (string): the units of the data point.
MdOldbString	Metadata for string data types. Not parameterized.	allowedValues (list): all the values the data point value is allowed to have. If the list is empty or *null*, all values are allowed. size (integer): the length of the string value. This is not a limit just informational filed.
MdOldbArray	Metadata for all vector data points. Parameterized with the type of the elements.	
MdOldbMatrix	Metadata for all matrix data points. Parameterized with the type of the elements. This is a subtype of MdOldbArray*	width (integer): The width of the matrix (the number of elements in each matrix row).
MdOldbBinary	Metadata for byte array data points.	

*Note: The matrix type data point actually holds the matrix flattened in to a vector. The metadata then holds the width of the matrix (number of elements in each row) so the user can reconstruct the matrix (e.g. if data point value is a vector of length 25 and the width property of the metadata is 5, the data point represents a 5x5 matrix). The length of the vector value must be divisible by the width otherwise the CiiConfigDataLimitsException is thrown.

Data Point Type

The Core Integration Infrastructure defines the types of all data values. All data point values are one of these predefined types. The predefined type effectively defines the type of the data point. Programmatically the data point type is defined in the AttributeType enumeration class (elt-common). See Appendix A for table of types relevant for data point values and their mappings to specific programming language types. The data point type is stored in the data point metadata in the genType field. Data point type cannot change during the lifetime of a data point. Data point with specific URI

must be deleted and created again if the type needs to be changed.

```
AttributeType type = metadata.getGenType();
```

Data Point Formula

Data point formula is a mathematical expression by which the value of the data point can be calculated. If data point defines a formula, it is a calculated data point. Calculated data points cannot be directly written to (trying to do so will result in the CiiOldbIllegalOperationException). They are handled by the Calculation Service. The data point formula is defined in the data point metadata.

```
String formula = metadata.getFormula();
```

The Calculation Service uses the python language, and the data point formula must be a valid python expression and must return a value of the data point type.

Data point formula can contain references to other data points. This makes the data point dependent on other data points. The dependent data point value will be recalculated when one of its dependencies change. A valid reference to another data point is via a call to one of the DP_xyz functions with the data point URI given as a string argument. The engine expects the full URI of the data point.

Note: The URI string argument MUST be enclosed in SINGLE-QUOTES. See the example below. Using double-quotes, or triple quotes, or other variants, will not work, due to shortcomings in the current implementation.

Example:

```
DP_VALUE('cii.oldb:///root/dp1') + 3 * sqrt(DP_VALUE('cii.oldb:///root/dp2'))
```

Data point formula must not establish cyclic dependencies. Doing that will result in a CiiOldbCyclicDependencyException at the data point creation or metadata update.

The following references can be used to access a datapoint:

```
DP_VALUE(uri)
DP_QUALITY(uri)
DP_TIMESTAMP(uri)
```

and to access earlier data (from the previous computation):

```
DP_PREV_VALUE(uri)
DP_PREV_QUALITY(uri)
DP_PREV_TIMESTAMP(uri)
```

Any syntax errors in the data point formula will result in data point quality to be set to BAD by the Calculation Service.

Data Point Quality Expression

The quality expression is an expression which defines the data point quality. The data point quality represents the quality state of the data point and can have values OK, SUSPECT or BAD.

Data point quality evaluation is done by the Calculation Service (Section 3.2.3) and must be a valid python expression. The data point quality expression must return one of the strings “OK”, “SUSPECT” or “BAD”.

Example:

```
if_else (DP_VALUE('cii.oldb:///root/dp1') > 1.0,  "OK",  "BAD")
```

Any syntax error in the quality expression will result in data point quality to be set to BAD by the Calculation Service.

A quality expression can only be set on a calculated datapoint (i.e. one that also has a value formula).

A quality expression may reference the value of its own datapoint (in a value formula, this would be an illegal self-reference).

Data point value aging

Metadata attribute maxAge enables aging of data point values.

If defines number of seconds after data point value goes stale in case it was not updated. Once data point value is stale, data point quality is changed to BAD. Callback method ValueStale() of data point subscriptions is also invoked.

Default value for maxAge attribute is 0, causing data point values never to become stale.

To turn on data point value aging, metadata attribute maxAge must be set to non zero positive value and updated metadata applied to the data point.

Example:

```
// C++
auto metadata = datapoint->GetMetadata();
metadata.setComment("Enabled aging, values stale after 2 seconds");
metadata.set_maxAge(2);
::elt::oldb::CiiOldbUtil::SaveOrUpdateMetadata(metadata);
// Make sure data point is aware that metadata has changed
datapoint->SetMetadata(metadata->getInstanceName());
```

```
# Python
metadata = datapoint.get_metadata()
metadata.set_maxAge(2)
elt.oldb.CiiOldbUtil.save_or_update_metadata(metadata)
# Make sure data point is aware that metadata has changed
datapoint.set_metadata(metadata.get_instance_name())
```

7.7 OLDB CLI

This section describes how to use OLDB command line application. The oldb-cli tool can be used for read, write, subscribe, create, delete operations and with data points that have textual outputs. The output of the tool can be parsed and used in other scripts.

New in CII v4: oldb-cli allows to omit the oldb-scheme prefix from a <data point uri>. If you omit the scheme, “cii.oldb://” will be assumed.

Example:

```
# with scheme
$ oldb-cli read cii.oldb:///glob/root/child/testintdp

# without scheme
$ oldb-cli read /glob/root/child/testintdp
```

7.7.1 Read operations

The tool reads the current value of the data point at the given data point URI and displays it on the standard output. Depending on the options used it can also display data point’s quality expression and formula as well. Arrays and matrices are displayed in the standard JSON format. It is also possible to write the value to a file (Section 7.1.3).

The syntax for read operations is:

```
oldb-cli read <data point URI> [-f <output file>|-e|-vf|-time|-quality|-value]
```

The -h argument prints help description to standard output.

Read value

To read a value of a data point, use the client together with an URI that points to the desired data point. If the data type of value is BINARY or its string representation exceeds 500 characters, the value won’t be displayed on the standard output and you will have to use the output file option to retrieve it (Section 7.1.3), otherwise an error will be thrown. To output just a single part of the (timestamp/quality/value) group, the options can be used (-time, -quality, -value).

Example:

```
$ oldb-cli read cii.oldb:///glob/root/child/testintdp
Timestamp: 2020-01-13T14:49:54.702Z
Quality:  OK
Value:  13
```

Read Value Formula and Quality Expression

To also retrieve value formula or quality expression of a data point use the options -vf (for value formula) and/or -e (for quality expression).

Example:

```
$ oldb-cli read cii.oldb:///glob/root/child/testintdp -vf -e
Quality expression: if (CURR_VAL() > 0) SETQUAL(OK) else SETQUAL(BAD) endif
Value formula: DP_VALUE('/root/child/device/DP1') + 3 * DP_VALUE('/root/child/
↪device/DP2')
Timestamp: 2020-01-13T14:49:54.702Z
Quality:  OK
Value:  13
```

Save value to File

To save the data point's value to a file use the -f option, followed by a path to file to write to. Unless the datatype of value is BINARY or its string representation exceeds 500 characters, the value will still get displayed on standard output as well.

Example:

Saving to file:

```
$ oldb-cli read cii.oldb:///glob/root/child/teststringdp -f testStringDpValue
Timestamp: 2020-01-07T19:09:13.117Z
Quality:  OK
Value:  ABCDEF
```

Reading the value from file:

```
$ cat testStringDpValue
ABCDEF
```

7.7.2 Write operations

The tool gets the data point at the given URI and writes a new value to it. Data point's quality, timestamp, formula and quality expression can all be changed through the tool with the right options. Arrays and matrices must be given in the standard JSON format. Values can be given through a file too.

The syntax for writing is:

```
oldb-cli write <data point uri> <value>  [-f] [-h] [-q <quality>] [-e <quality␣
↪expression>] [-t <timestamp>] [-vf]
```

The -h argument prints help description to standard output.

Write value

To write a value to data point, use the client together with an URI that points to the desired data point and value to be written. If the datatype of the value to be written is BINARY, the value must be given through the -f option (Section 7.2.5).

Example:

```
$ oldb-cli write cii.oldb:///glob/root/child/testintdp 1234
Timestamp: 2020-01-14T15:34:52.392Z
Quality:  OK
Value:  1234
```

Write Quality and Timestamp

When setting a data point's value, the quality and timestamp can also be set.

To set the quality use the -q option followed by the desired quality (OK/SUSPECT/BAD).

Example:

```
$ oldb-cli write cii.oldb:///glob/root/child/testintdp 1234 -q BAD
```

```
Timestamp: 2020-01-14T15:34:52.392Z
```

```
Quality:  OK
Value:  1234
```

To set a timestamp to the data point use the -t option, followed by the timestamp in ISO 8601 extended date-time format.

Example:

```
$ oldb-cli write cii.oldb:///glob/root/child/testintdp 1234 -t 2020-01-
↪07T19:09:13.117Z
Timestamp: 2020-01-07T19:09:13.117Z
Quality:  OK
Value:  1234
```

Note: all three data point properties (value, quality, timestamp) can be set in one operation using all the above options.

Write Formula

If you want to set a formula to a data point, replace the value argument with the value formula, and use the -vf option. The functionality will fist read the metadata instance name that belongs to the given data point and then writes the formula to the metadata instance.

Note: Formula is a part of the data point metadata instance. As one metadata instance can belong to many datapoints, this change can affect multiple data points.

Example:

```
oldb-cli write cii.oldb:///glob/root/child/testintdp3 'DP_VALUE(\'cii.oldb:///
↪glob/root/child/testintdp2\') + 3 * sqrt(DP_VALUE(\'cii.oldb:///glob/root/
↪child/testintdp\'))' -vf
```

Write Quality Expression

If instead of setting a fixed quality, you want to set a quality expression, you can do so by using the -e option followed by the desired quality expression. In this case, the value argument can be omitted.

Example:

```
oldb-cli write cii.oldb:///glob/root/child/testintdp2 -e 'if_else (DP_VALUE('cii.
↪oldb:///glob/root/child/testintdp2') > 0,  "OK", "BAD")'
```

Write Value from File

To write a value from a file, replace the value argument with a path to the file containing the value and use the -f option.

Example:

Writing value from a file:

```
$ oldb-cli write cii.oldb:///glob/root/child/testintdp /home/esodev/values/
↪testIntDpValue -f
Timestamp: 2020-01-14T15:40:34.783Z
Quality:  OK
Value:  25
```

Write Value directly from Formula

Value to specific data point can be written directly from the expression formula. To use the formula, the value must start with the "=" character.

Example:

Writing values from a formula:

```
$ oldb-cli write cii.oldb:///glob/root/child/testdoubledp '=e(3)'
Timestamp: 2020-07-17T08:48:22.399Z
Quality:  OK
Value:  20.085536923187668


$ oldb-cli write cii.oldb:///glob/root/child/testintdp '=DP_VALUE(\'cii.oldb:///
↪glob/root/child/testintdp2\') + DP_VALUE(\'cii.oldb:///glob/root/child/
↪testintdp3\')'

Timestamp: 2020-07-17T08:50:00.445Z
Quality:  OK
Value:  4435
```

7.7.3 Subscribe operations

The tool retrieves the data point at the given URI, subscribes to it and displays value changes to the standard output until user interrupts it or the data point is deleted. Arrays and matrices are displayed in the standard JSON format.

The syntax for subscribe functionality is:

```
$ oldb-cli subscribe <data point uri>  [-f <output file prefix>] [-h]
```

Subscribe to datapoint

To subscribe to a data point, use the client together with an URI that points to the desired data point. If the data type of value is BINARY or its string representation exceeds 500 characters, the value won't be displayed on the standard output and you will have to use the output file prefix option (Section 7.3.2) to have changes written to files.

Example:

```
$ oldb-cli subscribe cii.oldb:///glob/root/child/testintdp
Subscribed to cii.oldb:///glob/root/child/testintdp
To stop the subscription interrupt the program with Ctrl-C.
Timestamp: 2020-01-10T13:04:52.788Z
Quality:  OK
```

(continues on next page)

(continued from previous page)

```
Value:  30
Timestamp: 2020-01-10T13:05:11.474Z
Quality:  OK
Value:  25
Timestamp: 2020-01-10T13:05:19.885Z
Quality:  OK
Value:  20
Timestamp: 2020-01-10T13:05:27.098Z
Quality:  OK
Value:  15
Timestamp: 2020-01-10T13:05:34.134Z
Quality:  OK
Value:  10
(base) [esodev@localhost values]$
```

Subscribe and output to Files

To write the received values to files, use the -f option with a file name prefix. The files will then be placed in the current directory and named as <prefix_timestamp> (for each value update a new file is created).

Example:

```
$ oldb-cli subscribe cii.oldb:///glob/root/child/testintdp -f testIntDpValues
Subscribed to cii.oldb:///glob/root/child/testintdp
To stop the subscription interrupt the program with Ctrl-C.
Timestamp: 2020-01-10T13:04:52.788Z
Quality:  OK
Value:  30
Timestamp: 2020-01-10T13:05:11.474Z
Quality:  OK
Value:  25
Timestamp: 2020-01-10T13:05:19.885Z
Quality:  OK
Value:  20
Timestamp: 2020-01-10T13:05:27.098Z
Quality:  OK
Value:  15
Timestamp: 2020-01-10T13:05:34.134Z
Quality:  OK
Value:  10
^C(base) [esodev@localhost values]$ ls
testIntDpValues_2020-01-10T13:04:52.788Z  testIntDpValues_2020-01-10T13:05:11.
↪474Z  testIntDpValues_2020-01-10T13:05:19.885Z testIntDpValues_2020-01-
↪10T13:05:27.098Z testIntDpValues_2020-01-10T13:05:34.134Z
```

The values are stored inside the files.

Example:

```
[esodev@localhost values]$ cat testIntDpValues_2020-01-10T13:05:19.885Z
20
[esodev@localhost values]$
```

7.7.4 Create operation

The create operation creates a datapoint of a specified type. For further configuration of the new data point, use the write operations. The datapoint will have its own dedicated metadata instance, therefore setting e.g. a formula on the datapoint will not affect other datapoints.

The syntax for the create operations is:

```
oldb-cli create <datapoint_uri>  <type>

# The -h argument prints the operation-specific help
oldb-cli.py create -h
```

Example:

```
$ oldb-cli create cii.oldb:///glob/root/child/teststring  STRING

$ oldb-cli write  cii.oldb:///glob/root/child/teststring  "my string"
```

7.7.5 Delete operation

The delete operation deletes a datapoint.

The syntax for the delete operations is:

```
oldb-cli delete <datapoint_uri>

# The -h argument prints the operation-specific help
oldb-cli.py delete -h
```

Example:

```
$ oldb-cli delete cii.oldb:///glob/root/child/teststring
```

7.8 GUI

This section describes the components of OLDB GUI. OLDB GUI has functionalities to read, write, create, delete and update data points. The GUI can also be used to read metadata of data points, but changing of metadata is not supported.

7.8.1 General

OLDB GUI is a graphical application that can be used for managing the data points and displaying their values. The data points are retrieved from the remote OLDB service.

```
oldbGui [path_to_application_config_file] [path_to_log_config_file]
```

The oldb gui executable takes 2 optional arguments:

- path_to_application_config_file: a path to the file that contains the settings for the OLDB GUI application. An example value for the attribute: ../config/app-config.ini. A detailed description of the application configuration file can be found in section 8.2.
- path_to_log_config_file: a path to the file that contains the settings for the log4cplus logging library. If it is not provided, the basic log4cplus configuration is used. Usually, the configuration file for the log4cplus library is named log4cplus.properties. Example value for the attribute: ../log4cplus.properties.

7.8.2 Configuration File

The configuration file for the OLDB GUI application contains the information needed to connect to the remote configuration service (i.e. the data source). The configuration file contains:

- DataSource: this section contains all the data needed to connect to the primary configuration service. This service is used for retrieving the metadata about every selected data point. The **hostname** attribute provides the address of the service (in a form of an IP address or a host name) and the **port** attribute provides a port on which the service listens for incoming connections.
- MatrixEdit: this section determines the max number of rows (maxHeight) and columns (maxWidth) that a given vector or matrix can have in order for its data to be displayed and edited directly in the OLDB GUI (for vectors, the maxWidth defines the size limit). Vectors and matrices that are larger than the MatrixEdit values can only be downloaded to the local disk and uploaded from the local disk.

If “path_to_application_config_file” is not provided to the application, default values will be used.

7.8.3 Functionality

Main Window

The main window is displayed when we execute the OLDB GUI. The application screenshot is shown in Figure 8-1. As we can see from the figure, the functionality of the OLDB GUI is related to the management of data points.

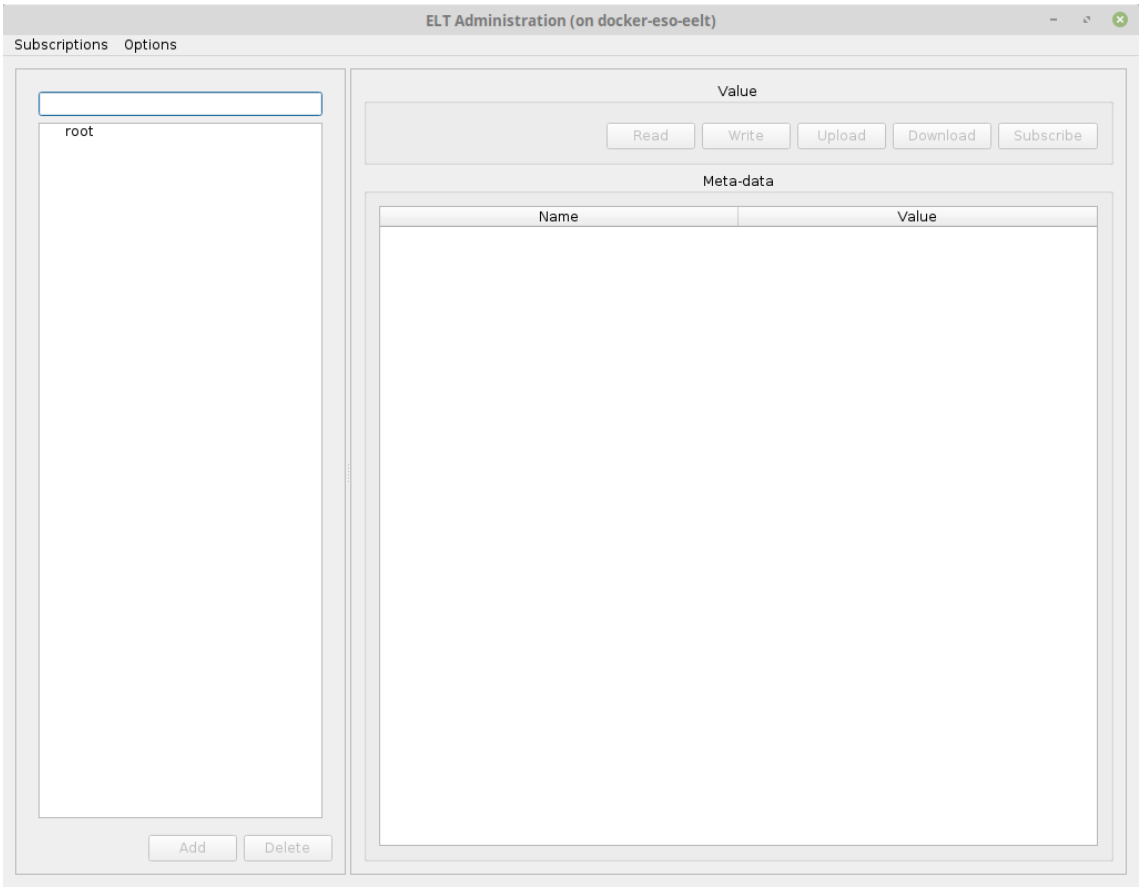


Figure 8-1: Main window

The GUI is divided into 2 parts. The left part displays the list of folders and data points retrieved from the remote service, search bar and the buttons for adding and deleting data points. The user can select the folders and data points by clicking them.

The right part is used for displaying the data about the selected data point: its value and metadata.

Menu

The OLDB GUI application only contains two menu entries, the **Subscriptions** (Figure 8-2) and **Options**. The Subscription menu entry contains the 3 items for managing data point subscriptions (i.e. the windows for continuously displaying the latest data point values):

- **Load Subscriptions**: loads the list of data point subscriptions from a file on disk (through file explorer) and opens them in windows in saved positions. All subscriptions are started (the value updates will be visible). If a subscription window is already displayed for a data point with a given URI, a new subscription window won't be displayed once more.
- **Save Subscriptions**: saves the list of currently displayed data point subscription windows (and their position on the screen) into a CSV file on disk. A file explorer window is opened and the user can enter the file name and location.
- **Close All Subscriptions**: closes all the data point subscription windows that are currently displayed (unsubscribes from the data points).

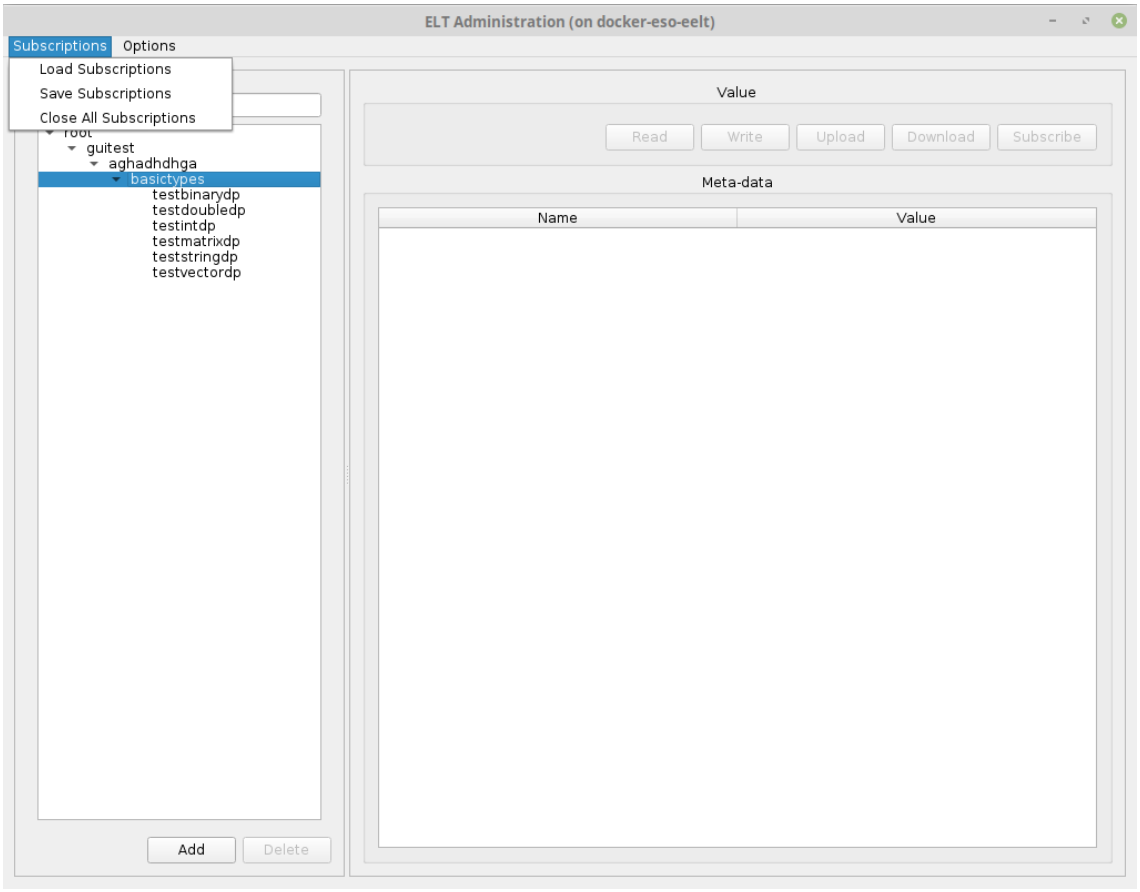


Figure 8-2: Application menu

The Options menu contains a checkbox item "write enabled". This checkbox must be checked to enable write operations on data points. If this checkbox is not checked, writing to data points will not be possible. It is not checked by default.

7.8.4 Hierarchy view

The hierarchical view displays the hierarchical structure of folders and data points. It enables the user to do the following actions:

- Browse and view the data points that are stored in the remote service.
- Add new data points.
- Delete existing data points.
- Trigger the loading of data point values.

1. Selecting Folder

The user selects a folder by clicking the inner node in the data point hierarchy tree (Figure 8-3). If the leaf tree node is clicked, the data point is selected. The folders are created automatically, based on the data points' URIs.

When a folder is selected, all the data in the data point frame is cleared. The folders don't have any values or metadata, they are only used for organizing data points. When a folder is selected the user can create a new data point by clicking on the Add button below the hierarchy tree.

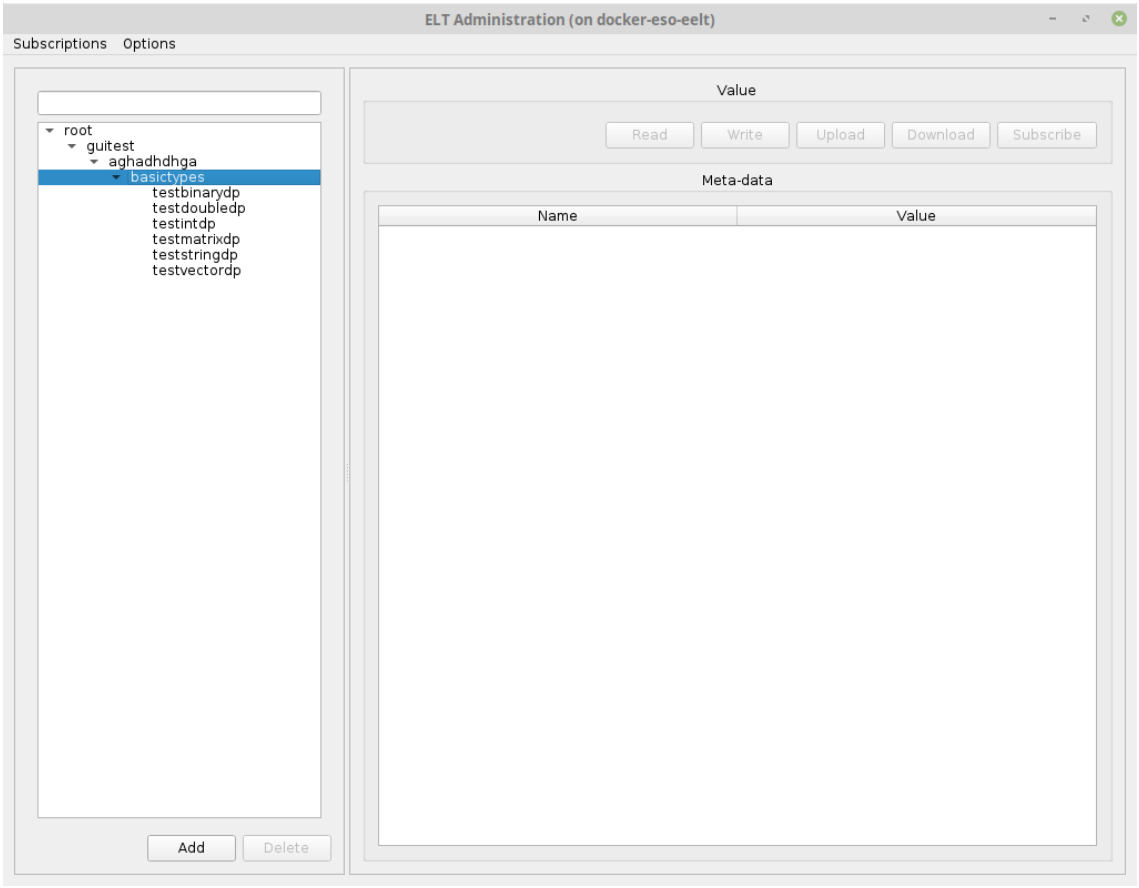


Figure 8-3: Selecting folder

2. Selecting Data Point

The user selects a data point by clicking the leaf node in the data point hierarchy tree (Figure 8-4). The data points are organized in the tree structure based on their URIs.

For the easier searching of data points, a search filter bar is located at the top of the hierarchy tree. The user can input a part of the URI and when he/she hits the Enter key, all the data points whose name doesn't contain the string input are filtered out (the string comparison is case insensitive). In order to display all the data points again, the user needs to clear the filtering text and hit the Enter key again.

When a configuration is selected, the following data is loaded in the data point frame of the GUI:

- data point value, quality and timestamp: the most recent value, quality and timestamp of the selected data point.
- the list of data point's metadata attributes and their values.

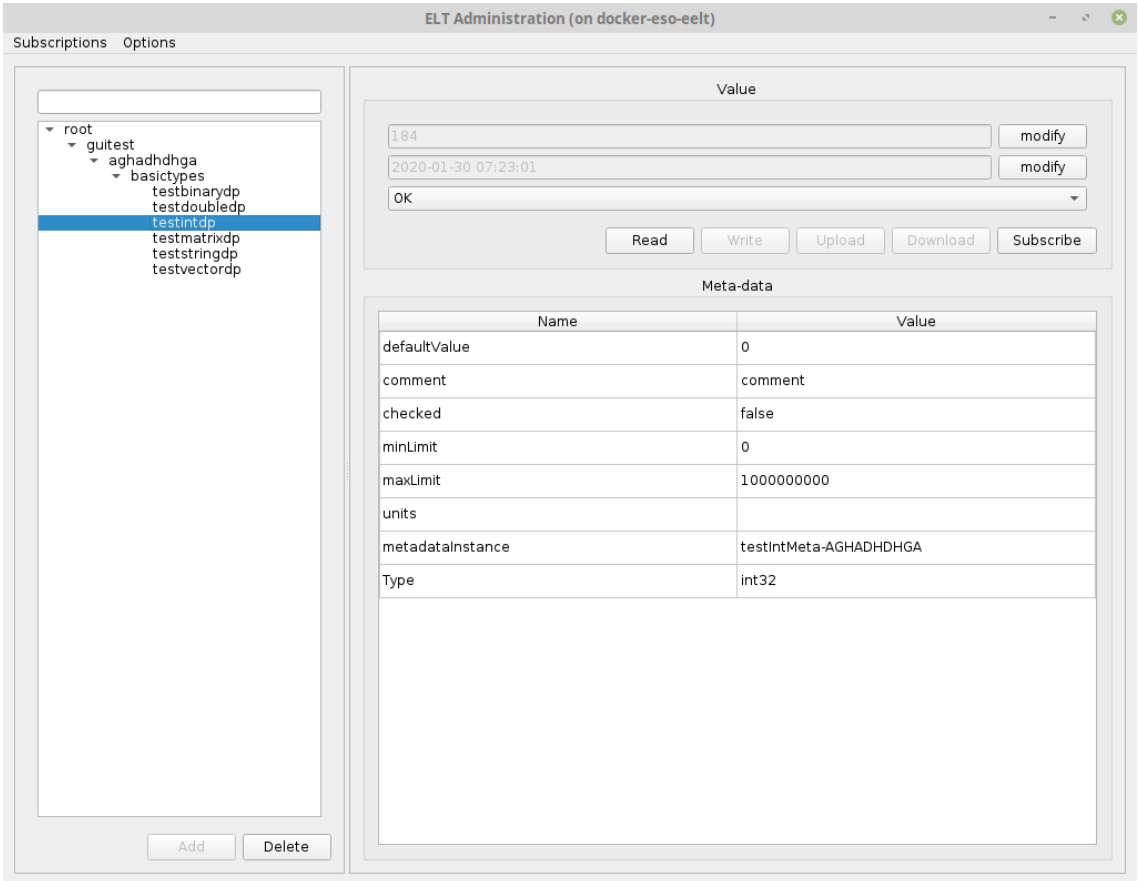


Figure 8-4: Data point selected

Note that it is not possible to change data point metadata through OLDB GUI application.

The data point URI is at the moment not showed anywhere in the data point frame. The user can reconstruct the data point URI from the tree path.

3. Adding a New Data Point

The adding of new data point is initiated by clicking the Add button below the hierarchy tree. The button is only enabled if a folder is selected.

When the Add button is clicked, a dialog for inserting a data point is displayed (Figure 8-5). The dialog contains the following fields:

- URI prefix: a read-only field that contains the URI prefix
- URI suffix: the input field where the user can input the URI suffix or the name of the data point. The final URI of the configuration is formed by concatenating the URI prefix and suffix.
- Metadata instance: here, the user selects the metadata instance for the newly created data point.

After the user clicks the OK button, the dialog is closed and a new data point is created on the remote server.

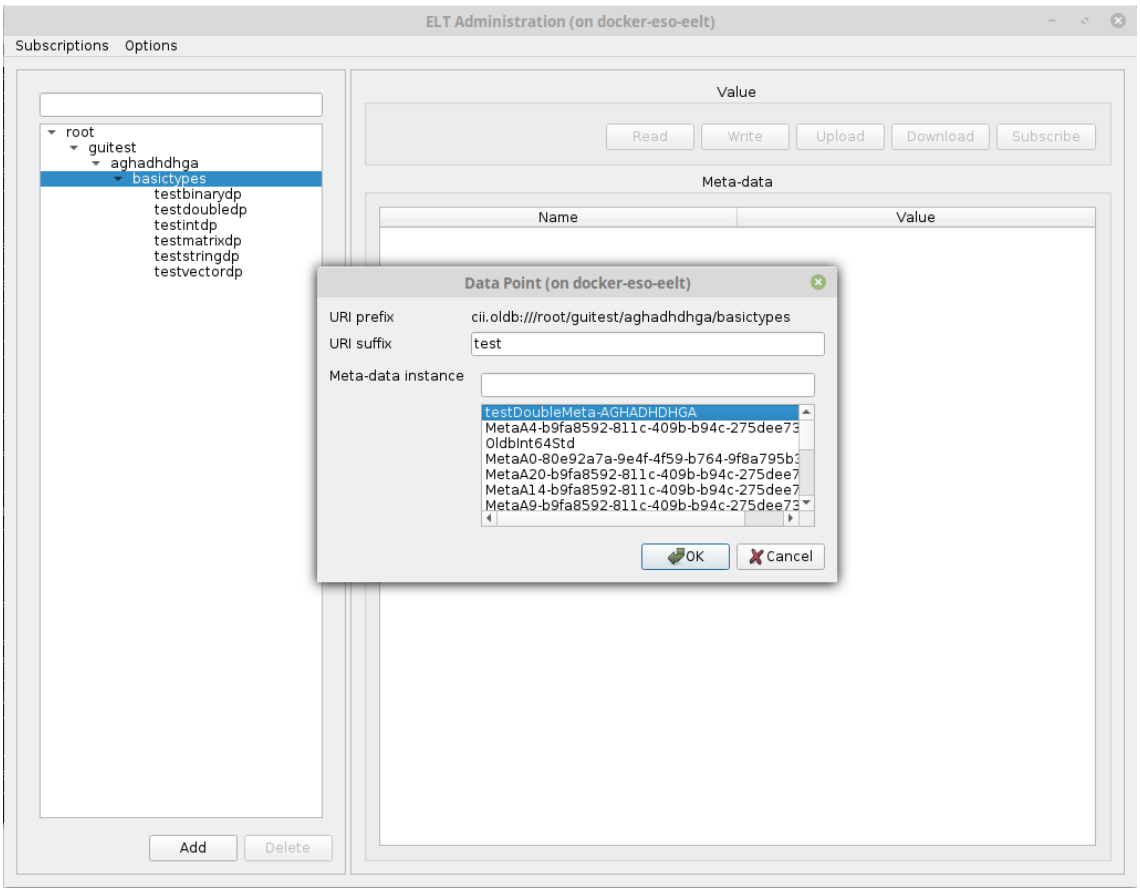


Figure 8-5: Adding new data point

4. Deleting an Existing Data Point

The deleting of existing data point is initiated by clicking the Delete button below the hierarchy tree. The button is only enabled if a data point is selected.

When the Delete button is clicked, a confirmation dialog is displayed where the user can confirm or cancel the deletion. On user confirmation, the data point is deleted on the remote server.

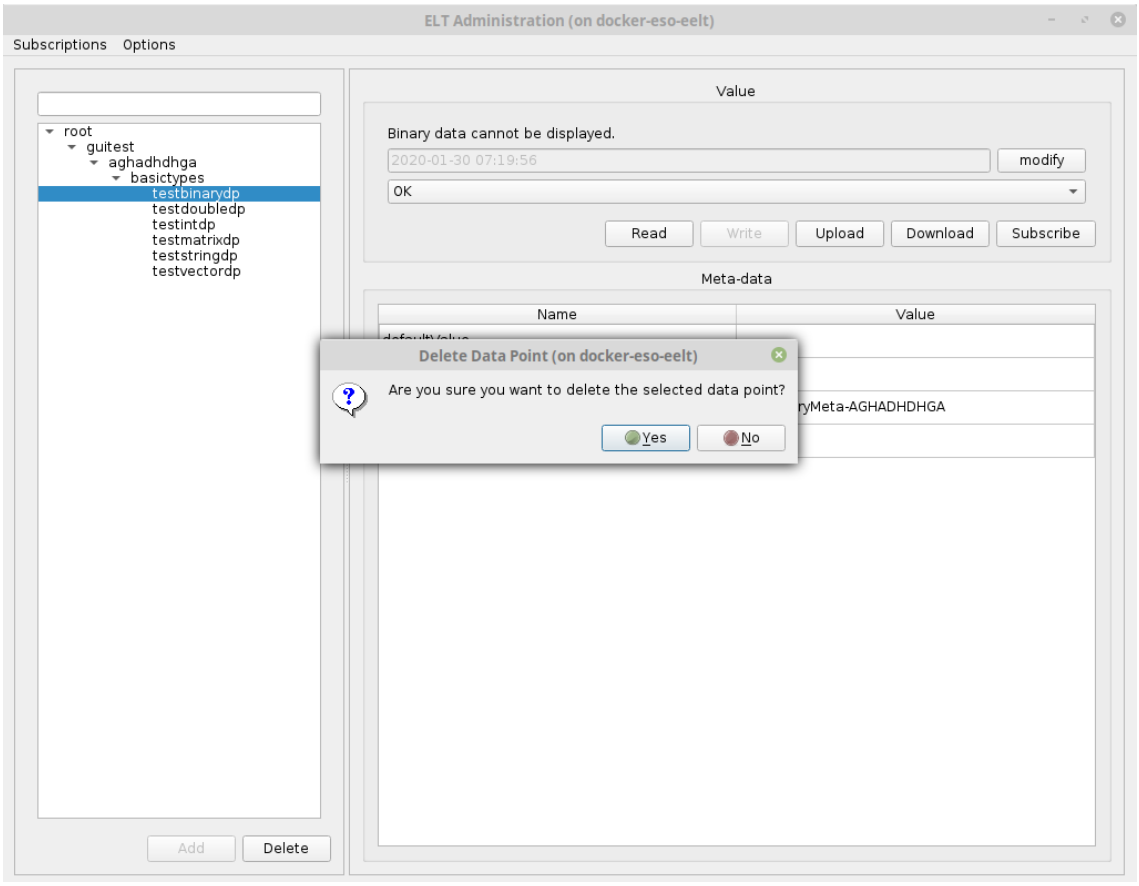


Figure 8-6: Data point deletion confirmation dialog

1. **Data Point Value and Metadata Frame**

The data point value and metadata frame occupy the right side of the application window. It displays the value and all the metadata values for the selected data point. It enables the user to do the following actions:

- Refresh data point values.
- Update data point values.
- Subscribe to data point value changes.

5. Manually Refreshing Data Point Value

The manual refreshing of the value for the selected data point is triggered by clicking the Read button just below the area for displaying the data point value. After doing this, a request is sent to a remote server, the most recent data point value is retrieved and displayed in the GUI.

6. Setting the Data Point Value

There are 3 groups of data point values:

- primitive values: the setting of the primitive value for the data point is performed using the input data dialog (Figure 8-7) that is displayed when clicking the modify button next to the text field for displaying the data point value. The dialog contains a simple input field where the user can input the data value. After the value is entered and the dialog is closed (by clicking OK), the Write button must be clicked for value to be written to the OLDB storage.
- vector & matrix values: they are displayed in a tabular form (Figure 8-8). The table values are set by clicking different table cells. After doing this, the same dialog displays as with the primitive values. Here, the user inputs the value for the table cell. In order for the modified values to be sent to the remote service, the Write button needs to be clicked.
- binary data values: the binary data values don't get displayed in the OLDB GUI. It must be stored to or loaded from the local disk.

Whenever the user modifies the data point value in the GUI the timestamp field updates to "now", so that when Write button is clicked the timestamp written to the OLDB is of the last change. The user can also modify the timestamp field in the same manner than the value.

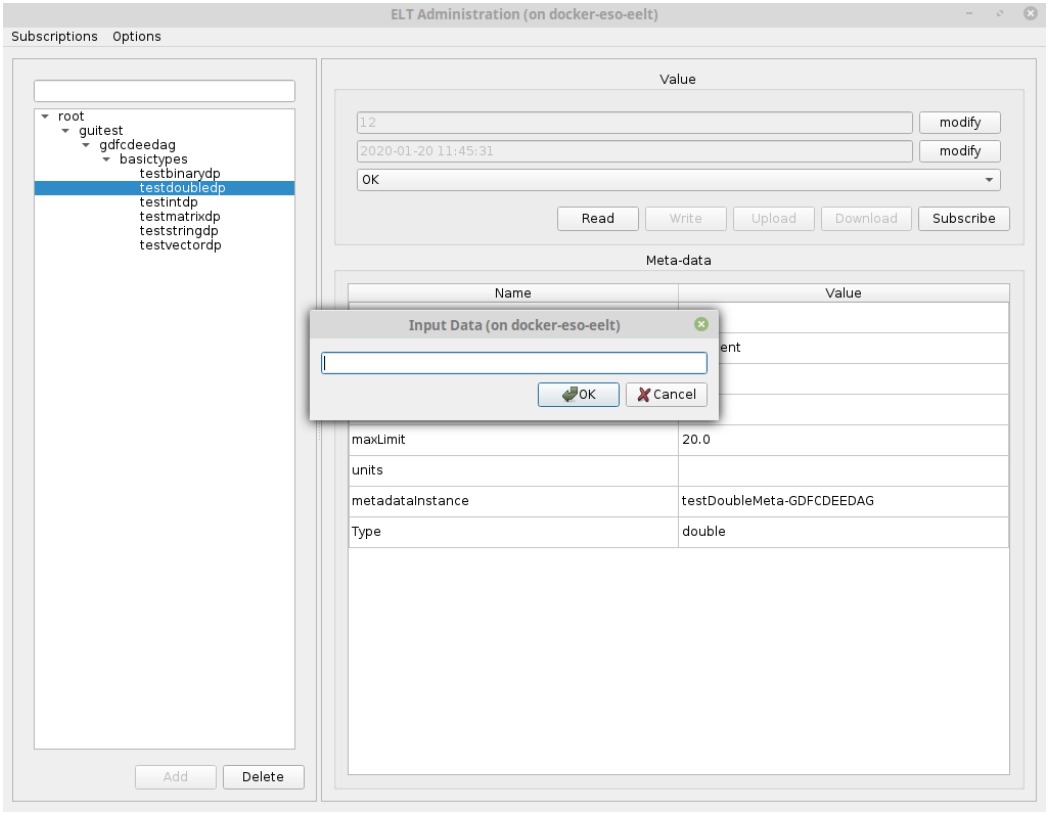


Figure 8-7: Data input dialog

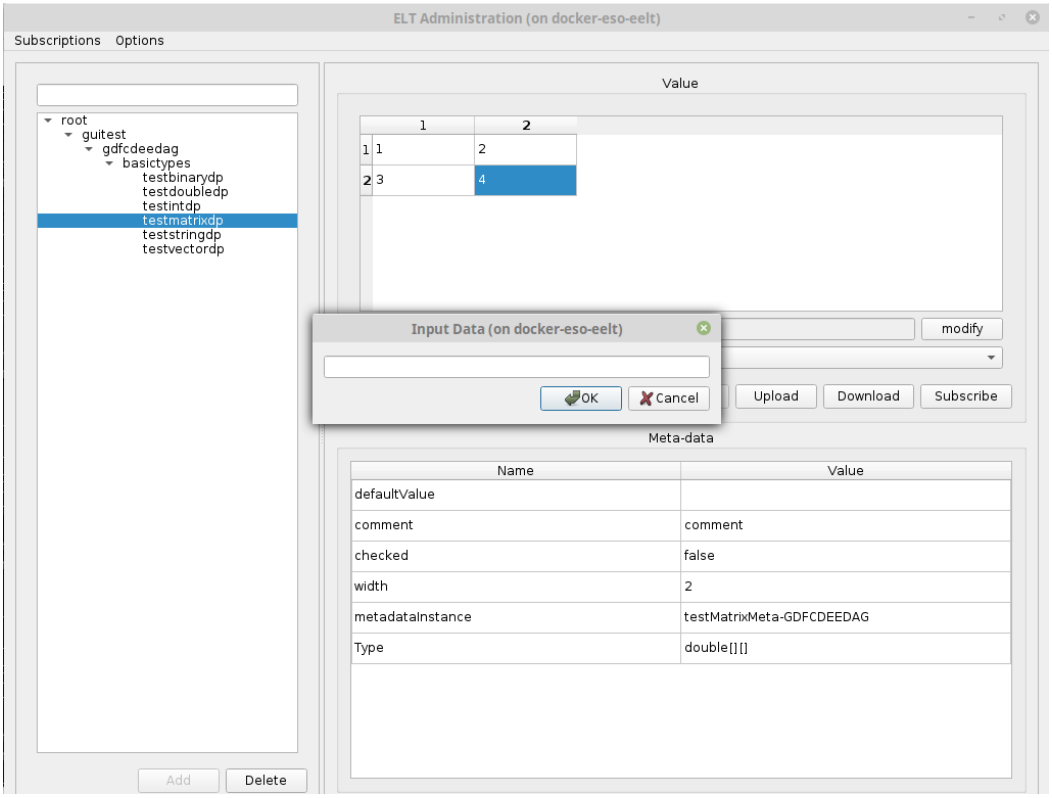


Figure 8-8: Vector & matrix data point value

The vector & matrix values as well as binary data values can also be loaded from a local disk and be stored to a local disk:

- the loading from a local disk can be started by clicking the Upload button. In the dialog that gets shown, the user selects the file on local disk, from which the data point values will be loaded. After the file has been loaded it can be sent to the remote service by clicking the Write button.
- the storing to the local disk can be started by clicking the Download button. In the dialog that gets shown, the user selects the file on local disk, to which the data point values will be stored.

For the vector & matrix data the user can also change the number of rows and columns. The row-and-column-setting dialog (Figure 8-10) is displayed by right-clicking the table and selecting the Resize table item from the context menu (Figure 8-9). If the matrix size is made smaller, any values already entered will be deleted for the removed columns or rows. If the matrix is made larger, the additional cells are empty.

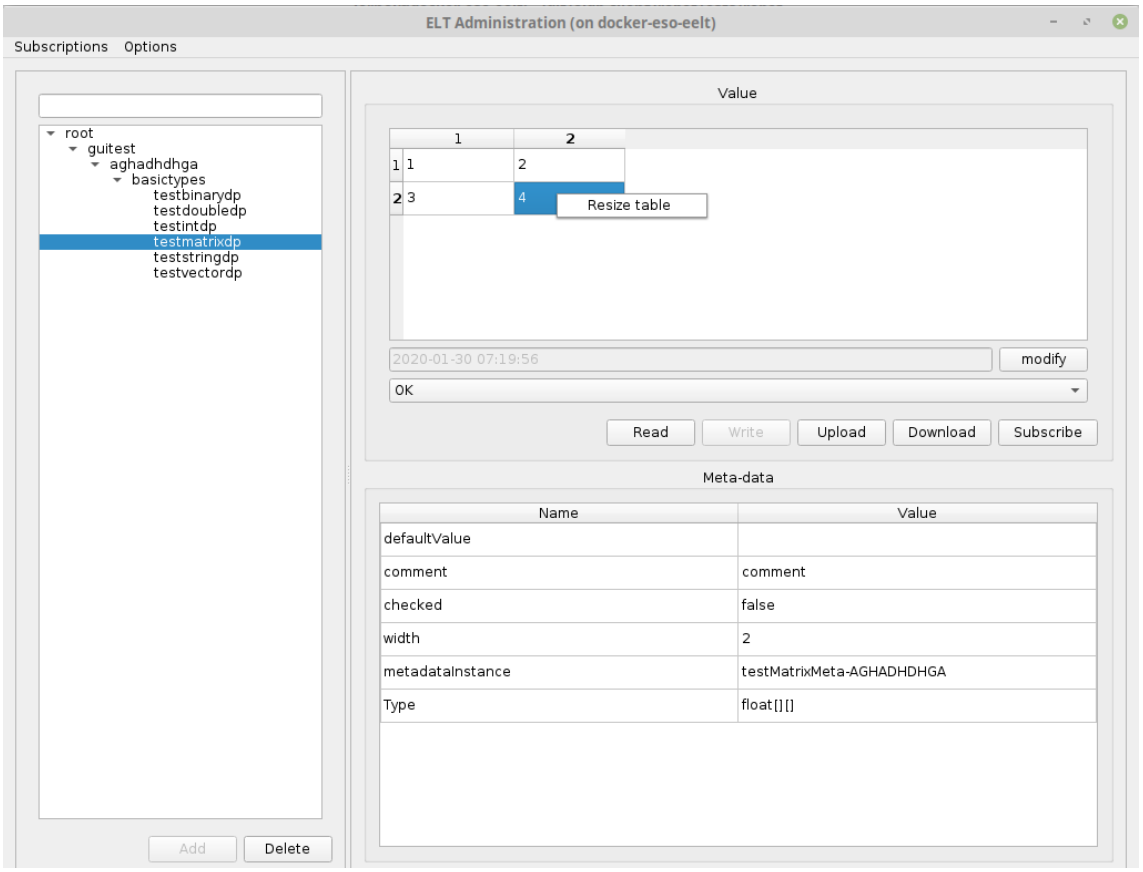


Figure 8-9: “Resize table” context menu

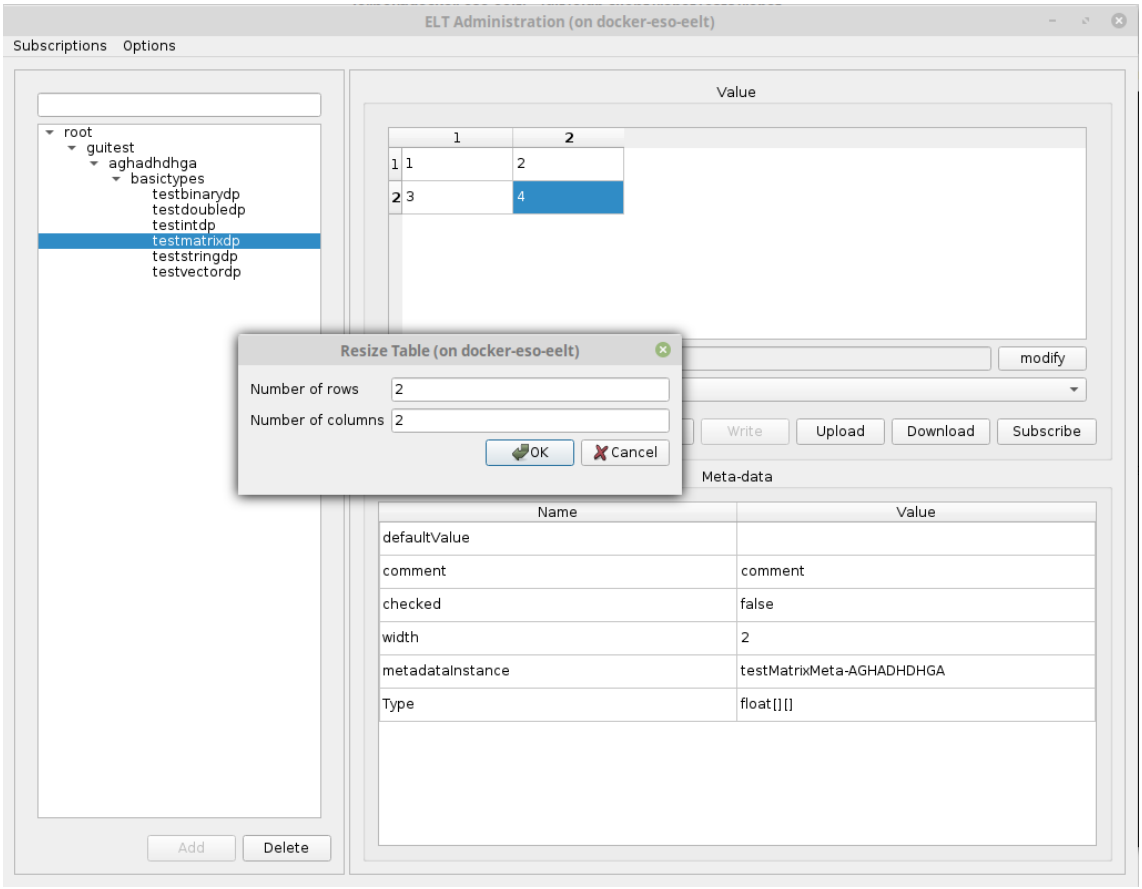


Figure 8-10: Dialog for setting the number of rows and columns.

7. Subscribing to Data Point Value Changes

Subscribing to the value changes of the selected data point is performed by clicking the Subscribe button that is located under the widget for displaying data point value. At this point the new window dialog is displayed that displays the most recent value of the given data point. Whenever the data point value changes on the server it is automatically updated in the subscription dialog. Multiple subscription dialogs can be displayed at the same time, meaning that the user can monitor the values of multiple data points. For a given data point, however, only a single subscription window can be displayed, having duplicate subscription windows for the same data point is not possible.

The user can organize the subscription windows in any way they like, they can even store the position of the subscription windows into a file. Later, the user can use the file to restore the windows. This enables the user to have a pre-defined sets of data points that they can monitor (see section 8.3.2 on how this is done).

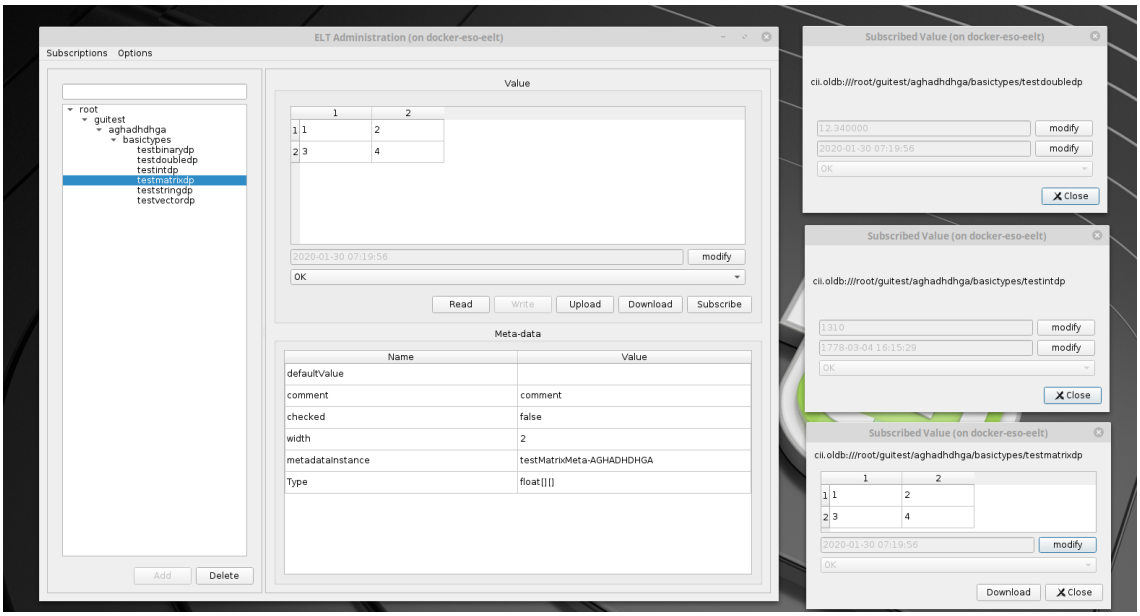


Figure 8-11: Arrangement of multiple subscription windows on the screen

7.9 Data Point Value Type to Language Types Mapping

Below is the mapping table between data point types and data types of all programming languages that the OLDB can use.

Table 8-1: Data point types mapping table

ELT CII BASIC TYPE	Java	C++		Python
INT8	Byte	std::int8_t		int
INT16	Short	std::int16_t		int
INT32	Integer	std::int32_t		int
INT64	Long	std::int64_t		int
UINT8	Byte	std::uint8_t		int
UINT16	Short	std::uint16_t		int
UINT32	Integer	std::uint32_t		int
UINT64	Long	std::uint64_t		int
SINGLE	Float	float		float
DOUBLE	Double	double		float
BOOLEAN	Boolean	bool		bool
STRING	String	std::string		str
BINARY	java.nio.ByteBuffer	std::vector<std::uint8_t>		bytes
VECTOR_INT32	List<Integer>	std::vector<std::int32_t>		list
VECTOR_SINGLE	List<Float>	std::vector<float>		list
VECTOR_DOUBLE	List<Double>	std::vector<double>		list
VECTOR_STRING	List<String>	std::vector<std::string>		List
MATRIX2D_INT32	List<Integer>	std::vector<std::int32_t>		list
MATRIX2D_SINGLE	List<Float>	std::vector<float>		list
MATRIX2D_DOUBLE	List<Double>	std::vector<double>		List
YAML		elt::oldb::datatypes::YamlNode		elt.oldb.datatypes.YamlNode

7.10 Default Metadata Instance Names

ELT CII BASIC TYPE	Metadata Instance Name	Metadata Class name
INT8	OldbInt8Std	MdOldbNumber
INT16	OldbInt16Std	MdOldbNumber
INT32	OldbInt32Std	MdOldbNumber
INT64	OldbInt64Std	MdOldbNumber
UINT8	OldbUInt8Std	MdOldbNumber
UINT16	OldbUInt16Std	MdOldbNumber
UINT32	OldbUInt32Std	MdOldbNumber
UINT64	OldbUInt64Std	MdOldbNumber
SINGLE	OldbSingleStd	MdOldbNumber
DOUBLE	OldbDoubleStd	MdOldbNumber
STRING	OldbStringStd	MdOldbString
BINARY	OldbBinaryStd	MdOldbBinary
VECTOR_INT32	OldbInt32ArrayStd	MdOldbArray
VECTOR_SINGLE	OldbFloatArrayStd	MdOldbArray
VECTOR_DOUBLE	OldbDoubleArrayStd	MdOldbArray
VECTOR_STRING	OldbStringArrayStd	MdOldbArray
MATRIX2D_INT32	OldbInt32MatrixStd	MdOldbMatrix
MATRIX2D_SINGLE	OldbFloatMatrixStd	MdOldbMatrix
MATRIX2D_DOUBLE	OldbDoubleMatrixStd	MdOldbMatrix
YAML	OldbYamlStd	MdOldbYaml

7.11 Metadata attribute constraints

METADATA ATTRIB.	CONSTRAINT	CONSTRAINT VALUE
Comment	Max size in characters	80 chars

7.12 YAML Data point type

Inclusion of the support for YAML data point type was requested with ECII-702.

It was requested that OLDB should support data points that should accept Yaml::Node data type defined by yaml-cpp.

Directly mapping Yaml::Node datatype was not feasible as it is unknown to Python, therefore intermediate data type elt::oldb::datatypes::YamlNode (this is actually an alias to elt::config::datatypes::YamlNode) was defined in C++.

Datatype internally serialises to yaml document (string). Python binding for elt::oldb::datatypes::YamlNode is available as elt.oldb.datatypes.YamlNode.

YamlNode can be initialized in one of the following ways in C++:

```
#include <datatypes/yamlNode.hpp>
...
// Empty yaml document
auto node = elt::oldb::datatypes::YamlNode();
// Initialize with valid yaml document source. When argument cannot be
// parsed as valid yaml, elt::config::CiiValue is thrown
node = elt::oldb::datatypes::YamlNode("[this, is, yaml, list]");
// Initialize with YAML::Node argument
YAML::Node source = ...
node = elt::oldb::datatypes::YamlNode(source);
// Extract yaml document as string
std::string doc = node.AsString();
// Extract yaml document as YAML::Node
YAML::Node ydoc = node.AsYamlObject();
// Assignment of new value from string
node = "[this, is, another, yaml, list]";
// Assignment of new value from YAML::Node
node = ydoc;
```

Python bindings replicate and extend C++ functionality:

```
from elt.oldb.datatypes import YamlNode
# Empty yaml document
node = YamlNode()
# Initialize yaml document from string containing valid yaml. Again␣
↪CiiValueError is raised in # case argument does not contain valid yaml.
node = YamlNode('[10, 20, 30, 40]')
# Initialize yaml document from python object
node = YamlNode(['this', 'is', 'python', 'list'])
# Update value with new yaml document source
node.set_source('[90, 20, 30, 40]')
# Update yaml node with new python object
node.set_value([90, 20, 30, 40])
# Extract yaml document source as string
```

(continues on next page)

(continued from previous page)

```
s = node.as_string()
# Extract yaml document as python object
assert(node.as_value(), [90, 20, 30, 40])
```

Apart from the fact that YamlNode data type is not native and must be initialized in the one of the ways listed above, OLDB API supports data points that contain this type.

Vectors of YamlNode elements are not supported by OLDB.

7.13 OLDB API LISTING

This a listing of OLDB API methods with short description for both CiiOldb and CiiOldbDataPoint classes. The method signatures are written in Java. The signatures are equal (with some exceptions in some of createDataPoint methods) in all three languages except for the native data types of each language (e.g. for string, double, integer). For data type mapping between languages see Table 8-1. For URI variables java uses java.net.URI, C++ uses <mal/utility/Uri.hpp> and Python uses elt.config.Uri.

Note that all OLDB API methods are synchronous (blocking) methods.

For detail explanation of exceptions thrown by these methods, consult the oldb-client source code.

7.13.1 CiiOldb

static CiiOldb getInstance()
The method returns the singleton instance implementation of the OLDB client CiiOldb.

CiiOldbDataPoint<?> createDataPoint(URI uri, String metadataInstanceName, String metadataClassName)
The method creates a data point at the defined URI with defined metadata. The data point data type and other restrictions are defined by its metadata which must already exist. Once the data point is created its data type cannot be changed. Note: This method signature is different in C++ and Python where the String metadataClassName parameter is omitted. This is because of Java language specifics.

CiiOldbDataPoint<String> createDataPointByValue(URI uri, String value)
Creates The method creates a data point of type STRING at the defined URI with defined value. It uses the default OLDB string metadata with metadata instance name “OldbStringStd”. The metadata must already exist.

CiiOldbDataPoint<Byte> createDataPointByValue(URI uri, Byte value)
Creates The method creates a data point of type INT8 at the defined URI with defined value. It uses the default OLDB string metadata with metadata instance name “OldbInt32Std”. The metadata must already exist.

CiiOldbDataPoint<Short> createDataPointByValue(URI uri, Short value)
Creates The method creates a data point of type INT16 at the defined URI with defined value. It uses the default OLDB string metadata with metadata instance name “OldbInt16Std”. The metadata must already exist.

CiiOldbDataPoint<Integer> createDataPointByValue(URI uri, Integer value)
Creates The method creates a data point of type INT32 at the defined URI with defined value. It uses the default OLDB string metadata with metadata instance name “OldbInt32Std”. The metadata must already exist.

CiiOldbDataPoint<Float> createDataPointByValue(URI uri, Float value)
Creates The method creates a data point of type SINGLE at the defined URI with defined value. It uses the default OLDB string metadata with metadata instance name “OldbSingleStd”. The metadata must already exist.

CiiOldbDataPoint<Double> createDataPointByValue(URI uri, Short value)
Creates The method creates a data point of type DOUBLE at the defined URI with defined value. It uses the default OLDB string metadata with metadata instance name “OldbDoubleStd”. The metadata must already exist.

CiiOldbDataPoint<ByteBuffer> createDataPointByValue(URI uri, ByteBuffer value)
Creates The method creates a data point of type BINARY at the defined URI with defined value. It uses the default OLDB string metadata with metadata instance name “OldbBinaryStd”. The metadata must already exist.

CiiOldbDataPoint<List<T>> createDataPointByValue(URI uri, List<T> value, Class<T> clazz, boolean isMatrix)
Creates The method creates a data point of matrix or vector type at the defined URI with defined value. It uses the default OLDB string metadata with metadata instance name “Oldb<element-type>ArrayStd”. The metadata must already exist. The <element-type> is “Int32”, “Single”, “Double” or “String” (vector only). The isMatrix parameter defines whether a matrix or vector data point is created. The clazz parameter defines the class of the elements of matrix/vector. Note: The Class<T> clazz parameter is only needed in Java version of the method because of language specifics.

CiiOldbDataPoint<List<T>> createDataPointByValue(URI uri, List<T> value, Class<T> clazz)
Creates The method creates a data point of vector type at the defined URI with defined value. It uses the default OLDB string metadata with metadata instance name “MdO<element-type>ArrayStd”. The metadata must already exist. The <element-type> is “Int32”, “Single”, “Double” or “String”. The clazz parameter defines the class of the elements of vector. Note: The Class<T> clazz parameter is only needed in Java version of the method because of language specifics.

void deleteDataPoint(URI uri)
The method deletes a data point referenced by the URI.

Map<URI, Boolean> getChildren(URI uri)

Finds all direct children of the given URI. Returns the map of all found URI with boolean value indicating whether the URI represents a data point (true if data point, false if only folder). Since OLDB allows to create a data point with an URI that is a sub-path of another URI, there can be both folder and data point URIs present. In that case the boolean value is true.
Note: Because OLDB is case insensitive all URIs returned will be in lower case.
Example
Consider the following OLDB URIs:
`cii.oldb:///root/child/device/dp1`
`cii.oldb:///root/child/dp2`
`cii.oldb:///root/child/dp2/dp3`
A call to method getChildren with URI = cii.oldb:///root/child will return the map in Table 8-2.
Table 8-2: Example of getChildren call result

Key	Value
cii.oldb:///root/child/device	FALSE
cii.oldb:///root/child/dp2	TRUE

Note that path cii.oldb:///root/child/dp2 hierarchically represents both data point and a folder. In this case the getChildren returns TRUE for this path. This means that a TRUE for an URI does not mean that the URI doesn't also represent a folder.

CiiOldbDataPoint<?> getDataPoint(URI uri)
Returns a data point at a specified URI. Note that the caller of the getDataPoint method must know the data point type to cast the received object to the right type (only required for the strongly typed languages like C++). The type of the data point can be obtained from metadata (Section 6.2.3). Note: this method does not provide transactional safety. There is no guarantee that the data point is not deleted the moment after it was retrieved (data point object of a deleted data point is not usable).

List<CiiOldbDataPoint<?>> getDataPoints(List<URI> uris)
The method returns a list of data points matching the specified URIs. The URIs in the list can be written as a simple glob pattern using three special characters. * for any number of characters restricted to the URI hierarchy level ** for any number of characters spanning multiple levels ^ for exactly one character Example: If we have a simple OLDB that contains data points with the following URIs: cii.oldb:///root/child/device/dp1 cii.oldb:///root/child/device/dp21 cii.oldb:///root/child/device/dp23 cii.oldb:///root/child1/motor/dp2 Then calls to getDataPoints with these glob patterns would result in following data points: cii.oldb:///root/child/device/dp2* -> dp21, dp23 cii.oldb:///root/child/** -> dp1, dp21, dp23 cii.oldb:///root/child/device/dp^ -> dp1 Note: this method does not provide transactional safety. There is no guarantee that a data point is not deleted the moment after it was retrieved (data point object of a deleted data point is not usable).

List<CiiOldbDataPoint<T>> getDataPoints(List<URI> uris, AttributeType dataType, T minValue, T maxValue)
The method returns a list of data points matching the specified URIs, filtered by type and min and max limits. As with the getDataPoints above, this method supports the simple glob patterns. Note: this method does not provide transactional safety. There is no guarantee that a data point is not deleted the moment after it was retrieved (data point object of a deleted data point is not usable).

void setWriteEnabled(boolean writeEnabled)
Enables or disables all writes to data points.

void close();
Closes connections and cleans up resources.

Below are the C++ CiiOldb API listings. The Python client uses Python bindings and has therefore the same API as the C++ client.

Listing 8-2: C++ CiiOldb API

```
std::shared_ptr<CiiOldbTypedDataBase> CreateDataPoint(const ::elt::mal::Uri& uri,
↪ const std::string& meta_data_instance_name);
```

(continues on next page)

(continued from previous page)

```
template<typename T>
 std::shared_ptr<CiiOldbDataPoint<T>> CreateDataPointByValue(
const ::elt::mal::Uri& uri, const T& value);

template<typename T>
std::shared_ptr<CiiOldbDataPoint<std::vector<T>>> CreateDataPointByValue(
const ::elt::mal::Uri& uri, const std::vector<T>& value, bool is_matrix);

std::shared_ptr<CiiOldbTypedDataBase> GetDataPoint(const ::elt::mal::Uri& uri)␣
↪const;

template<typename T> std::shared_ptr<CiiOldbDataPoint<T>> GetDataPoint(
const ::elt::mal::Uri& uri, ::elt::common::CiiBasicDataType data_type) const;

std::vector<std::shared_ptr<CiiOldbTypedDataBase>> GetDataPoints(
const std::vector<::elt::mal::Uri>& uris) const;

template<typename T> std::vector<std::shared_ptr<CiiOldbDataPoint<T>>>␣
↪GetDataPoints(const std::vector<::elt::mal::Uri>& uris,
::elt::common::CiiBasicDataType data_type, T min_value, T max_value) const;

std::map<::elt::mal::Uri, bool> GetChildren(const ::elt::mal::Uri& uri) const;

static void SetWriteEnabled(bool enabled) noexcept;

static bool IsWriteEnabled() noexcept;

void DeleteDataPoint(const ::elt::mal::Uri& uri);
```

7.13.2 CiiOldbDataPoint

CiiOldbDpValue<T> ReadValue(bool check_bad_quality = true)
Returns a copy of this Data Point value. Every call to ReadValue retrieves an up to date value from the OLDB, but the CiiOldbDpValue object is a snapshot at the time of retrieval and does not update automatically. The user should call ReadValue to get the updated value. To retrieve a datapoint with a not OK quality, 'check_bad_quality' argument should set to false.

void WriteValue(T value, int timestamp = Now(), bool is_disable_publishing = false)
Writes the value to the data point. Timestamp is UTC in nanosecs since epoch,The quality is unchanged and timestamp is by default now.

void WriteValue(T value, int timestamp, CiiOldbDpQuality quality, bool is_disable_publishing = false, string overridden_server_alias = nullptr)
Writes all three fields (value, time and quality) to the data point. Timestamp is UTC in nanosecs since epoch. 'is_disable_publishing' is used in case the user only need a notification and not a the new value. 'overridden_server_alias' allows changing the data point server configuration and transfering the value from default data provider to external or vice versa

void WriteValue(T value, int size, int timestamp = Now())
Writes the value with a specific istream size and timestamp to the data point. The quality is unchanged. Timestamp is UTC in nanosecs since epoch.

void Subscribe(CiiOldbDpSubscription listener, bool answer_immediately = false)
Subscribes an object listener to the data point to listen to value changes. NOTE: when answer_immediately = true, meaning that the user provided listener will be called immediately with the data point value, the Subscribe call will be slower for the bigger data points since the whole value needs to be read, otherwise only a "header" for the data point is read.

void Unsubscribe(CiiOldbDpSubscription listener)
Unsubscribes an object listener from the data point

void setQuality(CiiOldbDpQuality quality)
Sets the data point quality.

<U extends MdOldb<T>> U getMetadata(Class<U> clazz)
Returns the metadata of the data point. Caller must provide the metadata class (extends MdOldb).

<U extends MdOldb> void setMetadata(String metadataInstanceName)
Updates the data point metadata with the metadata which instance name is provided. The changes made in the metadata must be saved to config service otherwise this call will have no effect. If the instance name is different from the one this data point already has (new metadata instance name has been assigned to the data point), the new metadata must respect the data point type.

Below are the C++ CiiOldbDataPoint API listings. The Python client uses Python bindings and has therefore the same API as the C++ client.

Listing 8-4: C++ CiiOldbDataPoint API

```
std::shared_ptr<CiiOldbDpValue<T>> ReadValue(bool check_bad_quality = true);

void WriteValue(const T& value, int64_t timestamp = CiiOldbUtil::Now());

void WriteValue(const T& value, int64_t timestamp, CiiOldbDpQuality quality,
const std::string *overridden_server_alias = nullptr,
const std::size_t *overriden_value_size_limit_hdfs = nullptr);

void WriteValue(std::istream& value, std::uint64_t size, int64_t timestamp =␣
↪CiiOldbUtil::Now());

void WriteValue(std::istream& value, std::uint64_t size, int64_t timestamp,
CiiOldbDpQuality quality);

void Subscribe(const std::shared_ptr<CiiOldbDpSubscription<T>>& listener);

void Unsubscribe(const std::shared_ptr<CiiOldbDpSubscription<T>>& listener);

std::shared_ptr<::elt::config::classes::meta::MdOldb<T>> GetMetadata() const;

void SetMetadata(const std::string& meta_data_instance_name);

void SetQuality(CiiOldbDpQuality quality) override;

elt::common::CiiBasicDataType GetType() const override;
```

CHAPTER

EIGHT

ERROR HANDLING

Document ID:	
Revision:	1.3
Last modification:	March 18, 2024
Status:	Released
Repository:	https://gitlab.eso.org/cii/info/cii-docs
File:	error_api.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
0.9	9.4.2019	bterpinc	All	Created.
1.0	16.9.2019	bterpinc	All	Updated after internal review.
1.1	10.2.2020	bterpinc	All	Major update of all sections after Topical review comments.
1.2	20.11.2020	bterpinc	2.2	Added explanation about the error severity.
1.3	18.03.2024	mschilli	0	Public doc

Confidentiality

This document is classified as Public.

Scope

This document is a user manual for the error handling system of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

Glossary of Terms

API	Application Programmers Interface
CII	Core Integration Infrastructure
GUI	Graphical User Interface
ICD	Interface Control Document
IDE	Integrated development environment
MAL	Middleware Abstraction Layer
SVN	Subversion version control system

References

1. Cosylab, Interface Control Document, Specification, CSL-DOC-17-147262, version 1.3
2. Cosylab, Error handling design document, CSL-DOC-17-167386, version 1.2
3. ELT Config, Error Handling Transfer document, CSL-DOC-19-172644, version 1.5
4. ElasticSearch Query DSL, https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html
5. ElasticSearch, Standard Analyzer, https://www.elastic.co/guide/en/elasticsearch/reference/6.8/analysis-standard-analyzer.html

8.1 Overview

This document is a user manual for CII Error handling system.

8.2 Introduction

Error Handling in CII provides the APIs and tools necessary to conveniently implement a common error handling mechanism across all EELT Control System applications and thus streamline the diagnosis of abnormal behavior of the system.

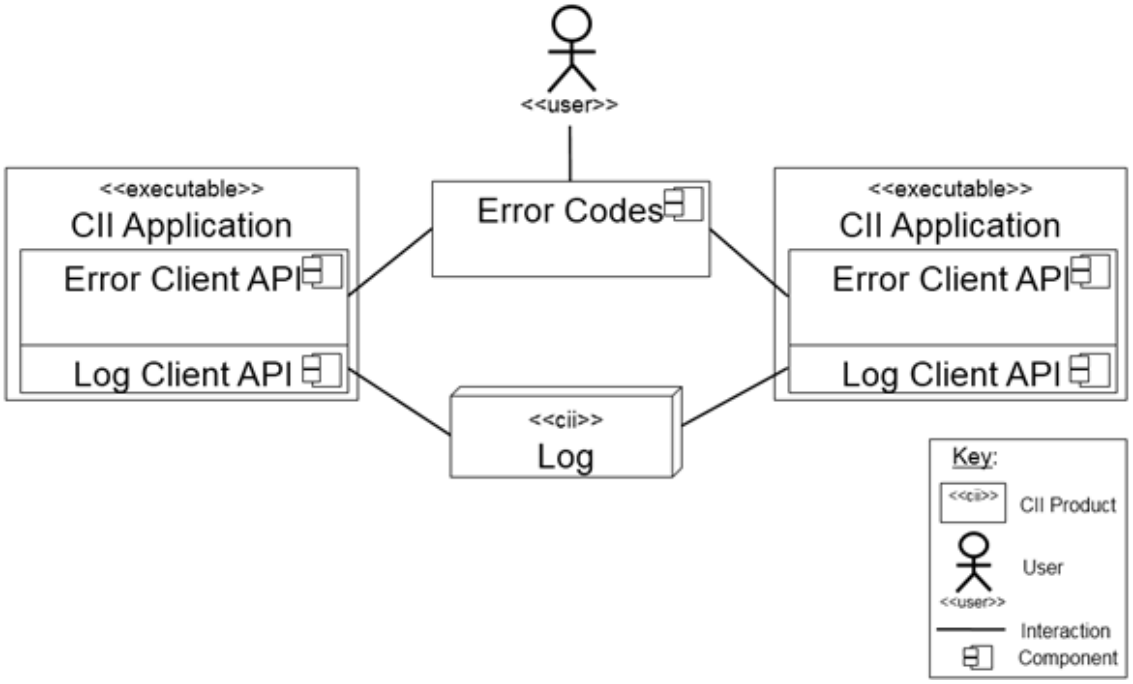


Figure 2-1 Interaction diagram of error handling

In general, error handling covers the following error management activities:

Error detection: language-specific try/catch mechanisms provide error detection.

Error handling: errors are based on exceptions. The name of the exception with the namespace defines the error type and error severity. Catching errors is based on the try/catch mechanism of each language. Error grouping is supported by exception inheritance. Error message and description is a part of exception class.

Propagation of error information: Error information is contained inside the exception entities. This information is passed between the different program parts. Passing information over the network is realized with ICD-based exceptions.

Administration and collection of all-important information: Definition of the error information is contained inside the exception class. New classes can be added and changed and it is also possible to change existing exception classes. All error information can be logged to the logging system.

Error messages will be displayed to the user: A QT GUI error display widget provides a GUI component for displaying the error messages. In the case of an error, the error message is displayed in a pop-up dialog.

All components of the error handling are provided for 3 languages: Java, CPP and Python.

8.2.1 Standard workflow

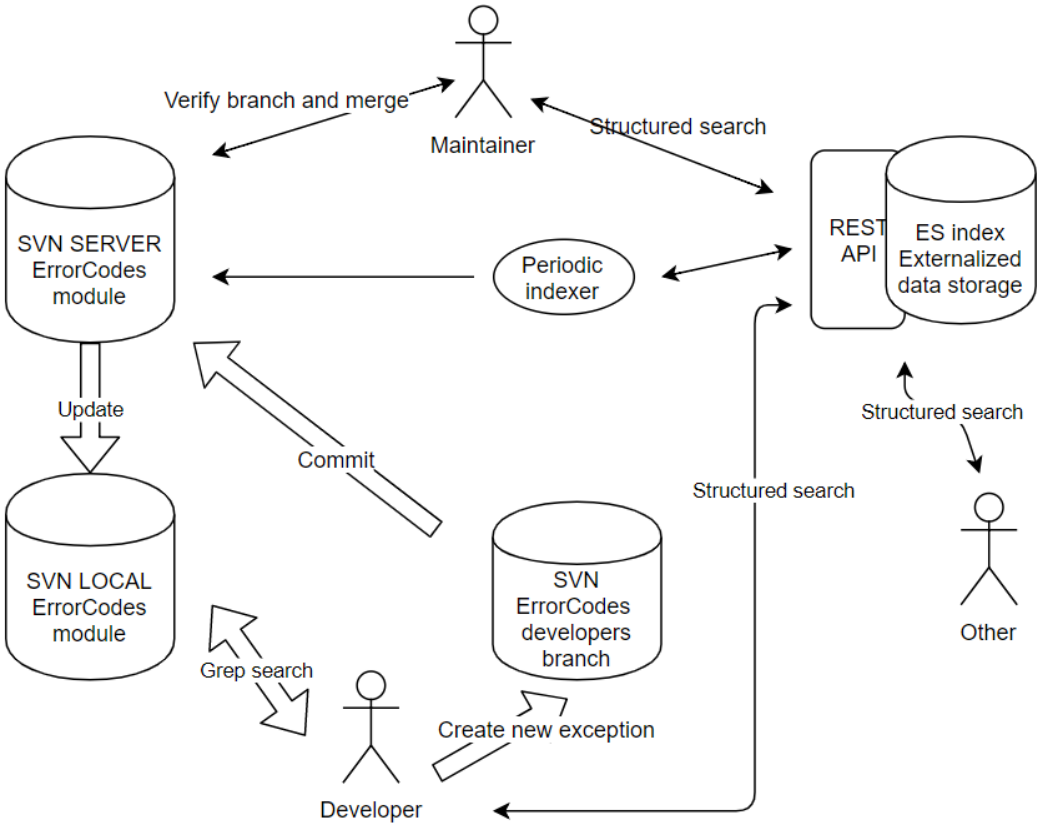


Figure 2-2 Error code repository and workflow

Figure 2-2 presents the typical workflow for storing the exception information in a central repository. It is advised to have one or more general maintainers (lead developers) of the error code system. The maintainer is responsible for the error code system integrity and has an overview of all the exceptions in the system (or a specific section).

Whenever the developers need a specific exception, they first checkout the latest version of the SVN module where needed exception is located. They then use *grep*, *find* or any other file text search tool to find the desired exception. They can also browse the folder tree to find the exceptions that belong to specific groups. A helper search script *searchExceptions.sh* is provided to aid the developer when searching. If a developer cannot find the needed exception, one creates a new exception that is stored in a separate development branch. The new exception can be used immediately in the development branch. Then the developer commits the proposed new exception to central storage in the separate branch. The developer informs the maintainer about the new exception. The maintainer reviews the new exception and either merges the exception into the trunk or proposes the use of an existing exception from the trunk to the developer. When a new exception is merged into the system, the

maintainer sends the message to the developer. The developer checks out the latest version of the *trunk* and uses the new exception from the *trunk*.

If a change to the exception information is needed, the same process as for creating an exception is used; the only difference is that the existing exception is modified and stored to the development branch. Subsequent steps are identical.

8.2.2 CII Exceptions

The CII error handling mechanism is based on CiiException, which is the base exception from which all other exceptions are derived. The CiiException derives from the special ICD defined CiiSerializedException, which provides serialization (2.4). Derived from the CiiException, two additional exceptions are defined as the part of the error handling system: CiiError and CiiBaseException. These two exceptions correspond to the error severity:

- Normal - CiiBaseException – these exceptions cover the errors which can be recovered, and can be handled at higher software layers.
- Error - CiiError – these exceptions cover the unrecoverable errors. When unrecoverable error happens, the program shall exist with the corresponding error message.

The severity of the error is defined by the exception name and the parent exception from which all errors of the selected severity are defined. The exception and error severities are pre-defined with exception names. All exceptions defined for severity »error« must contain the word error in the exception class name. All exceptions with »normal« severity must contain the word exception in the class name. Exceptions that define error severity are based on CiiError exception and exceptions with normal severity must derive from CiiBaseException.

All user defined exceptions must derive from the CiiBaseException or from CiiError.

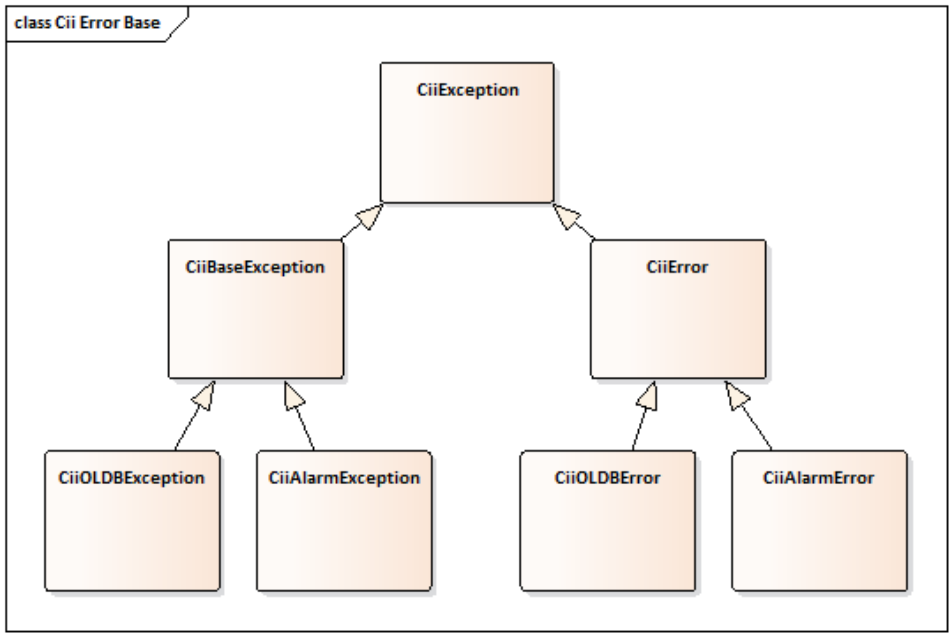


Figure 2-3 CII base exceptions and errors

Figure 2-3 shows the class inheritance diagram of the base errors and exceptions. The exceptions defined by the particular CII service all derive from the CiiBaseException for »normal« severity and from CiiError for »error« severity. Every CII service also defines root exceptions for the two severities (e.q., CiiOLDBException, CiiOLDBError). From these two exceptions, all the other exceptions for a particular system are derived.

Error stack

Single exception instance can nest exceptions and create an error stack. The stack is created by construction exceptions with the included information about the nested exception. A hierarchy of nested exceptions constructs the exception stack. Every nested exception presents an entry to the exception error stack. Exceptions (entries to the error stack) can only be added to the error stack, they cannot be changed or deleted. The error stack is automatically removed when reference to the exception with all the nested exceptions is lost.

In C++, a special throw mechanism must be used to create a stack of errors.

Whenever there is a need for the stack to be logged, an explicit entry to the log system must be programmed by the developer.

The examples of creating the exception stack can be found in 4.6.1.

Indexing Service

The Indexing service is a Java-based program that collects information about the exceptions and stores them in the Elasticsearch. The Indexing Service will traverse the SVN directory where exceptions are stored and crawl over the source code to extract the following information from the exceptions:

error type,

error message,

error description,

author.

Extraction uses the flowing rules:

The error type is extracted from the exception name and folder path.

The error description is extracted from the class comment. The description starts with the *@desc* tag.

The author of the exception is extracted from the class comment. It starts with the *@author* tag.

The error message is extracted from the exception class member variable named message. The variable is of type string.

The indexing service expects all information other than the description to be stored in one line.

To provide namespace information for the error type, the directory structure information is used. As the exceptions must be properly structured in the named folders that reflect the namespace, this information will provide the proper information for the error type namespace.

8.2.3 Conditions API

Error handling also provides conditions API. The purpose of the API is to provide a convenience API to enable checking of pre and post conditions, and method invariants. The API also supports enabling and disabling conditions and fail-fast checks. API provides static methods for verifying arguments, null values, states and valid index positions in lists and arrays. The methods, in general, accept one or more arguments for checking the condition and an argument that defines the exception type. The API interacts with the error handling system.

Whenever the condition is not met, the API will throw an exception. API contains an option for setting the fail-fast mode. When the fail-fast mode is enabled, the language-specific root exception will be thrown. This will cause the application to exit with the defined error message.

The error data is centrally stored in SVN. Chapter A.3 contains a description of Conditions API methods.

8.2.4 Serialization

Transportation of the exceptions over the distributed processes, written in different programming languages is supported with exception serialization. Serialized exceptions are transferred over MAL, which provides serialization of exceptions into plain objects. Therefore, exceptions are passed over the network with information serialized to string.

MAL uses ICD schema to define the exception structure. Custom user-defined ICD exceptions with desired information can be created. For these exceptions, there is no mechanism to implicitly transfer the data from the CiiException based exceptions to the ICD exception. Exception data can be read from the CiiException based exceptions and used to fill the ICD user-defined member fields. When an ICD based exception is transferred over the network, it is the developer's responsibility to catch the exception and manage the data transfer and pass the exception to the higher levels of error handling.

For simplicity of use, the ICD CiiSerializedException is defined. This is a special ICD exception with hard-coded additions. This exception has predefined member fields, which carry the data of the exception. CiiSerializedException is also a base exception for all CiiExceptions. This special exception simplifies transferring exceptions over the network from the sender side, as no data copying between exceptions is needed. All CiiExceptions are derived from the CiiSerializedException, this means that CiiSerializedException will automatically catch all user-defined CiiExceptions. All exceptions that use CiiException will be automatically transferred over the network as CiiSerializedException.

8.3 Installation

Error Handling provides the following components that need to be installed:

- Error indexer
- Error handling dialog widget
- Conditions API

See Configuration and Error Handling Transfer Document [3] for instructions to install the error handling system.

8.3.1 Prerequisites and waf modules

Error Handling uses SVN as central storage for exceptions. Error Handling also uses Elasticsearch as an engine for more structured searching of the exceptions. The two products must be installed and accessible from the local hosts to use the Error Handling:

- SVN server,
- Elasticsearch.

Error handling uses the following CII modules:

- elt-mal: MAL, MAL ZPB.
- client-api: Conditions API, Error Handling API.
- srv-error-indexer: Error indexer, search script.
- elt-qt-widgets: Error handling dialog widget.

8.4 Usage

The example in this manual is based on CLI SVN interface. Standard language IDE's (Eclipse, IntelliJ Idea, QT Creator) can be used for writing exception definitions and to communicate with the SVN server. Examples of IDE use are not part of this manual.

8.4.1 Includes/Imports

For basic usage of the Error Handling, the following programming language imports must be added:

Java

```
import elt.error.CiiError;
import elt.error.CiiBaseException;
import elt.error.CiiConditions;
```

CPP

```
#include <CiiException.hpp>
#include <CiiConditions.hpp>
```

Python

```
import elt.error
```

QT

```
#include <CiiException.hpp>
#include <CiiExceptionDialog.hpp>
```

8.4.2 Creating an SVN repository

To use the error codes system an SVN repository must be created, which will act as a storage for all CII Exceptions. It is expected that this repository is already created by the lead developer or repository maintainer. The developer should consult the responsible person for the information about the location.

In this manual we will create a repository in an SVN server hosted at *https://svnServer*, at */pathToErrorCodes/errorCodes* path. The repository can be created with the following command:

```
svn mkdir -m "Create error codes directory" https://svnServer/pathToErrorCodes/
↪errorCodes --username myusername
```

In the *errorCodes* repository, the basic error codes system structure must be created. Folders can be created in the same manner as the root folder, or a local copy of structure can be prepared and committed to the server. The following structure must be created:

errorCodes/trunk/java

errorCodes/trunk/cpp

errorCodes/trunk/python

errorCodes/branches

Note: The root folder in SVN for the exceptions can be changed to the specific path (e.g. *p8/trunk/CONTROL-SYSTEMS/CentralControlSystem/CII/CODE/srv-error-indexer/sampleErrorCodeTestRepo/*). However, the structure of folders after the root folder is fixed and cannot be changed in order for Indexing service (6) to properly collect the error codes from SVN repository. Indexing service expects *trunk* folder and subfolders *java, cpp* and *python*. The names of the subfolder's are a part of indexing service source code. A small change in Indexing service code must be made to use the different names for the folder structure.

8.4.3 Creating the exceptions

Exceptions must be created in the folder reflecting the namespace and with a file name reflecting the exception class name. Multiple exceptions with the same name are allowed, if they are put in separate namespaces. Exception name together with the namespace defines the error type and severity.

The workflow of error codes is described in section 2.1. Shortly, a developer must create exceptions in his own branch, which is then checked by the maintainer and merged into the trunk.

Note: It is expected that the development environment was properly set and that the variable *$INTROOT* points to the root location where all the CII components are installed (command *"module load introot eeltdev"* was run).

If the SVN *errorCodes* folder is not present on local developer's computer, we first need to check-out the *errorCodes* directory from the SVN.

```
cd $INTROOT/sources/
svn co https://svnServer/pathToErrorCodes/errorCodes --username myusername
```

New exceptions, defined by a developer, should be created in their own branch. It is suggested to create this branch with a folder named after developer username. For each language a separate folder is required (java, cpp, python).

```
cd branches/
mkdir developerName
```

(continues on next page)

(continued from previous page)

```
cd developerName
mkdir java cpp python
```

Exceptions source code must be structured as shown in the following examples. Exception files must be placed in the folders, whose names reflect the namespace. Each single exception is placed in own file with the filename named the same as the exception class.

In the example below we create CiiFileHandlingException, which is in the *elt.error.common* package. We create this exception for all 3 languages. Folders for this package must be created:

```
mkdir -p java/src/elt/error/common
mkdir -p python/src/elt/error/common
mkdir -p cpp/elt/error/src/include
```

Information from the exceptions is parsed by the indexing service. In order for the indexing service to properly parse the exceptions, the exceptions must be placed in the described folders. Also, the structure of exceptions source code must follow the structure presented in the following chapters (4.3.1, 4.3.2, 4.3.3). Information about the author and description of the exception is parsed from the source code comments. The information about the exception message is parsed from the message string.

Examples of the exceptions SVN repository can be found in the *srv-error-indexer* module under the *sampleErrorCodesRepo* directory. The following sections show the programming language specific examples.

Java

To create a Java exception, create CiiFileHandlingException.java file inside the *java/src/elt/error/common* folder with the desired exception specifics. The following example shows the Java exception structure:

```
package elt.error.common

/**
 * @desc This is the description of file handling error.
 * @author Somebody@eso.org
 */

public class CiiFileHandlingException extends CiiBaseException {
  public static final String MESSAGE = "Error while handling file: %s";

  public CiiFileHandlingException(String path){
    super(String.format(MESSAGE, path));
  }
}
```

CPP

For CPP exceptions to be browsable, they must follow the Java style for exception filenames. Each exception is created in its own .hpp file, with the file named the same as the exception class name. Exception details (description and author) must be added inline in the header code. CPP exceptions are header only based, no .cpp files shall be created.

To create a CPP exception, create CiiFileHandlingException.hpp file inside the *cpp/elt/error/common/src/include* folder with the desired exception specifics. The following example shows the CPP exception structure:

```
/**
 * @desc This is the description of file handling error.
 * @author Somebody@eso.org
 */

namespace elt {
namespace error {
namespace common {

class CiiFileHandlingException : public CiiBaseException {

const std::string getMessage(){
  static const std::string& message = "Error while handling file: %s";
  return message;
  //other option for message definition
  //return std::string("Bad version number %i");
}
public:
  CiiFileHandlingException(const char* fileName):
                  CiiException(getMessage(), fileName){}
};

//other option std::string
//if the argument passed to the CII Exception is string it has to be converted␣
↪to const char*
/*
public:
  CiiFileHandlingException(const std::string& fileName):
                  CiiException(getMessage(), fileName.c_str()){}
}; */
}
}
}
```

Python

For Python exceptions to be browsable, they must follow the Java style for exception filenames. Each exception is created in its own .py file, with the file named the same as the exception class name.

To create a Python exception, create CiiFileHandlingException.py file inside the *python/src/elt/error/common* folder with the desired exception specifics. The following example shows the Python exception structure:

```
"""
@desc This is the description of file handling error.
@author Somebody@eso.org
"""
class CiiFileHandlingException(CiiBaseException):
  _message = "Error while handling file: %s"
  def __init__(self, arg1):
    super().__init__(_message % arg1)
```

After the exceptions are created in a separate branch, the developer commits the exceptions to the SVN server and informs the exceptions maintainer about the existence of new exceptions. The maintainer then merges the exception into the trunk.

It is the maintainer's responsibility to check that the exception structure (folders and namespace) conforms to the convention.

8.4.4 Wrapping native exceptions

As native language exceptions follow their own inheritance tree, they cannot be directly caught by the CiiException. For this reason, all the native exceptions must be caught and rethrown with the details passed to the variant of the CiiException. This way information provided by the native exceptions can be part of the CII Exception.

The following example shows how to pass the native exception information to CII Exceptions in Java:

```
try {
  testList.get(arrayIndex);
} catch (IndexOutOfBoundsException ex) {
   throw new CiiBadIndexException(arrayIndex, ex);
}
```

Same mechanism should be used in Python and CPP.

8.4.5 Search script

Search script **searchExceptions.sh** is a simple bash script providing capabilities of automatic SVN update, local search and remote search. The script is a wrapper script around the Linux *find* and *grep* commands. The script automatically checks out the latest version of *errorCodes* from the SVN server and executes a search. It also provides search for indexed exceptions in Elasticsearch. While the functionality of the search in Elasticsearch is practically the same as for other *find* and *grep* style searches, it provides more structured output with more detailed information about the exceptions.

The script is intended to be used locally by developers. All the settings are embedded directly in the script.

Configuring the script

To configure the script, open it in a text editor:

```
nano $INTROOT/bin/searchExceptions.sh
```

Change the following settings:

- REMOTE_REPO: location of the root of remote SVN repository
- SVN_USERNAME: username with SVN permissions
- SVN_PASSWORD: password for the SVN user
- LOCAL_REPO: location of the root of local SVN repository

Example:

```
REMOTE_REPO="https://svnhq8.hq.eso.org/p8/trunk/CONTROL-SYSTEMS/
↪CentralControlSystem/CII/CODE/srv-error-indexer/sampleErrorCodesRepo"
SVN_USERNAME="myusername"
SVN_PASSWORD="mypassword"
LOCAL_REPO="myLocalRepoRoot/sampleErrorCodesRepo"
```

Usage of the search script

A search without a special flag argument will perform a grep style search:

```
./searchExceptions.sh "Error while handling file:"
```

The search with the -f argument will execute a find style search:

```
./searchExceptions.sh -f "CiiFileHandlingException.java"
```

Direct Elasticsearch index searches can be executed with the -r argument. The following example shows how to search the exception index using the Elasticsearch query:

```
searchExceptions.sh -r "file error"
```

This searches for the exceptions that contain either word *file* or word *error* in the following fields of the Elasticsearch index:

- Name
- Folder
- Namespace
- Author
- Description
- Message
- Language
- Comment

A more relevant match (i.e. all or many search words are present) appear sooner in the returned list of results.

To see all functions supported by the script, the -h option can be used:

```
searchExceptions.sh -h
```

8.4.6 Exceptions examples

The examples show usage of the CII error handling system for Java, CPP and Python. As the majority of native language exception handling is similar across all three languages, a general example is shown only for Java:

```
public class ErrorHandlingExample {

  //initialize logger
  private static Logger logger =
                CiiLogManager.getLogger("ErrorHandlingUsageExample");

  public static void main(String[] args) {

    /* Main method is the on the highest level (first) of exception handling.
     * All unhandled exception will end the application.
     */

    //initialize example object
    ErrorHandlingExample ex = new ErrorHandlingExample();

    /* Handle the exception from the runProductionArrayElementCode() and
```

(continues on next page)

(continued from previous page)

```
     * log the current exception to log (Level.WARNING).
     */
    try {
      ex.runProductionArrayElementCode();
    } catch (CiiException e) {
      /* Exception is treated as warning. Current exception stack
       * is logged on a warning.
       * Exception is not passed to higher level, so the stack is cleared.
       */
      logger.warn(CiiLogManager.formatException(e));
    }

    /* Error will not be logged at this level.
     * Stack will also contain entries from the ex.runProductionFileOpen();
     */
    try {
      ex.runProductionFileOpen();
    } catch (CiiException e) {

      /*
       * Access individual information
       */
      System.out.println(e.getTypeName()); //errorTypeID
      System.out.println(e.getCreationDate()); //detectionTimestamp
      System.out.println(e.getStackTraceAsString());//localization information
      System.out.println(e.getMessage()); //message


      /* This will exit the application with CiiException as the exception
       * is thrown from the main method.
       */

      throw e;
    }

  }

  /**
   * Methods on ErrorHandlingExample present the 2 level of exception
   * handling.
   */
  public void runProductionArrayElementCode() {

    /*
     * Error from lower level is handled as non-critical and logged
     * as warning.
     * The super-type CiiException also catches CiiArrayOutOfRangeException
     */
    ProductionCode pcHandled = new ProductionCode();
```

(continues on next page)

(continued from previous page)

```
    try {
      pcHandled.codePartThatGetsArrayElement();
    } catch (CiiException ex) {
      logger.warn(CiiLogManager.formatException(ex));
    }

    /*
     * Error from lower level is handled but passed also to higher level.
     */
    ProductionCode pc = new ProductionCode();
    try {
      pc.codePartThatGetsArrayElement();
    } catch (CiiBadIndexException ex) {
      /*
       * Stack will contain details form CiiBadIndexException
       * Error will be propagated to upper level
       */
      throw ex;
    }
  }

  public void runProductionFileOpen() {
    /*
     * Non handled, error will be handled in the higher level.
     */
    ProductionCode pcNonHandled = new ProductionCode();
    pcNonHandled.codePartThatTriesToOpenFile();
  }

  private class ProductionCode {

    /**
     * Application level 3 error handling
     */
    public void codePartThatGetsArrayElement() {

      final ArrayList<Integer> testList =
            new ArrayList(Arrays.asList(1, 2, 3, 4, 5));
      final int arrayIndex = 9;

      /*
       * Example of array out of range error, where java native exception
       * is wrapped into
       * CiiArrayOutOfRangeException. The error is propagated to higher level.
       */

      try {
        testList.get(arrayIndex);
      } catch (IndexOutOfBoundsException ex) {
```

(continues on next page)

(continued from previous page)

```
        /*
         * Internal exception will be caught, but nothing will be passed.
         * IndexOutOfBoundsException is native java exception.
         */
      }

      try {
        testList.get(arrayIndex);
      } catch (IndexOutOfBoundsException ex) {
        /* Internal exception will be caught, and passed
         * as a CiiBadIndexException
         *
         * Error message example:  "Index 9 is out of bounds!
         * Error will be propagated to higher level.
         */
        throw new CiiBadIndexException(arrayIndex, ex);
      }
    }


    public void codePartThatTriesToOpenFile() {

      /*
       * Non existing file is logged as info.
       */
      final String path = "data.txt";
      final File myDatafile = new File(path);
      if (!myDatafile.exists()) {
        /*
         * We treat this as non - error, we only log the information
         * to logger.
         */
        logger.log(Level.INFO, "File %s could not be opened", path);
      }

      /*
       * Non existing file is threaded as error and passed to higher level.
       */
      if (!myDatafile.exists()) {
        /*
         * Pass the exception to higher level.
         * Exception will have path added to the message.
         */
        throw new CiiFileHandlingException(path);
      }
    }
  }
```

Presented example will produce the following log messages:

```
14:53:41.056 [main] INFO ErrorHandlingUsageExample - Index 9 is out of bounds!,␣
↪ErrorTypeName=elt.error.CiiBadIndexException, ErrorDateTime=1579445621042,␣
↪ErrorMessage=Index 9 is out of bounds!, ErrorTrace=elt.error.
↪CiiBadIndexException
     at elt.error.ErrorHandlingExample$ProductionCode.
↪codePartThatGetsArrayElement(ErrorHandlingExample.java:127)
     at elt.error.ErrorHandlingExample.
↪runProductionArrayElementCode(ErrorHandlingExample.java:75)
     at elt.error.ErrorHandlingExample.main(ErrorHandlingExample.java:41)
Caused by: java.lang.IndexOutOfBoundsException: Index: 9, Size: 5
     at java.util.ArrayList.rangeCheck(ArrayList.java:657)
     at java.util.ArrayList.get(ArrayList.java:433)
     at elt.error.ErrorHandlingExample$ProductionCode.
↪codePartThatGetsArrayElement(ErrorHandlingExample.java:120)
     ... 2 more
,
14:53:41.069 [main] WARN ErrorHandlingUsageExample - File data.txt could not be␣
↪opened
Exception in thread "main" elt.error.CiiFileHandlingException: Error while␣
↪handling: data.txt
     at elt.error.ErrorHandlingExample$ProductionCode.
↪codePartThatTriesToOpenFile(ErrorHandlingExample.java:154)
     at elt.error.ErrorHandlingExample.
↪runProductionFileOpen(ErrorHandlingExample.java:90)
     at elt.error.ErrorHandlingExample.main(ErrorHandlingExample.java:49)
```

Source code of this example can be found in the client-api module under elt-common/java/common/test/elt/error/ErrorHandlingExample.java

Building an Error Stack

The following examples show sample code of building the exception stack in different languages. The exception stack is built by nesting the exceptions:

Building Error Stack in Java

```
try {
     try {
               throw CiiOldbDpExistsException("testDPUri");
               } catch (CiiOldbDpExistsException ex) {
                       throw (CiiBadIndexException(10, ex));
               }
     } catch (CiiBadIndexException ex) {
               ex.getCiiExceptionStackAsString();
     }
}
```

Building Error Stack in Python

```
try:
    try:
        raise CiiInvalidTypeException("")
    except CiiException as x:
        raise CiiBadIndexException(x, 10)
except CiiException as cix:
        print(cix.get_stack_trace())
```

Building Error Stack in CPP

```
try
{
  try
  {
    throw elt::oldb::CiiOldbDpExistsException("testDPUri");
  } catch (const elt::error::CiiOldbException& e) {
    // build exception stack of nested exceptions
    // CiiOldbDpExistsException is nested in CiiBadIndexException
    CII_THROW_WITH_NESTED(elt::error::CiiBadIndexException, e, 10);
  }
} catch (const elt::error::CiiException& e) {
  // flatten all nested exceptions and dump to string
  std::string nestedDetails = e.dumpWithNested();
}
```

Throwing a CPP Exception with Extra Information

CPP API provides additional macros which add line number and code filename to the error messages. Section A.2 in Appendix A contains the details.

The following example shows throwing a CPP exception with details (line number, function, file name, etc.). Standard CPP exception what() method (inherited from std::exception) shall be used to obtain the exception message. CPP API also provides the extra Dump() method (see details in section A.1).

```
try
{
  CII_THROW(elt::error::CiiFileHandlingException, "myfileName");
}
catch(const elt::error::CiiException& e)
{
  std::cout << e.What(); //error message
  std::cout << e.GetDetails(); //error details
  std::cout << e.Dump(); //error dump
}
```

Exception Serialization from CPP to Java

The following code shows an example of transferring the exception from CPP to Java. All the exceptions are serialized using the MAL CiiSerializableException.

```
//CPP method with CiiException on the sender side
std::string methodWithException(const std::string &par1) override
{
  throw elt::error::CiiInvalidDataTypeException(par1);
  return par1;
}

//Java - Receiving exception on the reciever side
try {
  final String unexpectedReply = serialzation.methodWithException("Test");
} catch (CiiSerializableException e) {
  logger.info(String.format("Type: %s", e.getCiiType()));
  logger.info(String.format("Host name: %s", e.getCiiHostName()));
  logger.info(String.format("File name: %s", e.getCiiFileName()));
  logger.info(String.format("Class name: %s", e.getCiiClassName()));
  logger.info(String.format("Method name: %s", e.getCiiMethodName()));
  logger.info(String.format("Message: %s", e.getCiiMessage()));
  logger.info(String.format("Creation Date: %d", e.getCiiCreationDate()));
  logger.info(String.format("Stack trace: %s", E.getCiiStackTrace()));
  logger.info(String.format("Exception stack: %s", e.getCiiExceptionStack()));
  logger.info(String.format("Details: %s", e.getCiiDetails()));
}
```

8.4.7 Using the Condition API

The following example shows the use of the condition API. Since the use is similar for Java, CPP and Python, only a Java example is presented.

```
public static void main(String[] args) {
  // Enable condition API for the process.
  CiiConditions.setEnabled(true);

  ErrorConditionsExample ex = new ErrorConditionsExample();

  try{
     ex.setRotation(250);
  } catch (CiiException x){
     System.out.println(x.getMessage());
  }
  ex.getListValue(10);
}

// get value from array of float
```

(continues on next page)

(continued from previous page)

```
public float getListValue(int index){
  // check is element is in range.
   CiiConditions.checkElementIndexRange(index, list.length,
                new CiiBadIndexException(index));
   return list[index];
}

public void setRotation(int degrees){
  //pre condition check -- check if rotation in smaller than 360 deg.
  CiiConditions.checkArgument(degrees <= 360, new CiiOverRotatedException());
  axis.setRotation(degrees)
  axis.rotate()
  final int readRotation = 200;
  // post condition check -- check if axis is properly rotated.
  CiiConditions.checkArgument(readRotation == degrees,
      new CiiNotProperlyRotatedException());
}
```

8.5 Using the QT Error Dialog Widget

To use the error dialog widget, CiiExceptionDialog.hpp must be included into the source code. The dialog is located under the elt::error::gui namespace. The CiiExceptionDialog.hpp derives from the QDialog. The method exec() is therefore directly inherited from this QDialog class.

Exception dialog constructor takes a CiiException as an input parameter. All exception details are automatically parsed into the dialog, based on the CiiException type.

To display the widget, exec() method must be called. The following code shows the typical usage:

```
#include <CiiExceptionDialog.hpp>

try {
  throw elt::error::CiiInvalidTypeException();
} catch (const elt::error::CiiException& e) {
  gui::CiiExceptionDialog w(e);
  w.exec();

}
```

Figure 5-1 shows an example of the Error widget dialog screen output. The widget displays all the important error information: Exception class (error type), exception message, time of the occurrence, thread ID, process ID, host, line number and stack trace.

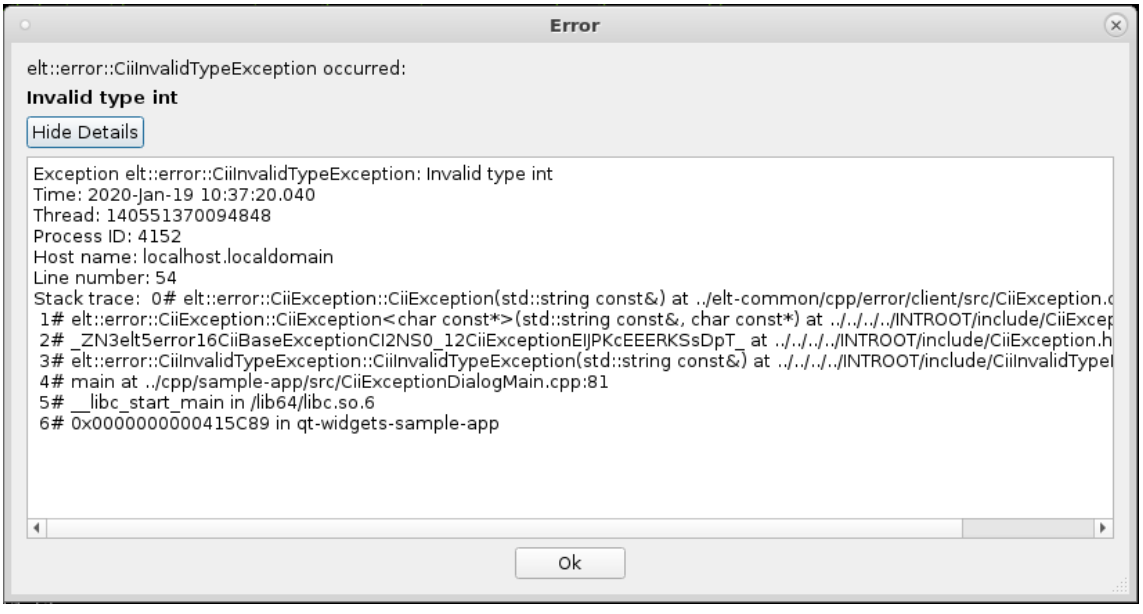


Figure 5-1 Error widget dialog screen output

8.6 Indexing service

Error indexer (2.2.2) is a tool that collects the data of exceptions and stores the exceptions data to Elasticsearch index. Using Elasticsearch for searching the exceptions provides structured and powerful search. Data provided by indexing service is used by the search script (4.5), when using the ElasticSearch option. As ElasticSearch returns data in JSON format, the data returned by the queries can be used for fast integration with other services.

8.6.1 Configuring indexing service

Error indexer uses configuration localDB for setting up the needed configuration. The configuration instance file must be created and deployed for error indexer to work properly. Use the following template to create the config file:

```
config:
  instance:
    __comment__: Configuration for error-indexer
    data:
      "@type": CiiErrorIndexerConfigClass
      fields:
        elasticSearchServiceAddress:
           "@type": MdString
           metadataInstance: ConfStringStd
           value: "localhost"
        elasticSearchIndexName:
           "@type": MdString
           metadataInstance: ConfStringStd
           value: "errors"
        centralErrorCodesRepositoryHost:
           "@type": MdString
           metadataInstance: ConfStringStd
           value: "https://svnhq8.hq.eso.org"
        centralErrorCodesRepositoryPaths:
           "@type": MdStringArray
           metadataInstance: ConfStringArrayStd
           value:
             - "p8/trunk/CONTROL-SYSTEMS/CentralControlSystem/CII/CODE/srv-error-
↪indexer/sampleErrorCodesRepo/"
        lastProcessedRevision:
           "@type": MdInt32
           metadataInstance: ConfInt32Std
           value: 0
        centralErrorCodesPassword:
           "@type": MdString
           metadataInstance: ConfStringStd
           value: "myPass"
        centralErrorCodesUserName:
           "@type": MdString
```

(continues on next page)

(continued from previous page)

```
            metadataInstance: ConfStringStd
            value: "myUsername"
```

After the file is created (we assume it is saved under ErrorIndexerConfig.cdi.yaml), it has to be deployed to the localDB. If localDB is not initialized, it should be initialized first.

```
config-tool init #LocalDb initialization.
config-tool deploy -i localDbConfiguration/ErrorIndexerConfig.cdi.yaml -a cii.
↪config://services/CiiErrorIndexer
```

Following properties are set:

- elasticSearchServiceAddress: Address of the elastic search server.
- elasticSearchIndexName: Index name, which will hold the indexed data.
- centralErrorCodesRepositoryHost: SVN hostname of central repository.
- centralErrorCodesRepositoryPaths: Array of paths in the repository. Indexer will look for exception code under these subfolders.
- lastProcessedRevision: Internally, last processed revision is stored. The setting should initially be set to 0.
- centralErrorCodesPassword: password credential for SVN.
- centralErrorCodesUserName: user name credential for SVN.

8.6.2 Running the indexing service

To manually run the indexer, execute the following command:

```
error-indexer
```

It is proposed to run the indexer every 30 minutes. A Linux time-based utility *cron* shall be used for this. To define the schedule to use the *cron*, put the following line into the */etc/crontab* (change the {INTROOT} to proper location of the CII root installation folder – varriable $INTOOT cannot be used here):

```
30 * * * * {INTROOT}/bin/error-indexer
```

Another option (not documented in this manual) is to hook running of indexer on every SVN update.

8.6.3 Accessing the indexed data

Elasticsearch provides more powerful search capabilities of the exception definitions and more structured output searched data. As the data is returned in JSON format, it is more suitable for integration with other services, for example logging. Elasticsearch uses REST API to access the data. Any REST client can be used to access the errors data. For the following examples we will use *curl*.

Displaying all the errors in the index:

```
curl -XPOST 'http://ciielastichost:9200/errors/_search?pretty'
```

Elasticsearch provides Query DSL language [4], with powerful search capabilities. The following fields can be used, when searching the error index:

- folder,
- author,
- namespace,
- description,
- language,
- message,
- version,
- revision,
- timestamp.

Elasticsearch supports different types of searches. When using the *wildcard* type of search, only lower case is supported (ElasticSearch standard analyzer is used [5]). Data in index can be in camel case, but the search will only work if the lower-case words are used in the search query. The search will not distinguish between lower and upper case.

Getting the error data based on the exception name:

```
curl -XPOST 'http://ciielastichost:9200/errors/_search?pretty' -d '{"query":{
↪"match":  {"name": " CiiFileHandlingException"}}}' -H "Content-Type:␣
↪application/json"
```

Wildcard search based on exception name:

```
curl -XPOST 'http://ciielastichost:9200/errors/_search?pretty' -d '{"query":{
↪"wildcard":  {"name": "ciifile*"}}}' -H "Content-Type: application/json"
```

Searching the errors with the desired description:

```
curl -XPOST 'http://ciielastichost:9200/errors/_search?pretty' -d '{"query":{
↪"match":  {"description": "This is the description"}}}' -H "Content-Type:␣
↪application/json"
```

Searching the errors with the desired description and name:

```
curl -XPOST 'http://ciielastichost:9200/errors/_search?pretty' -d '{"query": {
↪"bool": {"should": [{ "match": { "description": "This is the description"}}, {
↪"match": { "message": "Error while"}}]}}}' -H "Content-Type: application/json"
```

Wildcard search with the desired description and name:

```
curl -XPOST 'http://ciielastichost:9200/errors/_search?pretty' -d '{"query": {
↪"bool": {"should": [{ "wildcard": { "description": "* is the description"}}, {
↪"wildcard": { "message": "*while"}}]}}}' -H "Content-Type: application/json"
```

Search with the desired language and name:

```
curl -XPOST 'http://ciielastichost:9200/errors/_search?pretty' -d '{"query": {
↪"bool": {"must": [{ "match": { "language": "JAVA"}}, { "match": { "name": "␣
↪CiiFileHandlingException"}}]}}}' -H "Content-Type: application/json"
```

Return the last record of wildcard search sorted by timestamp:

```
curl -XPOST 'http://ciielastichost:9200/errors/_search?pretty' -d '{"query":{
↪"wildcard":  {"name": "ciifile*"}},"sort": [ { "timestamp": { "order": "desc" }
↪ } ], "size": 1}' -H "Content-Type: application/json"
```

8.7 API

8.7.1 CII Exception methods

CII-specific fields and methods are added to CiiException to provide the required functionalities:

stack trace: (execution stack trace - back trace) is provided by language-specific trace mechanisms in Java and Python. CPP uses the *boost libbacktrace* module. The getStackTraceAsString method provides a stack trace in a string format.

creation date: creation date/time is implicitly set by CiiException. The date/time is stored in milliseconds since Epoch. The getCreationDate method will return the number.

type name: type name is implicitly set by the exception definition. The getCiiType method will provide the type of the exception class.

The CPP CiiException class provides additional functionalities:

The following fields are provided as member fields with getter and setter methods:

threadId

lineNumber

functionName

filename

processId

hostname

details: additional exception trace details can be added to CPP exceptions. This attribute should be explicitly set by the developer. The getDetails and setDetails methods should be used to retrieve and store the information.

The two additional methods are added:

dump: method that returns string formatted information about all exception details and values.

dumpWithNested: method that returns string formatted information about all nested exceptions details and values.

8.7.2 CPP macros

CPP API provides macros which add details (file, function and line number) to the error messages. If these macros are not used, the exceptions will not contain the mentioned fields. The following macros are provided:

CII_THROW(exceptionType_t, . . .)
Throws the desired exception with additional information. First parameter is the exception type. All other parameters are variadic, and depend on the number of parameters the exception class defines.

CII_THROW_WITH_NESTED (exceptionType_t, nested_exception, . . .)
Throws the desired exception with additional information and nested exception. The first parameter is the exception type and the second parameter is the exception to be nested. All other parameters are variadic, and depend on the number of parameters the exception class defines.

8.7.3 Conditions API

The conditions API provides static methods for verifying arguments, null values, states and valid index positions in lists and arrays. It provides the following methods:

setEnabled(bool enabled) getEnabled()
Methods for enabling and disabling the Conditions API. If Conditions API is disabled, checks will not be executed. The getEnabled method will return the current state of the enabled flag.

setFailFastMode(bool enabled) getFailFastMode()
Methods for enabling and disabling the fail-fast mode. If fail-fast is enabled, failing the condition will cause the application to exit with an error. The getFailFastMode method will return the current state of the enabled flag.

checkArgument(bool expr, Exception ex)
Checks that the parameter expr is true. It can be used for validating arguments in methods. If the expression evaluates to false, the given exception is thrown.

checkNotNull(T value, Exception ex)
Checks that the value is not null. It returns the value directly, so you can use checkNotNull(value, ex) inline. In case of null, the given type exception is thrown.

checkState(boolean value, Exception ex)
Checks some state of the object, not dependent on the method arguments. For example, an iterator might use this to check that method next() has been called before any call to remove() method. If the expression evaluates to false, the given exception is thrown.

checkElementIndexRange (int index, int size, Exception ex)
Checks that index is a valid element index in a list, string, or array with the specified size. An element index may range from 0 inclusive to size exclusive. Instead of passing the whole list, string, or array, only it's size is passed as parameter. If the index exceeds the size, the given exception is thrown.

The condition API for specific languages (Java, CPP and Python) is shown in the following sections.

Java

```
public class CiiConditions {
    public static boolean getEnabled() {...}

    public static void setEnabled(boolean enabled) {...}

    public static boolean getFailFastMode(){...}

    public static void setFailFastMode(boolean failFastMode) {...}

    public static <E extends Exception> void checkArgument(boolean expr, E ex)␣
↪throws E {...}

    public static <T, E extends Exception> void checkNotNull(T value, E ex)␣
↪throws E {...}

    public static <E extends Exception> void checkState(boolean value, E ex)␣
↪throws E {...}

    public static <E extends Exception> void checkElementIndexRange(int index,␣
↪int size, E ex) throws E {...}


}
```

CPP

```cpp
class CiiConditions {
public:

    static bool GetEnabled();

    static void SetEnabled(bool enabled);

    static bool GetFailFastMode();

    static void SetFailFastMode(bool failFast);

    template<typename E, class T, typename ... Args>
    static void CheckArgument(bool expr, Args ... args);

    template<typename E, class T,  typename ... Args>
    static void CheckNotNull(T value, Args ... args)

    template<typename E, class T, typename ... Args>
    static void CheckState(T value, Args ... args)

    template<typename E, class T, typename ... Args>
    static void CheckElementIndexRange(size_t index,
                 size_t size, Args ... args);

};
```

Python

```python
class CiiConditions:

   @staticmethod
  def get_enabled():
      pass

  @staticmethod
  def set_enabled(enabled):
      pass

  @staticmethod
  def get_fail_fast_mode():
      pass

  @staticmethod
  def set_fail_fast_mode(failFastMode):
      pass
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def check_argument(expr, ciiException):
        pass

    @staticmethod
    def check_not_null(value, ciiException):
        pass

    @staticmethod
    def check_state(expr, ciiException):
        pass

    @staticmethod
    def check_element_index_range(index, size, ciiException):
        pass
```

8.8 Serialization code example

The following code presents example of serialization of exceptions over MAL. The example consists of MAL CPP server and Java client. Examples for other languages can be found in the client-api module source code:

- Java: client-api/elt-common/java/errorNonUnitTests
- CPP: client-api/elt-common/cpp/error/errorSerialClnt, errorSerialSrv
- Python: client-api/elt-common/runScripts/testZpbClientPythonException, testZpbServer-PythonException

8.8.1 CPP serialization server code

```
#include <CiiSerializableException.hpp>
#include <CiiSerializationTest.hpp>
#include <CiiInvalidDataTypeException.hpp>

#include <mal/rr/qos/QoS.hpp>
#include <mal/Cii.hpp>
#include <mal/Mal.hpp>
#include <mal/utility/LoadMal.hpp>
#include <mal/rr/ServerContextProvider.hpp>
#include <mal/rr/ServerContext.hpp>
#include <mal/rr/ServerAmi.hpp>
#include <mal/rr/RrEntity.hpp>

#include <stdexcept>
#include <memory>
#include <iostream>
#include <boost/thread/future.hpp>

namespace {

const std::string EXCEPTION_CALL = "error";

/**
 * Simulate some work
 */
static void simulateWork() {
  std::this_thread::sleep_for(std::chrono::seconds(2));
}

class CiiSerializationTestImpl : public virtual␣
↪elt::error::icdTest::CiiSerializationTest
{
  std::string methodWithException(const std::string &par1) override
  {
```

(continues on next page)

(continued from previous page)

```
    std::cout << "Got exception call with parameter: " << par1 << std::endl;
    if (par1 == EXCEPTION_CALL)
    {
      elt::mal::throw_exception(elt::error::CiiInvalidDataTypeException(par1));
    }
    return par1;
  }

  std::shared_ptr<::elt::mal::rr::Ami<std::string>>
  amiMethodWithException(const std::string& par1)
  {
    // Obtain ptr to ServerAmi
    std::shared_ptr<::elt::mal::rr::Ami<std::string>> ami =␣
↪elt::mal::rr::ServerContextProvider::
        getInstance<elt::mal::rr::ServerContext<std::string>>().createAmi();

    std::cout << "Ami call with param: " << par1 << std::endl;

    if (par1 != EXCEPTION_CALL)
    {
      auto future = boost::async(boost::launch::async, [=]() {
        simulateWork();
        // first response
        ami->complete("ONE");

        simulateWork();
        // second response
        ami->complete("TWO");

        simulateWork();
        // third response
        ami->completed("THREE");
      });
    }
    else
    {
      auto future = boost::async(boost::launch::async, [=]() {
        simulateWork();
        // first response
        ami->complete("ONE");

        simulateWork();
        // second response
        ::elt::error::CiiInvalidDataTypeException e =␣
↪elt::error::CiiInvalidDataTypeException(par1);
        e.setFileName(boost::filesystem::path(__FILE__).filename().string());
        e.setFunctionName(__FUNCTION__);
        e.setLineNumber(__LINE__);
        e.setClassName(boost::core::demangle(typeid(*this).name()));
```

(continues on next page)

(continued from previous page)

```
        ami->completeExceptionally(e);

        simulateWork();
        // third response
        ami->completed("THREE");
      });
    }
    return ami;
  }
};

} // namespace

int main(int ac, char* av[]) {
  std::cout << "CPP ZPB server test start." << std::endl;

  // Load DDS MAL mapping
  std::shared_ptr<::elt::mal::Mal> ddsMal = elt::mal::loadMal("zpb", {{"dds.
↪domain", "100"}});
  ::elt::mal::CiiFactory &factory = elt::mal::CiiFactory::getInstance();
  factory.registerMal("zpb", ddsMal);

  elt::mal::Uri uri("zpb.rr://0.0.0.0:12081/CiiSerializationTest");

  try {
    // Create server, default QoS.
    std::unique_ptr<elt::mal::rr::Server> server = factory.createServer(uri,␣
↪elt::mal::rr::qos::QoS::DEFAULT, {});

    // Register the service.
    std::shared_ptr<elt::mal::rr::RrEntity> exSer(new␣
↪CiiSerializationTestImpl());
    server->registerService<elt::error::icdTest::CiiSerializationTest, true>(
↪"inst_1", exSer);

    // Process commands until server->shutdown() is called.
    server->run();
  } catch (std::exception& exc) {
    std::cerr << "EXCEPTION: " << exc.what() << std::endl;
  }

  std::cout << "Server stopped." << std::endl;
  return 0;
}
```

8.8.2 Java serialization client code

```
import java.net.URI;
import java.util.Properties;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import org.apache.logging.log4j.Logger;
import elt.error.icd.CiiSerializableException;
import elt.error.icdTest.CiiSerializationTestSync;
import elt.log.CiiLogManager;
import elt.mal.CiiFactory;
import elt.mal.Mal;
import elt.mal.rr.Ami;
import elt.mal.rr.qos.QoS;
import elt.mal.rr.qos.ReplyTime;
import elt.mal.zpb.ZpbMal;

public class ExceptionSerializationClientTest {

  private static final Logger logger = CiiLogManager.
↪getLogger(ExceptionSerializationClientTest.class.getName());
  private static final String NO_EX_CALL = "first";
  private static StringBuilder outData = new StringBuilder();

  public static void main(String[] args) {
    logger.debug("Java test zpb client started.");

    final CiiFactory factory = CiiFactory.createInstance();

    final URI uri = URI.create("zpb.rr://127.0.0.1:12081/CiiSerializationTest/
↪inst_1");
    logger.debug("URI: " + uri.toString());

    Mal mal = new ZpbMal();
    factory.registerMal("zpb", mal);

    try (CiiSerializationTestSync serialzation =
        factory.getClient(uri, new QoS[] {new ReplyTime(3, TimeUnit.SECONDS)},␣
↪new Properties(),
            CiiSerializationTestSync.class)) {

      logger.info("Perfomrming simple call test.");
      try {
        final String reply = serialzation.methodWithException(NO_EX_CALL);
        if (!reply.equals(NO_EX_CALL)) {
          logger.error("Error!!!! Did not get expected string.");
          System.exit(1);
        }
        final String unexpectedReply =
```

(continues on next page)

(continued from previous page)

```
            serialzation.methodWithException(ExceptionSerializationServerTest.
↪EXCEPTION_CALL);
        logger.error("Error!!!! Should not be here.");
      } catch (CiiSerializableException e) {
        final CiiException ex = CiiException.fromSerializable(e);
        logger.debug("Got CiiSerializableException.");
        logger.info(String.format("Type: %s", ex.getCiiType()));
        logger.info(String.format("Host name: %s", ex.getCiiHostName()));
        logger.info(String.format("File name: %s", ex.getCiiFileName()));
        logger.info(String.format("Class name: %s", ex.getCiiClassName()));
        logger.info(String.format("Method name: %s", ex.getCiiMethodName()));
        logger.info(String.format("Message: %s", ex.getCiiMessage()));
        logger.info(String.format("Creation Date: %d", ex.getCiiCreationDate()));
        logger.info(String.format("Stack trace: %s", ex.getCiiStackTrace()));
        logger.info(String.format("Exception stack: %s", ex.
↪getCiiExceptionStackAsString()));
        logger.info(String.format("Details: %s", ex.getCiiDetails()));

        //the data to be checked by the integration test
        outData.append("Type -- ").append(e.getCiiType()).append(";");
        outData.append("Message -- ").append(e.getCiiMessage()).append(";");
        outData.append("Exception stack -- ").append(ex.
↪getCiiExceptionStackAsString());

      }

      logger.info("Perfomrming Ami call test.");
      try {
        logger.info("OK call.");
        final Ami<String> okTasks = serialzation.amiMethodWithException(NO_EX_
↪CALL);
        for (CompletableFuture<String> reply : okTasks) {
          final String result = reply.get();
          logger.info("Reply: " + result);
          logger.info("Task complete: " + okTasks.isDone());
        }

        logger.info("Error call.");
        final Ami<String> errorTasks = serialzation.
↪amiMethodWithException(ExceptionSerializationServerTest.EXCEPTION_CALL);
        for (CompletableFuture<String> reply : errorTasks) {
          final String result = reply.get();
          logger.info("Reply: " + result);
          logger.info("Task complete: " + errorTasks.isDone());
        }
        logger.error("(Ami) Error!!!! Should not be here.");
      } catch (InterruptedException e) {
        logger.error("Unexpected error: ", e);
      } catch (ExecutionException e) {
```

(continues on next page)

(continued from previous page)

```
        final Throwable th = e.getCause();
        if (th instanceof CiiSerializableException) {
          final CiiSerializableException se = (CiiSerializableException) th;
          final CiiException ex = CiiException.fromSerializable(se);
          logger.debug("Got CiiSerializableException.");
          logger.info(String.format("Type: %s", ex.getCiiType()));
          logger.info(String.format("Host name: %s", ex.getCiiHostName()));
          logger.info(String.format("File name: %s", ex.getCiiFileName()));
          logger.info(String.format("Class name: %s", ex.getCiiClassName()));
          logger.info(String.format("Method name: %s", ex.getCiiMethodName()));
          logger.info(String.format("Message: %s", ex.getCiiMessage()));
          logger.info(String.format("Creation Date: %d", ex.
↪getCiiCreationDate()));
          logger.info(String.format("Stack trace: %s", ex.getCiiStackTrace()));
          logger.info(String.format("Exception stack: %s", ex.
↪getCiiExceptionStackAsString()));
          logger.info(String.format("Details: %s", ex.getCiiDetails()));
        } else {
          logger.error("Unexpected error: ", e);
        }
      } catch (Throwable e) {
        logger.error("Unexpected throwable: ", e);
      }

    } finally {
      mal.close();
      factory.close();
    }
    logger.debug("Client done.");
    System.out.print(outData.toString());
    System.out.print("PASS");
  }

}
```

CHAPTER

NINE

LOGGING

Document ID:	
Revision:	2.2
Last modification:	March 18, 2024
Status:	Released
Repository:	https://gitlab.eso.org/cii/info/cii-docs
File:	log.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
0.1	25.02.2020	mfernandez	All	Document creation.
1.0	06.05.2020	mfernandez/ bterpinc	All	Updated all sections and added tracing. Final review.
1.1	11.09.2020	dkumar	9.2.3 9.2.2.2 9.2.3	Tracing changes. New heading 9.2.3.
1.2	28.09.2020	mfernandez	All	Updates after LAT topical review
1.3	28.09.2020	dkumar	9.2.4 9.2.3	OLDB Context propagation . Mal pub/sub examples.
1.4	29.09.2020	mfernandez	4.8 5	Rearranged TOC structure. Moved String formatting and CLI tool sections
1.5	15.10.2020	mfernandez	2, 8.1 6.1.1 8.2.1.1, 8.2.1.2	Added “separation ” between logging and tracing. Added description of Filebeat multiline options. New chapters added.
1.6	01.12.2020	mfernandez	4	Fixed inconsisten cies in C++ example executables names
1.7	27.01.2021	mfernandez	2, 4, 6 , 7.1	Replaced usages of INTROOT with CIISRV_ROOT
1.8	14.03.2022	mschilli	5, 6, 7	logSend(5) now part of Tools(now 6) Added ciiLogViewer
1.9	02.02.2023	mschilli	all	corrected logsink: /var/log/elt and $CII_LOGS
1.10	27.04.2023	mschilli	2	new section “Includes and Imports”
2.0	20.06.2023	mschilli	2,3,4,6	ELK-stack replaced by rsyslog
2.1	28.02.2024	mschilli	4, 6	Configure C++ logging from Python Regex-based view filter
2.2	18.03.2024	mschilli	0	Public doc

Confidentiality

This document is classified as Public.

Scope

This document is a manual for the Logging system of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

Glossary of Terms

API	Application Programming Interface
CII	Core Integration Infrastructure
MAL	Middleware Abstraction Layer
OLDB	Online Database
EA	Engineering Archive
CLI	Command Line Interface
GUI	Graphical User Interface
JSON	JavaScript object notation
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language
TTL	Time-To-Live

References

1. ESO, Core Integration Infrastructure Requirements Specification, ESO-192922 Version 7
2. Cosylab, ELT CII Log Transfer Document, CSL-DOC-20-181435, Version X.X
3. Cosylab, ELT CII Error handling Manual Document, CSL-DOC-20-181435, Version X.X
4. Log4j2, https://logging.apache.org/log4j/2.x/
5. Log4cplus, https://sourceforge.net/p/log4cplus/wiki/Home/
6. Python logging, https://docs.python.org/3/library/logging.html
11. Log4j2 Async Logging, https://logging.apache.org/log4j/2.x/manual/async.html
17. Checker Framework, https://checkerframework.org/
20. Log4cplus logging macros, https://log4cplus.sourceforge.io/docs/html/loggingmacros_8h_source.html
22. OpenTracing API docs, https://opentelemetry.io/docs/
23. Jaeger Tracing docs, https://www.jaegertracing.io/docs/1.17/
24. Jaeger Client Libraries, https://www.jaegertracing.io/docs/1.17/client-libraries/
25. Jaeger Trace Adjusters, https://pkg.go.dev/github.com/jaegertracing/jaeger/model/adjuster

9.1 Overview

This document is a user manual for usage of CII Logging and tracing systems. It explains how to use the Logging Client API through ELT CII common library and GUI Applications to interact with it. It also explains how to use the different services that comprise the distributed logging CII system.

All examples in this manual are also presented in git repository https://gitlab.eso.org/cii/info/cii-demo under logging-examples.

9.2 Introduction

CII Log provides the logging and tracing services of the Control System. Logging pertains to all aspects of CII services. Its purpose is to provide basic and detailed information of the different services and systems in the CII in order to have a better understanding of the behaviour of the distributed system, and in particular to troubleshoot the system in case of problems. Figure 2-1 shows how the CII Log Service interacts with other elements of the ELT Core Integration Infrastructure. Logging is considered a basic service and doesn't use any other CII service.

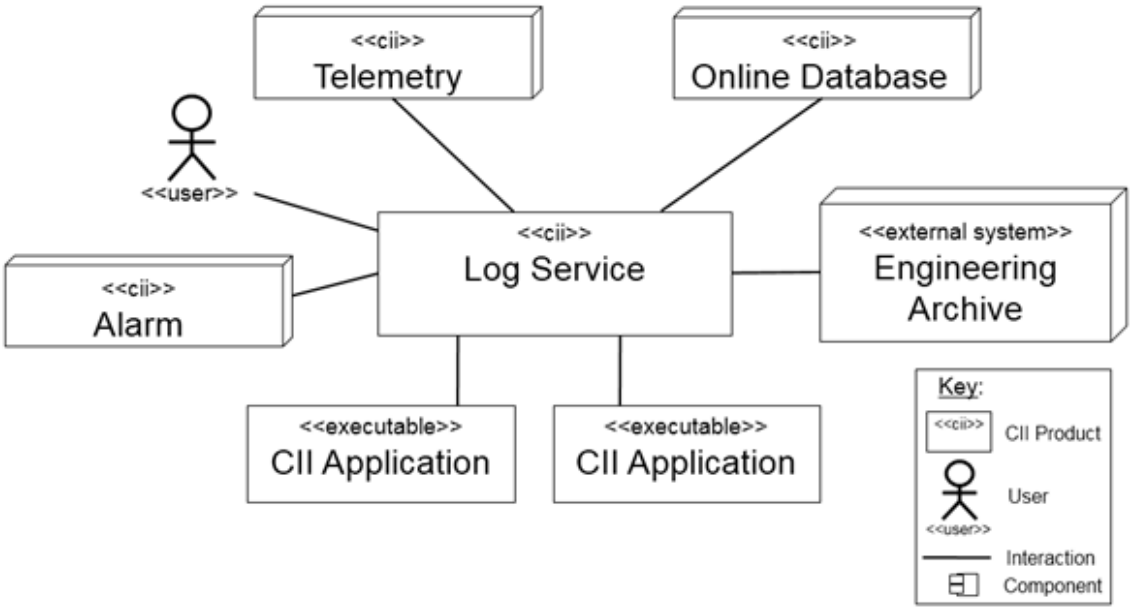


Figure 2-1 Log interaction

CII Log system components can be divided into four subcomponents according to their responsibilities:

- Client API (Log API)
- Transmission
- Persistence
- Visualization

Figure 2-2 Data flow between log producers (Host n) and central log sink (Collector).

```
Host 1,2,...
============
                [ Log Config:     ]
                [  handler=syslog ]
                [  formatter=cii  ]
                /
               /
```

(continues on next page)

(continued from previous page)

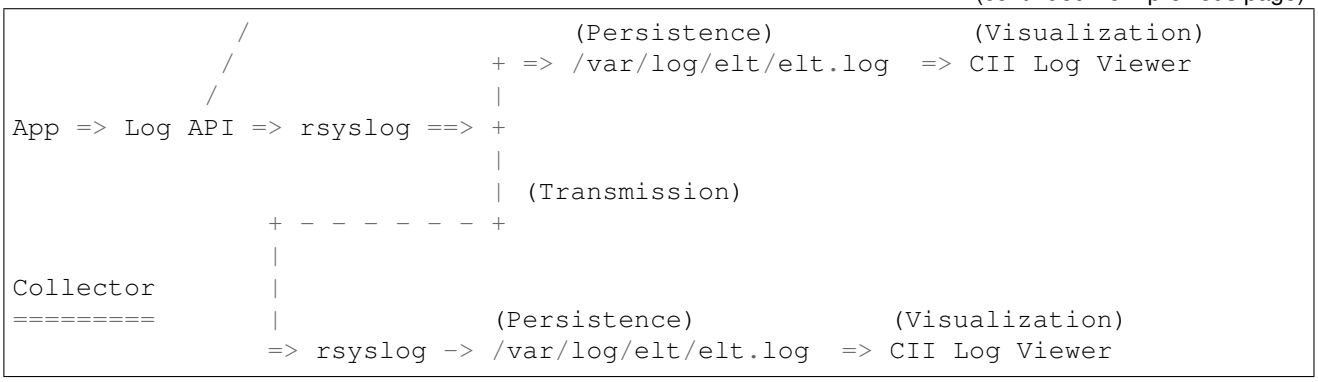


Figure 2-2 Standard workflow of CII Log system

The CII Log Client API enables users to produce log messages from their applications using Logger objects (see 2.2). These messages are produced as strings and follow a predefined format defined in the CII Logging library (see 2.3). The API leverages on existing logging frameworks and extends them in order to provide all CII functionalities. The logging frameworks it leverages on are:

- Log4j (Java) [4]
- Log4cplus (C++) [5]
- Python logging (Python) [6]

As of CII v4, rsyslog is used as transmission and management service. A client-side rsyslog daemon is responsible for pushing CII logs from a single host (see 2.1) to a remote rsyslog daemon on another host (so-called log collector). This log message is transported using rsyslog's own format, wrapping the CII-specific fields that carry way more information than a rsyslog log would have by itself.

The serialized message is not parsed by the transporting rsyslog, instead it is written verbatim to file, both on the sending host and on the collecting host. On each host, a single file is used to aggreate all log messages: `/var/log/elt/elt.log`. The file is automatically rolled over based on time. See the administration section for details.

In CII v4, there is no log harvester that would read the file and inject its content into a database. This functionality will be re-established once an updated version of the Engineering Archive is available.

A Qt GUI and a Qt widget are available as visualization tools for querying and browsing CII logs.

Tracing is separated from logging, as it serves a different purpose. Tracing is explained in detail in section 8.

9.2.1 Logsink directory

Applications have the freedom to write logs to any place they configure in their log configurations (for examples see below: “Basic usage example with custom configuration”).

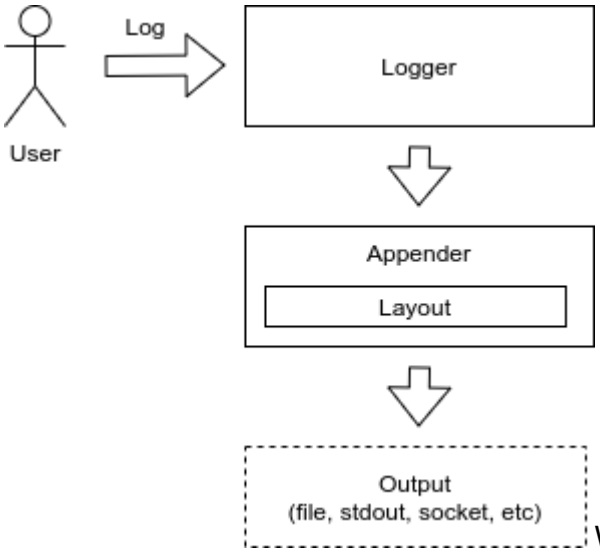
To participate in the central log transport, applications need to use rsyslog as a handler/appender in combination with the standard CII log layout format. Note that the CII brief or simple format will not work.

Logs emitted this way will be written to the log sink (`/var/log/elt`).

The Log Viewer (see below) by defaults operates on /var/log/elt, but can also be instructed to find logs in a different location.

9.2.2 Loggers and Appenders

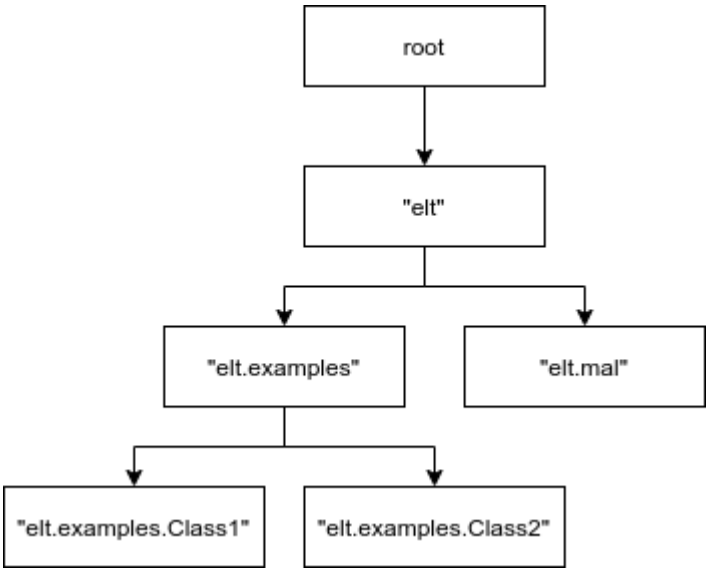
The architecture of the three used logging frameworks is relatively similar. The main elements of this architecture are the loggers, the appenders, and the layouts.



While the loggers are the elements responsible for exposing the API to create log events, appenders are responsible for writing those events to the appropriate outputs, whether it is stdout, a file, a database, a socket, or any other.

Loggers are organized in a named hierarchical structure in a parent-child relationship fashion. Each logger has a name with dot-separated components, which indicates the position of the logger in the hierarchy. An example hierarchy is shown in Figure 2-4.

There must always be a root logger with an empty name.



Appenders are attached to loggers. Each logger can have one or more appenders attached, and the same appender can be attached to multiple loggers.

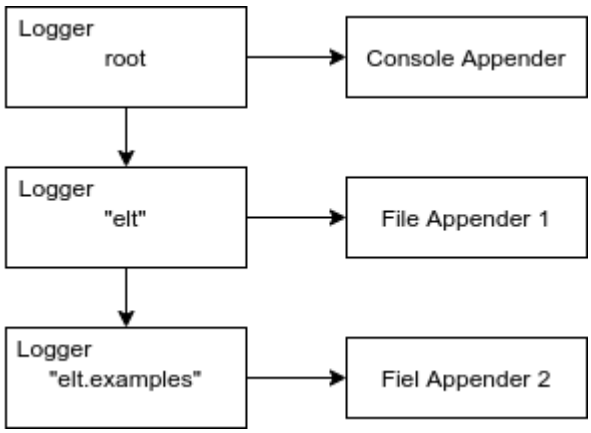


Figure 2-5 Loggers and Appenders example

Another possible scenario is having multiple child loggers with no appenders attached, having the appenders attached only to the root/parent logger. This can be useful to handle logs from different classes or packages through the child loggers but leveraging the output of those log messages to the parent/root logger and its appenders.

9.2.3 Log Layouts

Log layouts are components of the logging libraries responsible for serializing a log event into the format that will be output by the appenders, usually a string.

CII Logging library provides two different layouts that produce plain strings according to the CII Log format:

- CiiLayout
- CiiSimpleLayout

The CiiLayout is intended both for human and machine consumption, and **it must be used for logs written to the central log transport** (see 2.1). It contains the fields described in Appendix B, in the same order as shown there.

The CiiSimpleLayout is a simplified version and is intended for human consumption. It is composed of the date and time of the log event, the log type, a log ID, the message and all the optional fields described in Appendix B.

Additional details and usage instructions can be found in section 4.4.

Example log messages using both the CiiLayout and the CiiSimpleLayout are included in section 4.5 as well as examples how to use it in configurations are included in section 4.3.

Appendix B describes the fields of which a CII log is composed. All optional fields can be added to the log message using the CII Log Client API. See section 4.7 for further details.

TraceID and SpanID are used to log information about the current tracing context (see section 8).

ErrorTypeName, ErrorDateTime, ErrorMessage, and ErrorTrace are used to log a formatted CiiException.

9.2.4 Postmortem buffer

CII Log Client API offers a logs postmortem buffer facility. It enables users to obtain on demand logging information that otherwise would have been lost due to filtering by the active logging configuration.

The postmortem buffers stores produced log events for all logging levels, regardless of the filtering configurations of the loggers and appenders.

This buffer is time-bound. Events are evicted after a defined time is elapsed. The time-to-live (TTL) for the buffered log events can be configured by the user.

The flushing of this buffer is triggered on demand through the CII Log Client API. When this happens, the buffered events are flushed to the appropriate destinations.

By default, this buffer is disabled. Configuration and usage instructions can be found in section 4.8.

9.3 Prerequisites

This section describes the prerequisites for using the CII Logging libraries and applications. Note that the preparation of the environment (i.e. installation of required modules, configuration, running of services) is not in the scope of this document. Consult your administrator and the Log TRD document for information on how to establish the environment for usage of CII Logging libraries and applications.

9.3.1 Includes and imports

For basic usage of the Logging library, the user needs the CiiLogManager class. This class contains the API for interacting with the underlying Logging framework using the CII Log extensions, whether it's Log4j (Java), Log4cplus (C++), or Python logging.

It provides methods for configuring the logging system, getting a Logger object, or flushing the post-mortem buffer. Detailed API per language is provided in Appendix A. Additional details can be found in section 4.1.

In addition, the language-specific Logger and Level classes shall be imported. Level class is not necessary if convenience log methods for the different levels are used, such as debug("message") instead of log(Level.DEBUG, "message").

C++ also needs to include ciiException class, since the ciiLogManager configuration methods can throw an exception in case of an error during configuration initialization.

For more advanced use, it is necessary include the CiiLogMessageBuilder class. The CiiLogAudience enum must be imported as well if Audience field value is provided. Also, the appropriate CiiException class must be imported.

These includes, in addition to linked libraries, are shown in the example code in logging-examples module.

9.3.2 C++

To use the logging API from C++, the following directives are needed:

wscript (project)

```
cnf.check_wdep(wdep_name='client-api.elt-common.cpp.log', uselib_store='cii-
↪logging')
# Transitive Dependencies:
cnf.check_cfg(package='log4cplus', uselib_store='log4cplus', args='--cflags --
↪libs')
```

wscript (module)

```
use=[...,
 'cii-logging'
```

(continues on next page)

(continued from previous page)

```
# Transitive Dependencies:
'log4cplus'
```

source

```
#include <log4cplus/loggingmacros.h>
#include <ciiLogManager.hpp>
#include <ciiLogMessageBuilder.hpp>
#include <ciiLogConfigurator.hpp>
#include <ciiLogAudience.hpp>
#include <ciiException.hpp>
```

9.3.3 Python

To use the logging API from Python, the following directives are needed:

wscript (project)

```
(nothing to be done, just require 'cxx python cii')
```

wscript (module)

```
(nothing to be done)
```

source

```
import logging
from elt.log import CiiLogManager
from elt.log import CiiLogMessageBuilder
from elt.log import CiiLogAudience
```

9.3.4 Java

source

```
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.Level;
import elt.log.CiiLogManager;
import elt.log.CiiLogMessageBuilder:
import elt.log.CiiLogAudience;
import elt.log.layout.CiiSimpleLayout;
import elt.log.layout.CiiLayout;
```

9.4 Logging Library Usage

This section describes the CII Logging Client API, and explains through examples how to configure and use the CII Logging API library in an application.

The CII Log Client API is split between two classes. The CiiLogManager and the CiiLogMessage-Builder.

Because of the fact that Cii Log Client API leverages on different libraries for Java (Log4j), C++ (Log4cplus) and Python (Python logging), and the constraints this imposes due to the different implementations and APIs, the CII Log client API methods signatures vary across languages and some methods are only available for certain languages.

A complete description of the CII Log Client API for Java, C++, and Python can be found in Appendix A.

9.4.1 CiiLogManager

This is the main class of the CII Log Client API. It provides static methods for configuring and initializing the application's logging, retrieving, creation of logger objects used to log messages, or flushing of the post-mortem appender.

The loggers retrieved using the CiiLogManager are used according to the APIs of the specific logging frameworks for each language [4][5][6].

9.4.2 CiiLogMessageBuilder

The CiiLogMessageBuilder is aimed at helping users to create correctly formatted log messages with extended information, such as:

- Audience
- Tracing context
- CII Error messages
- User-defined log ID

See section 4.7 for usage details and examples.

9.4.3 Logging Configuration

Each of the three logging libraries CII Log Client API provides different ways of configuration. The following sections explain how logging is configured for Java, C++, and Python.

If logging is initialized without providing any custom configuration, a default configuration will be used. This configuration will have:

- A root logger configured to an ERROR level with one appender attached.
- An stdout appender using a CiiSimpleLayout layout.

Thus, if using default configuration, logs will be output to console only.

The configuration is set through the CII Log Manager. The python version of the Log Manager additionally allows to also configure the C++ logging, to give python applications control over the logging of underlying C++ libraries. To do this, the developers passes both a python Logging configuration as well as C++ logging config to the Python Log Manager.

Examples of default configuration files for Log4j, Log4cplus, and Python are shown in sections 4.3.1, 4.3.2, and 4.3.2.

> It is relevant to notice the default path for any file creation or search is the compilation folder generated when the 'waf build' command is called. So any developer string containing a specific path file should be relative to the project build directory.

Java

Due to extensions to the Log4j library, only a subset of the configuration options is supported when using CII Log Client API. There is no support for configuring logging using Log4j's ConfigurationBuilder.

The following file configuration options are available:

- XML
- Properties
- JSON
- YAML

By default, Log4j will look for a file called log4j2.xml in the classpath. If this file is not found, it will fall back to the default CII logging configuration. If there's any issue creating the default CII configuration, it will fall back to the default Log4j configuration, which consists of a root logger with an ERROR level and a console appender attached to it.

It is possible to specify which configuration file will be loaded by Log4j by setting the system property log4j.configurationFile to the path of the configuration file.

It can be passed as a command line argument as:

```
-Dlog4j.configurationFile=path/to/log4j2configFile.xml
```

Or set at runtime before Log4j has been initialized:

```
System.setProperty("log4j.configurationFile", "path/to/log4jconfigFile.xml")
```

It is also possible to initialize a custom configuration dynamically using the Log4j Configurator class:

```
Configurator.initialize("configuration_name", "path/to/log4jconfigFile.xml")
```

Below it is shown how an XML with a configuration equivalent to the default CII configuration would like.

Listing 4-1a Example Java default configuration

```
<?xml version="1.0" ?>
<Configuration name="1584706371399" status="WARN">
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      <CiiSimpleLayout/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="ERROR" includeLocation="true">
      <AppenderRef ref="ConsoleAppender"/>
    </Root>
  </Loggers>
</Configuration>
```

The next example shows how to enable distributed logging, with log transmission to a central log collector host.

Listing 4-1b Example Java configuration with log tranmission

```
<?xml version="1.0" ?>
<Configuration name="1584706371399" status="WARN">
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      <CiiSimpleLayout/>
    </Console>
    <Syslog name="SyslogAppender" port="514" protocol="UDP">
      <CiiLayout/>
    </Syslog>
  </Appenders>
  <Loggers>
    <Root level="ERROR" includeLocation="true">
      <AppenderRef ref="ConsoleAppender"/>
      <AppenderRef ref="SyslogAppender"/>
    </Root>
  </Loggers>
```

(continues on next page)

(continued from previous page)

```
</Configuration>
```

Note the use of CiiLayout and CiiSimpleLayout. These are loaded as plugins by Log4j on initialization. It is required for this to work as intended that class CiiLogManager is loaded by the classloader before Log4j is initialized. CII Log layouts are described in more detail in section 4.4.

To troubleshoot the initialization, it is possible to enable the debug level on the internal Log4j StatusLogger by setting system property log4j2.debug. If this property is set, Log4j will output its internal logs to console.

```
-Dlog4j2.debug=true
```

C++

In addition to the default configuration, the CII Log Client API provides support for custom configuration of the Log4cplus logging framework. Two configuration options are available:

- Properties object
- Properties file

The CiiLogManager class offers methods that can take either an object or a file containing Log4cplus configuration options. It also provides a method without arguments that uses the default CII configuration. Listing 4-2a shows an example properties configuration equivalent to the default one.

Listing 4-2a Example C++ default configuration

```
log4cplus.appender.ConsoleAppender=log4cplus::ConsoleAppender
log4cplus.appender.ConsoleAppender.layout= elt::log::layout::CiiSimpleLayout

log4cplus.rootLogger=ERROR, ConsoleAppender
```

The next example shows how to enable distributed logging, with log transmission to a central log collector host.

Listing 4-2b Example C++ configuration with log transmission

```
log4cplus.appender.ConsoleAppender=log4cplus::ConsoleAppender
log4cplus.appender.ConsoleAppender.layout= elt::log::layout::CiiSimpleLayout

log4cplus.appender.SyslogAppender=log4cplus::SysLogAppender
log4cplus.appender.SyslogAppender.layout=elt::log::layout::CiiLayout

log4cplus.rootLogger=ERROR, ConsoleAppender, SyslogAppender
```

The next example shows how to log to a local file at a custom path writable by the user.

Listing 4-2c Example C++ configuration with log file

```
log4cplus.appender.ConsoleAppender=log4cplus::ConsoleAppender
log4cplus.appender.ConsoleAppender.layout= elt::log::layout::CiiSimpleLayout

log4cplus.appender.LogfileAppender=log4cplus::RollingFileAppender
log4cplus.appender.LogfileAppender.File=/tmp/custom-log-file.log
log4cplus.appender.LogfileAppender.MaxFileSize=16MB
log4cplus.appender.LogfileAppender.MaxBackupIndex=5
log4cplus.appender.LogfileAppender.layout=elt::log::layout::CiiLayout

log4cplus.rootLogger=ERROR, ConsoleAppender, LogfileAppender
```

Detailed CiiLogManager API can be found in Appendix A.

Python

Four configuration modes are supported:

- Default CII Log configuration
- Custom dictionary-based configuration
- Custom file configuration
- Configuration of the LOG4CPLUS logger used by underlying C++ libraries

If a file configuration is used, the file should contain the equivalent to a dictionary based configuration.

In Listing 4-3a an example of a configuration equivalent to the default configuration can be seen.

Listing 4-3a Example Python default configuration

```
'version': 1,
'cii_log_buffer': {
    'enabled': False,
    'ttl_ms': 60000
},
'handlers': {
    'cii-console': {
        'class': 'logging.StreamHandler',
        'formatter': 'cii-brief',
        'stream': 'ext://sys.stdout',
        'level': 'ERROR'
    }
},
'formatters': {
    'cii-brief': {
         'format': 'CII_SIMPLE_LOG_PATTERN',
         'datefmt': 'CII_LOG_PATTERN_DATE'
     }
},
'root': {
```

(continues on next page)

(continued from previous page)

```
    'level': 'ERROR',
    'handlers': ['cii-console']
}
```

The next example shows how to enable distributed logging, with log transmission to a central log collector host.

Listing 4-3b Example Python configuration with log transmission

```
'version': 1,
'handlers': {
    'cii-syslog': {
        'class': 'logging.handlers.SysLogHandler',
        'formatter': 'cii',
        'address': '/dev/log'
    }
},
'formatters': {
    'cii': {
         'format': 'CII_LOG_PATTERN',
         'datefmt': 'CII_LOG_PATTERN_DATE'
     }
},
'root': {
    'level': 'INFO',
    'handlers': ['cii-syslog']
}
```

The next example shows how to log to a local file at a custom path writable by the user.

Listing 4-3c Example Python configuration with log file

```
'version': 1,
'handlers': {
    'logfile': {
        'class': 'logging.handlers.RotatingFileHandler',
         'backupCount": 5,
         'filename': '/tmp/custom-log-file.log',
         'formatter': 'cii',
         'maxBytes': 10485760
    }
},
'formatters': {
    'cii': {
         'format': 'CII_LOG_PATTERN',
         'datefmt': 'CII_LOG_PATTERN_DATE'
     }
},
'root': {
```

(continues on next page)

(continued from previous page)

```
    'level': 'DEBUG',
    'handlers': ['logfile']
}
```

The next example shows how a python application can forward logs to C++ logging, allowing a joint log of the Python application and its underlying (via Python bindings) C++ libraries.

Listing 4-3d Example Python configuration with joint C++ logging

```
'version': 1,
'handlers': {
    'forward-handler': {
        'class': 'elt.log.CiiLogHandler',
        'cii_logger_name': 'name_of_cpp_logger'
     }
},
'root': {
    'level': 'INFO',
    'handlers': ['forward-handler']
}
```

Detailed API can be found in Appendix A.

9.4.4 Log Layouts

Two CII Log Layouts are provided:

- CiiLayout
- CiiSimpleLayout

These layouts are described in section 2.3.

Examples on how to use the CII Log Layouts can be found in the configurations shown in section 4.3.

Java

The CII Log Layouts are provided as Log4j plugins, implemented by the classes CiiLayout and CiiSimpleLayout. They can be used as any other Log4j layout plugin. Usage example can be found in Listing 4-1.

C++

The CII Log Layouts are provided as Log4cplus plugins, implemented by the classes CiiLayout and CiiSimpleLayout. They can be used as any other Log4j layout plugin. Usage example can be found in Listing 4-2.

Python

To use the CII Log Layouts, the cii and cii-brief formatters can be used to enable the CiiLogLayout and the CiiSimpleLayout respectively. The definition of those formatters is injected in the configuration by the CiiLogManager configuration methods. Usage example can be found in Listing 4-3.

9.4.5 Basic usage example with default configuration

The following examples show a basic usage workflow of the CII Logging API to produce log messages of different levels using a default CII configuration, as described in 4.3.

Java

This example can be found in the cii-demo repository: run-logging-example-default-config

Listing 4-4 Java basic example

```
public class LoggingBasicCustomExample {

    public static void main(String[] args) {

        /* Automatically initialize the logging framework and get the root
    * logger.
         * No configuration is provided so a default configuration is used.
         * This default configuration has two appenders: one that outputs to
         * stdout using CiiSimpleLogLayout, and another one that outputs to a
         * rolling file in /var/tmp/elt directory.
         * Only log events of levels ERROR and FATAL will be output to the
         * appenders.
         */
        Logger rootlogger = CiiLogManager.getLogger("");
```

(continues on next page)

(continued from previous page)

```
        /* Send INFO level message to root logger
         * This log will be discarded since the minimum log level for the root
         * logger is ERROR
         */
        rootlogger.log(Level.INFO, "First root logger message");

        /* Send ERROR level message to root logger
         * This log will be passed to the appenders for processing, since the
         * minimum log level for the root logger is ERROR
         */
        rootlogger.log(Level.ERROR, "Second root logger message");

        /* Get a logger with name 'elt.log.test'. There's no logger configured
         * with that name, therefore it will be created inheriting the
         * configuration of it's nearest ancester in the loggers named
         * hierarchy. In this case, since there is only root logger
         * configured, the configuration of the root logger will be used.
         */
        Logger logger = CiiLogManager.getLogger("elt.log.test");

        /* Send INFO level message to elt.log.test logger
         * This log will be discarded since the minimum log level is ERROR
         */
        logger.log(Level.INFO, "First elt.log.test logger message");

        /* Send ERROR level message to elt.log.test logger
         * This log will be passed to the appenders for processing, since the
         * minimum log level is ERROR
         */
        logger.log(Level.ERROR, "Second elt.log.test logger message");
    }
}
```

The logs output produced by this example would be the following in stdout:

```
2020-03-19T20:05:03.104+0100, ERROR, elt.LoggingBasicDefaultExample/main, Second␣
↪root logger message
2020-03-19T20:05:12.571+0100, ERROR, elt.LoggingBasicDefaultExample/main, Second␣
↪elt.log.test logger message
```

And in log file in logsink directory:

```
Cref=LoggingBasicDefaultExample.java-31, Date=2020-03-19T20:05:03.104+0100,␣
↪HostName=docker-eso-eelt, LogType=ERROR, SourceID=elt.
↪LoggingBasicDefaultExample/main, LogID=elt.log.CiiLogManager.main-31,␣
↪Message=Second root logger message
Cref=LoggingBasicDefaultExample.java-49, Date=2020-03-19T20:05:12.571+0100,␣
↪HostName=docker-eso-eelt, LogType=ERROR, SourceID=elt.
↪LoggingBasicDefaultExample/main, LogID=elt.log.test.main-49, Message=Second␣
↪elt.log.test logger message
```

(continues on next page)

(continued from previous page)

C++

This example can be found in the cii-demo repository: logging-example-app

Listing 4-5 C++ basic example

```
int main(int ac, char *av[]) {
  try {
    /* Automatically initialize the logging framework and get the root logger.
     * No configuration is provided so a default configuration is used. This
     * default configuration has two appenders: one that outputs to stdout
     * using CiiSimpleLogLayout, and another one that outputs to a rolling
     * file in /var/tmp/elt directory
     * Only log events of levels ERROR and FATAL will be output to the
     * appenders.
    */
    log4cplus::Logger root_logger = ::elt::log::CiiLogManager::GetLogger();

    std::vector<log4cplus::SharedAppenderPtr> appenders = root_logger.
↪getAllAppenders();
    for (const auto& appender: appenders) {
       std::cout << "Appender:" << appender->getName() << '\n';
    }

    /* Send INFO level message to root logger
     * This log will be discarded since the minimum log level for the root
     * logger is ERROR
     */
    root_logger.log(log4cplus::INFO_LOG_LEVEL, "First root logger message");

    /* Send ERROR level message to root logger
     * This log will be passed to the appenders for processing, since the
     * minimum log level for the root logger is ERROR
     */
    root_logger.log(log4cplus::ERROR_LOG_LEVEL, "Second root logger message");

    /* Get a logger with name 'elt.log.test'. There's no logger configured
     * with that name, therefore it will be created inheriting the
     * configuration of it's nearest ancester in the loggers named hierarchy.
     * In this case, since there is only root logger configured, the
     * configuration of the root logger will be used.
     */
    log4cplus::Logger logger = ::elt::log::CiiLogManager::GetLogger("elt.log.test
↪");

    /* Send INFO level message to elt.log.test logger
```

(continues on next page)

(continued from previous page)

```
     * This log will be discarded since the minimum log level is ERROR
     */
    root_logger.log(log4cplus::INFO_LOG_LEVEL,
                    "First elt.log.test logger message");

    /* Send ERROR level message to elt.log.test logger
     * This log will be passed to the appenders for processing, since the
     * minimum log level is ERROR
     */
    root_logger.log(log4cplus::ERROR_LOG_LEVEL,
                    "Second elt.log.test logger message");

    return 0;
  } catch (const ::elt::error::CiiException& ex) {
    std::cerr << "CiiException occured while executing sample code. What: "
      << ex.what() << '\n';
  }
  return -1;
}
```

This would produce the following output on stdout:

```
2020-04-19T21:49:41.894+0200, ERROR, root/140400261358272, Second root logger␣
↪message
2020-04-19T21:49:41.894+0200, ERROR, root/140400261358272, Second elt.log.test␣
↪logger message
```

And this one on the log file in logsink directory:

```
Cref=../cpp/logging-app/src/logging-app.cpp-35, Date=2020-04-19T21:49:41.
↪894+0200, HostName=docker-eso-eelt, LogType=ERROR, SourceID=140400261358272,␣
↪LogID=root.main-35, Message=Second root logger message
Cref=../cpp/logging-app/src/logging-app.cpp-53, Date=2020-04-19T21:49:41.
↪894+0200, HostName=docker-eso-eelt, LogType=ERROR, SourceID=140400261358272,␣
↪LogID=root.main-53, Message=Second elt.log.test logger message
```

Python

This example can be found in the cii-demo repository: cii-logging-example-app-py

Listing 4-6 Python basic example

```
def main():
    """
    Main application code
    @return status code, int
    """
```

(continues on next page)

(continued from previous page)

```
    result = 0
    try:
        # Automatically initialize the logging framework and get the root
        # logger.
        # No configuration is provided so a default configuration is used.
        # This default configuration has two handlers: one that outputs to
        # stdout using elt.log.CiiLogConfigurator.CII_SIMPLE_LOG_PATTERN and
        # another one that outputs to a rolling file in /var/tmp/elt
        # directory.
        # Only log events of levels ERROR and FATAL will be output to the
        # handlers.
        root_logger = CiiLogManager.get_logger()

        # Send INFO level message to root logger
        # This log will be discarded since the minimum log level for the root
        # logger is ERROR
        root_logger.info("First root logger message")

        # Send ERROR level message to root logger
        # This log will be passed to the handlers for processing, since the
        # minimum log level for the root logger is ERROR
        root_logger.error("Second root logger message")

        # Get a logger with name 'elt.log.test'. There's no logger configured
        # with that name, therefore it will be created inheriting the
        # configuration of it's nearest ancestor in the loggers hirearchy. In
        # this case, since there is only root logger configured,
        # the configuration of the root logger will be used
        logger = CiiLogManager.get_logger("elt.log.test")

        # Send INFO level message to elt.log.test logger.
        # This log will be discarded since the minimum log level is ERROR
        logger.info("First elt.log.test logger message")

        # Send ERROR level message to elt.log.test logger
        # This log will be passed to the handlers for processings, since the
        # minimum log level is ERROR
        logger.error("Second elt.log.test logger message")
    except Exception as e:
        print('Exception occured while executing sample code: %s', e)
        result = 5
    return result
if __name__ == '__main__':
    sys.exit(main())
```

This would produce the following output on stdout:

```
2020-04-19T21:49:41.894+0200, ERROR, cii-logging-example-app-py/MainThread,␣
↪Second root logger message
```

(continues on next page)

(continued from previous page)

```
2020-04-19T21:49:41.894+0200, ERROR, cii-logging-example-app-py/MainThread,␣
↪Second elt.log.test logger message
```

And this one on the log file in logsink directory:

```
Cref cii-logging-example-app-py.main-35, Date=2020-04-19T21:49:41.894+0200,␣
↪HostName=docker-eso-eelt, LogType=ERROR, SourceID= cii-logging-example-app-py/
↪MainThread, LogID=main-35, Message=Second root logger message
Cref= cii-logging-example-app-py.main-50, Date=2020-04-19T21:49:41.894+0200,␣
↪HostName=docker-eso-eelt, LogType=ERROR, SourceID= cii-logging-example-app-py/
↪MainThread, LogID=main-50, Message=Second elt.log.test logger message
```

9.4.6 Basic usage example with custom configuration

This section shows how the use a custom configuration. In this case, we have a configuration with two different loggers, and each logger has a different appender, as described in Table 4-1.

Table 4-1 Example loggers and appenders

Logger name	Logger level	Appender type
“” (root)	WARN	File
“elt.log.test”	TRACE	Console

This configuration can be useful if, for example, we want to send all logs of severity WARN or higher to the logsink directory, but we are trying to troubleshot a bug in package elt.log.test, therefore we want logs of severity as low as TRACE to be output as well for that specific package.

In certain cases, it is necessary or useful for a developer to be able to change the existing logging configuration at runtime. In the C++ and Python cases, the respective log4cplus and Python logging APIs are unaffected by the CII Log extensions of those frameworks, and appropriate documentation on runtime configuration of the frameworks is available in their respective APIs documentation [4][5][6]. In the case of Java, due to the implementation details of the postmortem buffer, it is not possible to change levels or filters of the loggers or appenders if it is enabled. If the postmortem buffer is not enabled the Log4j API is unaffected and can be modified according to the Log4j API documentation [4]. More details on the limitations on the usage of the Log4 API are described in section 4.9.1.

The example code and configurations for each language and outputs produced by the examples from section 4.4 with these configurations are shown below. These examples can be also found in logging-examples module.

Java

The example shown in Listing 4-7 uses the configuration shown in Listing 4-8.

This example can be found in the cii-demo repository: run-logging-example-custom-config

Listing 4-7 Java example app with custom configuration

```
public class LoggingBasicExampleCustomConfig {

    public static void main(String[] args) {

        /* Logging is initialized with custom configuration
    * custom-example-log4j2-config.xml located in resources directory.
         * This custom configuration has two loggers:
         *     - root: With a INFO level and a rolling file appender to
         *       /var/tmp/elt directory
         *     - "elt.log.test": With a TRACE level, a console appender, and
         *       additivity enabled
         *
         * It is also possible to provide the path of the configuration file
         * as system property log4j.configurationFile. This can be achieved
         * passing it as an argument when running the application:
         *     $ java -Dlog4j.configurationFile=<path/to/config/file>
         * Or setting it programmatically before Log4j is initialized:
         *     System.setProperty(
         *             "log4j.configurationFile", "path/to/config/file");
         */
        Configurator.initialize("custom-config",
               "java/src/resources/custom-example-log4j2-config.xml");

        /* Get the root logger.
         */
        Logger rootlogger = CiiLogManager.getLogger("");

        /* Send INFO level message to root logger
         * This log will be output to the log file in logsink directory
         */
        rootlogger.log(Level.INFO, "First root logger message");

        /* Send TRACE level message to root logger. This log message will be
         * discarded due to the root logger having a threshold level of WARN
         */
        rootlogger.log(Level.TRACE, "Second root logger message");

        /* Get a logger with name 'elt.log.test'
         */
        Logger logger = CiiLogManager.getLogger("elt.log.test");

        /* Send TRACE level message to elt.log.test logger
         * This log will be output to console since the minimum log level is
```

(continues on next page)

(continued from previous page)

```
         * TRACE.
         * It will be also passed to its parent logger (root) since additivity
         * property of this logger is set to true. The root logger will accept
         * it also, despite of its level being set to INFO. This is due to the
         * fact that the logger level threshold is only checked at the logger
         * where the message is created. If a threshold level would be set to
         * on the appender attached to the root logger, that one would be
         * applied.
         */
        logger.log(Level.TRACE, "First elt.log.test logger message");

        /* Send ERROR level message to elt.log.test logger
         * This log will be output to console since the minimum log level is
         * TRACE. It will be also passed to its parent logger (root) since
         * additivity property of this logger is set to true. The root logger
         * will also pass it to the logsink file appender.
         */
        logger.log(Level.ERROR, "Second elt.log.test logger message");
    }
}
```

Listing 4-8 Log4j custom configuration

```
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <CiiSimpleLayout/>
    </Console>
    <RollingFile name="FileLogsink" append="true"
      createOnDemand="true"
      fileName="/var/tmp/elt/custom-example.log"
      filePattern="/var/tmp/elt/custom-example.%i.log">
      <Policies>
        <SizeBasedTriggeringPolicy/>
      </Policies>
      <CiiLayout/>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Logger name="elt.log.test" level="TRACE" additivity="true">
      <AppenderRef ref="Console"/>
    </Logger>
    <Root level="INFO">
      <AppenderRef ref="FileLogsink"/>
    </Root>
  </Loggers>
</Configuration>
```

This configuration would produce the following output on console:

```
2020-04-19T23:32:38.600+0200, TRACE, elt.log.examples.
↪LoggingBasicExampleCustomConfig/main, First elt.log.test logger message
2020-04-19T23:32:38.600+0200, ERROR, elt.log.examples.
↪LoggingBasicExampleCustomConfig/main, Second elt.log.test logger message
```

And the following output on the the custom-example.log file in the logsink directory:

```
Cref=LoggingBasicExampleCustomConfig.java-38, Date=2020-04-19T23:32:38.597+0200,␣
↪HostName=docker-eso-eelt, LogType=INFO, SourceID=elt.log.examples.
↪LoggingBasicExampleCustomConfig/main, LogID=.main-38, Message=First root␣
↪logger message
Cref=LoggingBasicExampleCustomConfig.java-58, Date=2020-04-19T23:32:38.600+0200,␣
↪HostName=docker-eso-eelt, LogType=TRACE, SourceID=elt.log.examples.
↪LoggingBasicExampleCustomConfig/main, LogID=elt.log.test.main-58,␣
↪Message=First elt.log.test logger message
Cref=LoggingBasicExampleCustomConfig.java-66, Date=2020-04-19T23:32:38.600+0200,␣
↪HostName=docker-eso-eelt, LogType=ERROR, SourceID=elt.log.examples.
↪LoggingBasicExampleCustomConfig/main, LogID=elt.log.test.main-66,␣
↪Message=Second elt.log.test logger message
```

C++

This example can be found in the cii-demo repository: logging-example-customconfig-app

Listing 4-9 C++ example app with custom configuration

```
int main(int ac, char *av[]) {
  try {
    /*
     * Logging is initialized with custom configuration.
     *
     * This custom configuration has two loggers:
     *     - root: With a INFO level and a rolling file appender to
     *             /var/tmp/elt directory
     *     - "elt.log.test": With a TRACE level, a console appender, and
     *                       additivity enabled
     */
    ::elt::log::CiiLogManager::Configure("log4cplus-test.prop");

    /* Get the root logger.*/
    log4cplus::Logger root_logger = ::elt::log::CiiLogManager::GetLogger();

    /* Send INFO level message to root logger
     * This log will be output to the log file in logsink directory
     */
    root_logger.log(log4cplus::INFO_LOG_LEVEL, "First root logger message");

    /* Send TRACE level message to root logger. This log message will be
```

(continues on next page)

(continued from previous page)

```
     * discarded due to the root logger having a threshold level of INFO
     */
    root_logger.log(log4cplus::TRACE_LOG_LEVEL, "Second root logger message");

    /* Get a logger with name 'elt.log.test' */
    log4cplus::Logger logger = ::elt::log::CiiLogManager::GetLogger("elt.log.test
↪");

    /*
     * Send TRACE level message to elt.log.test logger
     * This log will be outputed to console since the minimum log level is
     * TRACE.
     * It will be also passed to its parent logger (root) since additivity
     * property of this logger is set to true. The root logger will accept it
     * also, despite of its level being set to INFO. This is due to the fact
     * that the logger level threshold is only checked at the logger where the
     * message is created.
     * If a threshold level would be set to on the appender attached to the
     * root logger, that one would be applied.
     */
    logger.log(log4cplus::TRACE_LOG_LEVEL,
               "First elt.log.test logger message");

    /*
     * Send ERROR level message to elt.log.test logger
     * This log will be output to console since the minimum log level is
     * TRACE.
     * It will be also passed to its parent logger (root) since additivity
     * property of this logger is set to true. The root logger will also pass
     * it to the logsink file appender.
     */
    logger.log(log4cplus::ERROR_LOG_LEVEL,
               "Second elt.log.test logger message");

    return 0;
  } catch (const ::elt::error::CiiException& ex) {
    std::cerr << "CiiException occured while executing sample code. What: "
      << ex.what() << '\n';
  }
  return -1;
```

Listing 4-10 Log4cplus custom configuration

```
log4cplus.rootLogger=INFO, ROLLINGFILE
log4cplus.logger.elt.log.test=TRACE, STDOUT
log4cplus.additivity.elt.log.test=TRUE
log4cplus.appender.STDOUT=log4cplus::ConsoleAppender
log4cplus.appender.STDOUT.layout=elt::log::layout::CiiSimpleLayout
log4cplus.appender.ROLLINGFILE=log4cplus::RollingFileAppender
```

(continues on next page)

(continued from previous page)

```
log4cplus.appender.ROLLINGFILE.File=/home/rfernandez/INTROOT/logsink/custom-
↪example.log
log4cplus.appender.ROLLINGFILE.MaxFileSize=16MB
log4cplus.appender.ROLLINGFILE.MaxBackupIndex=1
log4cplus.appender.ROLLINGFILE.layout=elt::log::layout::CiiLayout
```

This configuration would produce the following output on console:

```
2020-04-19T23:32:38.600+0200, TRACE, elt.log.test/139781441696448, First elt.log.
↪test logger message
2020-04-19T23:32:38.600+0200, ERROR, elt.log.test/139781441696448, Second elt.
↪log.test logger message
```

And the following output on the the custom-example.log file in the logsink directory:

```
Cref=../cpp/logging-customconfig-app/src/logging-customconfig-app.cpp-105,␣
↪Date=2020-04-19T23:32:38.597+0200, HostName=docker-eso-eelt, LogType=INFO,␣
↪SourceID=139781441696448, LogID=root.main-105, Message=First root logger␣
↪message
Cref=../cpp/logging-customconfig-app/src/logging-customconfig-app.cpp-125,␣
↪Date=2020-04-19T23:32:38.600+0200, HostName=docker-eso-eelt, LogType=TRACE,␣
↪SourceID=139781441696448, LogID=elt.log.test.main-125, Message=First elt.log.
↪test logger message
Cref=../cpp/logging-customconfig-app/src/logging-customconfig-app.cpp-134,␣
↪Date=2020-04-19T23:32:38.600+0200, HostName=docker-eso-eelt, LogType=ERROR,␣
↪SourceID=139781441696448, LogID=elt.log.test.main-134, Message=Second elt.log.
↪test logger message
```

Python

This example can be found in the cii-demo repository: cii-logging-example-customconfig-app-py

Listing 4-11 Python example app with custom configuration

```
def main():
    """
    Main application code
    @return status code, int
    """
    result = 0
    try:
        # Logging is initialize with custom configuration
        #
        # This custom configuration has two loggers:
        #   - root: With a INFO level and rolling file handler to
        #           /var/tmp/elt directory
        #   - "elt.log.test": With a TRACE level, a console appender and
```

(continues on next page)

(continued from previous page)

```
        #                       propagation enabled
        CiiLogManager.configure(CONFIG_FILE_NAME)

        # Get the root logger
        root_logger = CiiLogManager.get_logger()

        # Send INFO level message to root logger
        # This log will be forwarded to the lof file in logsink directory
        root_logger.info("First root logger message")

        # Send TRACE level message to root logger.
        # This log message will be discarded due to the
        # root logger having ad threshold level of INFO
        root_logger.trace("Second root logger message")

        # Get a logger with name 'elt.log.test'
        logger = CiiLogManager.get_logger("elt.log.test")

        # Send a TRACE level message to elt.log.test logger
        # This log will be forwarded to console since the minimum log level is
        # TRACE.
        # It will be also passed to its parent logger (root) since propagate
        # property of this logger is set to True (by default). The root logger
        # will also pass it to the logsink file handler. This is due to the
        # fact that the logger level threshold is only checked at the logger
        # where the message is created.
        logger.trace("First elt.log.test logger message")

        # Send ERROR level message to elt.log.test logger
        # This log will be displayed on console since the minimum log level is
        # TRACE.
        # It will be also passed to its parent logger (root) since propagate
        # property of this logger is set to True (by default). The root logger
        # will also pass it to the logsing file handler.
        logger.error("Second elt.log.test logger message")
    except Exception as e:
        print('Exception occured while executing sample code: %s', e)
        result = 5
    return result
if __name__ == '__main__':
    sys.exit(main())
```

Listing 4-12 Python logging custom configuration

```
{
    "version": 1,
    "handlers": {
        "logsink": {
            "class": "logging.handlers.RotatingFileHandler",
```

(continues on next page)

(continued from previous page)

```
            "backupCount": 5,
            "filename": "my_log_file",
            "formatter": "cii",
            "maxBytes": 10485760
        },
        "console": {
            "class": "logging.StreamHandler",
            "formatter": "cii-brief",
            "stream": "ext://sys.stdout"
        }
    },
    "root": {
        "handlers": [
            "logsink"
        ],
        "level": "INFO"
    },
    "loggers": {
        "elt.log.test": {
          "handlers": [
              "console"
          ],
          "level": "TRACE"
        }
    }
}
```

This configuration would produce the following output on console:

```
2020-04-19T23:32:38.600+0200, TRACE, cii-logging-example-customconfig-app-py/
↪MainThread, First elt.log.test logger message
2020-04-19T23:32:38.600+0200, ERROR, cii-logging-example-customconfig-app-py/
↪MainThread, Second elt.log.test logger message
```

And the following output on the eelt-dev-output.log file in the logsink directory:

```
Cref= cii-logging-example-customconfig-app-py.main-92, Date=2020-04-19T23:32:38.
↪597+0200, HostName=docker-eso-eelt, LogType=INFO, SourceID= cii-logging-
↪example-customconfig-app-py/MainThread, LogID=main-92, Message=First root␣
↪logger message
Cref= cii-logging-example-customconfig-app-py.main-108, Date=2020-04-19T23:32:38.
↪600+0200, HostName=docker-eso-eelt, LogType=TRACE, SourceID= cii-logging-
↪example-customconfig-app-py/MainThread, LogID=main-108, Message=First elt.log.
↪test logger message
Cref= cii-logging-example-customconfig-app-py.main-115, Date=2020-04-19T23:32:38.
↪600+0200, HostName=docker-eso-eelt, LogType=ERROR, SourceID= cii-logging-
↪example-customconfig-app-py/MainThread, LogID=main-115, Message=Second elt.log.
↪test logger message
```

9.4.7 Adding optional information to log messages

Some of the fields in a CII log message are optional and have to be explicitly added to a log event by the user.

Via LogManager

The following optional fields can be set directly through the CiiLogManager. They must be set *before* retrieving the loggers, and they will be applied to all subsequent logging in the application.

- AppName: The name of the application (or one particular instance of it)

```
elt::log::CiiLogManager::SetApplicationName("MyApplication");
log4cplus::Logger logger = elt::log::CiiLogManager::GetLogger("MyAppLogger");
```

Via MessageBuilder

The following optional fields can be set through the CiiLogMessageBuilder, letting deveopers create a log message with additional information. Also see CII Log Client API (section X.4.2 of this document).

This optional information can be added to a log message:

- Audience: The intended audience of the log message
- UserLogID: A log ID provided by the user.
- Formatted CiiException: A CII Exception [3] provided by the user that will be formatted into the fields:
  - ErrorMessage
  - ErrorTypeName
  - ErrorDateTime
  - ErrorTrace
- Tracing info: The builder is responsible for extracting the following data from an existing active Tracing context:
  - TraceID
  - SpanID

Once the optional parameters are defined, the user can call the build() method on the builder to produce the formatted string with the selected optional fields.

The builder also provides convenience factory methods for the most common logging use cases: setting a log's audience and logging a formatted CII Error.

Support for tracing information is currently not available for C++ and Python. It will be added in following releases.

Complete API can be found in Appendix A.

Sections 4.7.1, 4.7.2, 4.7.3 show usage examples for the CiiLogMessageBuilder on Java, C++, and Python. All examples follow the same workflow.

First, the language-specific logging frameworks are initialized with the default CII configuration and a logger named myLogger is retrieved.

(How to enable tracing to be added in following versions)

Then, the following logs with optional information are produced:

- A log with a message, a custom user ID, and an audience value.
- A log with a message and a formatted CII exception using the builder pattern.
- A log with a message and a formatted CII exception using the builder's convenience method.
- A log with a message and an audience value using the builder's convenience method.
- A log without message and a formatted CiiException. Since no message is passed, the CiiException error message will be used as log message field.
- A log with current tracing context information. Note that these examples will not output any tracing information, since tracing is not enabled. These examples are meant to illustrate the usage of the CiiLogMessageBuilder.

Java

This example can be found in the cii-demo repository: run-logging-example-message-builder-java

Listing 4-13 Adding optional info to logs (Java)

```
package elt.log.examples;

import elt.error.CiiException;
import elt.error.CiiInvalidTypeException;
import elt.log.CiiLogAudience;
import elt.log.CiiLogMessageBuilder;
import org.apache.logging.log4j.Logger;
import elt.log.CiiLogManager;

public class LoggingMessageBuilderExample {

    public static void main(String[] args) {

        //initialize Log4j with default CII configuration and retieve
        //"myLogger" Logger
        Logger logger = CiiLogManager.getLogger("myLogger");

        //Build and log message with message value, user ID and DEVELOPER
        //audience
```

(continues on next page)

(continued from previous page)

```
        logger.error(CiiLogMessageBuilder.create("Message value")
                             .withUserLogId("user_log_id")
                             .withAudience(CiiLogAudience.DEVELOPER)
                             .build());

        CiiException e =
                    new CiiInvalidTypeException("Exception error message");

        //Build and log message with formatted CiiException using builder
        //pattern
        logger.error(CiiLogMessageBuilder.create()
                             .withMessage("This is and exception")
                             .withCiiException(e)
                             .build());

        //Build and log message with formatted CiiException using the
        //convenience method
        logger.error(CiiLogMessageBuilder.createAndBuildWithException(
                "This is an exception", e));

        //Build and log message with DEVELOPER audience
        logger.error(CiiLogMessageBuilder.createAndBuildWithAudience(
                "Message for DEVELOPER audience", CiiLogAudience.DEVELOPER));

        //Build and log message with formatted CiiException using the
        //convenience method without passing a message field value
        logger.error(
                CiiLogMessageBuilder.createAndBuildWithException(null, e));

        //Build and log a message with tracing context information
        logger.error(CiiLogMessageBuilder.create(
                "This is a log message with tracing info")
                             .withTracingInfo()
                             .build());
    }
}
```

The log output from this code, using the CiiSimpleLayout, would be the following:

Listing 4-14 Output CiiLogMessageBuilder usage example Java

```
2020-03-30T10:05:46.377+0200, ERROR, elt.log.examples.
↪LoggingMessageBuilderExample/main, Message value, Audience=DEVELOPER,␣
↪UserLogID=user_log_id

2020-03-30T10:05:46.401+0200, ERROR, elt.log.examples.
↪LoggingMessageBuilderExample/main, This is and exception, ErrorTypeName=elt.
↪error.CiiInvalidTypeException, ErrorDateTime=1585523146401,␣
↪ErrorMessage=Invalid type detected Exception error message, ErrorTrace=elt.
↪error.CiiInvalidTypeException
```

(continues on next page)

(continued from previous page)

```
    at elt.log.examples.LoggingMessageBuilderExample.
↪main(LoggingMessageBuilderExample.java:23)

2020-03-30T10:05:46.402+0200, ERROR, elt.log.examples.
↪LoggingMessageBuilderExample/main, This is an exception, ErrorTypeName=elt.
↪error.CiiInvalidTypeException, ErrorDateTime=1585523146401,␣
↪ErrorMessage=Invalid type detected Exception error message, ErrorTrace=elt.
↪error.CiiInvalidTypeException
    at elt.log.examples.LoggingMessageBuilderExample.
↪main(LoggingMessageBuilderExample.java:23)

2020-03-30T10:05:46.402+0200, ERROR, elt.log.examples.
↪LoggingMessageBuilderExample/main, Message for DEVELOPER audience,␣
↪Audience=DEVELOPER

2020-03-30T10:05:46.402+0200, ERROR, elt.log.examples.
↪LoggingMessageBuilderExample/main, Invalid type detected Exception error␣
↪message, ErrorTypeName=elt.error.CiiInvalidTypeException,␣
↪ErrorDateTime=1585523146401, ErrorMessage=Invalid type detected Exception␣
↪error message, ErrorTrace=elt.error.CiiInvalidTypeException
    at elt.log.examples.LoggingMessageBuilderExample.
↪main(LoggingMessageBuilderExample.java:23)

2020-03-30T10:05:46.403+0200, ERROR, elt.log.examples.
↪LoggingMessageBuilderExample/main, This is a log message with tracing info
```

C++

This example can be found in the cii-demo repository: logging-example-messagebuilder

Listing 4-15 Adding optional info to logs (C++)

```
int main(int ac, char *av[]) {
  try {
    log4cplus::Logger logger = ::elt::log::CiiLogManager::GetLogger("myLogger");

    //Build and log message with message value, user ID and DEVELOPER audience
    logger.log(log4cplus::ERROR_LOG_LEVEL,
      ::elt::log::CiiLogMessageBuilder::Create().WithMessage("Message value")
        .WithUserLogId("user_log_id")
        .WithAudience(::elt::log::CiiLogAudience::DEVELOPER)
        .Build());

    try {
      CII_THROW(::elt::error::CiiException, "TEST EXCEPTION");
    } catch (const elt::error::CiiException& e) {
      //Build and log message with formatted CiiException using builder
      //pattern
```

(continues on next page)

(continued from previous page)

```
      logger.log(log4cplus::ERROR_LOG_LEVEL,
        ::elt::log::CiiLogMessageBuilder::Create()
       .WithMessage("This is and exception")
          .WithException(e)
          .Build());

      //Build and log message with formatted CiiException using the
      //convenience method
      logger.log(log4cplus::ERROR_LOG_LEVEL,
        ::elt::log::CiiLogMessageBuilder::CreateAndBuildWithException(
                        "This is an exception", e));
    }

    //Build and log message with DEVELOPER audience
    logger.log(log4cplus::ERROR_LOG_LEVEL,
      ::elt::log::CiiLogMessageBuilder::CreateAndBuildWithAudience(
        "Message for DEVELOPER audience",
         ::elt::log::CiiLogAudience::DEVELOPER));

    //Build and log a message with tracing context information
    logger.log(log4cplus::ERROR_LOG_LEVEL,
      ::elt::log::CiiLogMessageBuilder::Create(
           "This is a log message with tracing info")
        .WithTracingInfo(true).Build());

    return 0;
  } catch (const ::elt::error::CiiException& ex) {
    std::cerr << "CiiException occured while executing sample code. What: "
      << ex.what() << '\n';
  }
  return -1;
}
```

The output of the code in Listing 4-15 using the CiiSimpleLayout would be the following:

Listing 4-16 Output CiiLogMessageBuilder usage example C++

```
2020-04-19T21:56:56.130+0200, ERROR, myLogger/140218499314368, Message value,␣
↪Audience=DEVELOPER, UserLogID=user_log_id
2020-04-19T21:56:56.303, ERROR, myLogger/140218499314368, This is and exception,␣
↪ErrorTypeName=elt::error::CiiException, ErrorDateTime=1587333416,␣
↪ErrorMessage=TEST EXCEPTION, ErrorTrace= 0#␣
↪elt::error::CiiException::CiiException(std::string const&) at ../elt-common/
↪cpp/error/client/src/ciiException.cpp:29
 1# main at ../cpp/logging-messagebuilder-app/src/logging-messagebuilder-app.
↪cpp:25
 2# __libc_start_main in /lib64/libc.so.6
 3# 0x0000000000414039 in cii-logging-example-messagebuilder-app
```

(continues on next page)

(continued from previous page)

```
2020-04-19T21:56:56.303+0200, ERROR, myLogger/140218499314368, This is an␣
↪exception, ErrorTypeName=elt::error::CiiException, ErrorDateTime=1587333416,␣
↪ErrorMessage=TEST EXCEPTION, ErrorTrace= 0#␣
↪elt::error::CiiException::CiiException(std::string const&) at ../elt-common/
↪cpp/error/client/src/ciiException.cpp:29
 1# main at ../cpp/logging-messagebuilder-app/src/logging-messagebuilder-app.
↪cpp:25
 2# __libc_start_main in /lib64/libc.so.6
 3# 0x0000000000414039 in cii-logging-example-messagebuilder-app

2020-04-19T21:56:56.303+0200, ERROR, myLogger/140218499314368, Message for␣
↪DEVELOPER audience, Audience=DEVELOPER
2020-04-19 21:56:56.303, ERROR, myLogger/140218499314368, This is a log message␣
↪with tracing info
```

Python

This example can be found in the cii-demo repository: cii-logging-example-messagebuilder-py

Listing 4-17 Adding optional info to logs (Python)

```
def main():
    """
    Main application code
    @return status code, int
    """
    result = 0
    try:
        logger = CiiLogManager.get_logger("myLogger")

        # Build and log message with message value, user ID and DEVELOPER
        # audience
        logger.error(CiiLogMessageBuilder()
                     .with_message("Message value")
                     .with_user_log_id("user_log_id")
                     .with_audience(CiiLogAudience.DEVELOPER)
                     .build())
        try:
            raise CiiException("TEST EXCEPTION")
        except CiiException as e:
            # Build and log message with formatted exception using builder
            # pattern
            logger.error(CiiLogMessageBuilder.create()
                         .with_message("This is an exception")
                         .with_exception(e)
                         .build())
            # Build and log message with formatted exception using convenience
            # method
```

(continues on next page)

(continued from previous page)

```
            logger.error(CiiLogMessageBuilder
                  .create_and_build_with_exception("This is an excaption", e))
        # Build and log message with DEVELOPER audience
        logger.error(CiiLogMessageBuilder
                     .create_and_build_with_audience("Message for DEVELOPER␣
↪audience", CiiLogAudience.DEVELOPER))
        # Build and log a message with tracing context information
        logger.error(CiiLogMessageBuilder
                     .create("This is a log message with tracing info")
                     .with_tracing_info(True)
                     .build())
        # Build and log a message with keyword args
        logger.error(CiiLogMessageBuilder
                     .create_and_build_with_args(message='This is a message',
                                                 user_log_id='USER_LOG_ID',
                                                 audience=CiiLogAudience.DEVELOPER))
    except Exception as e:
        print('Exception occured while executing sample code: %s', e)
        result = 5
    return result

if __name__ == '__main__':
    sys.exit(main())
```

The output of the code in Listing 4-17 using the CiiSimpleLayout would be the following:

Listing 4-18 Output CiiLogMessageBuilder usage example Python

```
2020-04-19T21:56:56.130+0200, ERROR, cii-logging-example-messagebuilder-app-py/
↪MainThread, Message value, Audience=DEVELOPER, UserLogID=user_log_id
2020-04-19T21:56:56.303+0200, ERROR, cii-logging-example-messagebuilder-app-py/
↪MainThread, This is an exception, ErrorTypeName=elt.error.Exceptions.
↪CiiException, ErrorDateTime=1588727172250, ErrorMessage=TEST EXCEPTION,␣
↪ErrorTrace=Traceback (most recent call last):
  File "/home/rfernandez/INTROOT/bin/cii-logging-example-messagebuilder-app-py",␣
↪line 30, in main
    raise CiiException("TEST EXCEPTION")
elt.error.Exceptions.CiiException: TEST EXCEPTION

2020-04-19T21:56:56.303+0200, cii-logging-example-messagebuilder-app-py/
↪MainThread, This is an excaption, ErrorTypeName=elt.error.Exceptions.
↪CiiException, ErrorDateTime=1588727172250, ErrorMessage=TEST EXCEPTION,␣
↪ErrorTrace=Traceback (most recent call last):
  File "/home/rfernandez/INTROOT/bin/cii-logging-example-messagebuilder-app-py",␣
↪line 30, in main
    raise CiiException("TEST EXCEPTION")
elt.error.Exceptions.CiiException: TEST EXCEPTION

2020-04-19T21:56:56.303+0200, ERROR, cii-logging-example-messagebuilder-app-py/
↪MainThread, Message for DEVELOPER audience, Audience=DEVELOPER
```

(continues on next page)

(continued from previous page)

```
2020-04-19T21:56:56.303+0200, ERROR, cii-logging-example-messagebuilder-app-py/
↪MainThread, This is a log message with tracing info
2020-04-19T21:56:56.303+0200, ERROR, cii-logging-example-messagebuilder-app-py/
↪MainThread, This is a message, Audience=DEVELOPER, UserLogID=USER_LOG_ID
```

9.4.8 String formatting

Usage of formatted strings in the printf style is supported in both Java, C++, and Python.

In Java, the String.format method can be used to construct the required message. Also, the Logger instances returned by CiiLogManager.getlogger support string formatting.

The following examples would produce a log with the message “Number: 100”.

```
logger.debug(String.format("Number: %d", 100));
logger.debug("Number: %d", 100);
```

In C++ it is supported through the LOG4CPLUS_<LEVEL>_FMT macros [20], as shown in the following example:

```
LOG4CPLUS_WARN_FMT(root, "Number: %d", 100);
```

In Python it is also supported using the language’s regular string formatting constructs.

```
logger.error("Number: %d", 100)
logger.error("Number: %d" % 100)
```

Compile-time validation

Compile-time validation of the formatted strings is supported for Java and for C++.

In the C++ case, compile time validation is automatically done if the LOG4CPLUS_<LEVEL>_FMT macros [20] are used.

In Java, this is supported through the Checker Framework [17]. This solution doesn’t only apply to formatted Strings used in log messages, but to any formatted string in the source code.

The Format String Checker prevents the use of incorrect format strings in format methods such as System.out.printf, String.format, or the Logger log methods. It warns the developer if an invalid format string is written, and it warns if the other arguments are not consistent with the format string (in number of arguments or in their types).

Checker Framework library is installed in the system as part of srv-support-libs when client-apis module is installed. Once installed it can be used during compilation as shown in the example below that compiles class TestUnitLogStringCheckerTest.java:

```
export CHECKERFRAMEWORK=$CIISRV_ROOT/lib/srv-support-libs/checker-framework-3.2

javac -J-Xbootclasspath/p:$CHECKERFRAMEWORK/javac.jar \
-Xbootclasspath/p:$CHECKERFRAMEWORK/jdk8.jar \
-cp $CHECKERFRAMEWORK/checker-qual.jar:$CIISRV_ROOT/lib/srv-support-libs/*:
↪$CIISRV_ROOT/lib/ \
-processorpath $CHECKERFRAMEWORK/checker.jar \
-processor org.checkerframework.checker.formatter.FormatterChecker \
-Astubs=$CHECKERFRAMEWORK/Logger.astub \
 -source 8 -target 8 TestUnitLogStringCheckerTest.java
```

In Python this is not supported, since it is not compiled.

9.4.9 Using the Postmortem Buffer

This section describes the usage of the postmortem buffer. By default, this buffer is disabled and it is the responsibility of the user to enable it.

The postmortem buffer is configured through two environment variables:

- CII_LOG_BUFFER_ON: If set to true, the postmortem buffer will enabled. If not set, or set to any other value, the postmortem buffer will be disabled.
- CII_LOG_BUFFER_TTL_MS: Time-To-Live in milliseconds of the buffered log events.

Changes to these environment variables, once the logging has been initialized, will not have any effect on the current configuration. In order for it to be applied, logging must be shut down and initialized again.

Programatic and file-based configurations of the postmortem buffer will take precedence over these environment variables.

Due to the differences in implementations and APIs of the logging frameworks CII Log Client API leverages on, the implementation and usage of the postmortem buffer vary from language to language. Configuration and usage instructions are described for each of the three languages in the following subsections.

Note: Enabling the postmortem buffer can make significant performance impact. The impact depends on the number of produced logs. When it is, enabled, all the logging events (regardless of its level) will gather all the log context information (hostname, line number, file name, etc. . .). Gathering this information will take significant time, when a lot of logs are produced.

Java

In addition to the configuration of the postmortem buffer through environment variables, it is possible to configure it using system properties.

- cii.log.buffer.on: This property defines if the buffer is enabled or not. By default, it is disabled. Must be set to true to enable the buffer.
- cii.log.buffer.ttl.ms: This property defines the TTL value in milliseconds for entries in the buffer. Default is 60000 ms (1 min).

These properties must be set before initialization of the Log4j framework in order for them to have effect.

If any of these system properties are defined, it will override the environment variable for that configuration parameter.

Once the logging framework is initialized, it is not possible to enable or disable the postmortem buffer.

If the postmortem buffer is enabled, **once the configuration is initialized it is not possible to perform any changes to the active configuration using the Log4j API**. The Logger.getLevel() method or any method to retrieve levels or filters configuration at runtime can return an incorrect value and should not be relied upon.

The example in Listing 4-19 describes a standard workflow using the postmortem buffer.

In the first place, the cii.log.buffer.on property is set in order to enable the postmortem buffer. Also, the cii.log.buffer.ttl.ms property is set to a custom value of 30 seconds, setting the events TTL in the buffer to that value.

Then, a logger object is obtained with the name of the current class. Since no custom configuration has been defined, it will automatically initialize the logging framework to the default CII configuration. Default configuration only has the root logger configured; therefore, the retrieved object will use the root logger configuration, which filters out logs from levels below ERROR.

After, a log message is produced for every log level, from TRACE to FATAL. Only logs of levels ERROR and FATAL will be logged by the appenders.

Finally, the CiiLogManager.flushBufferedAppender() method is called. This will output all the buffered logs to all appenders that originally rejected them, whether it was filtered out by the logger or by the appender itself. Listing 4-20 shows the console output of the execution of the example. As can be seen, first, the ERROR and FATAL messages are printed to console, and then, when the postmortem buffer is flushed, the rest of the log messages are printed to the console.

This example can be found in the cii-demo repository: run-logging-example-postmortem-buffer-java

Listing 4-19 Example postmortem appender (Java)

```
package elt.log.examples;

import elt.log.CiiLogManager;
import org.apache.logging.log4j.Level;
```

(continues on next page)

(continued from previous page)

```java
import org.apache.logging.log4j.Logger;


public class PostmortemAppenderExample {

    public static void main(String[] args) {

        // Enable the postmortem appender
        // This must be done before initializing Log4j
        System.setProperty("cii.log.buffer.on", "true");
        // Set the time-to-live of the log events in the buffer to 30s
        System.setProperty("cii.log.buffer.ttl.ms", "30000");

        // Automatically initialize the logging framework with console and
        // file appender and get the logger with the name
        // elt.log.examples.PostmortemAppenderExample
        Logger logger = CiiLogManager.getLogger(
                PostmortemAppenderExample.class.getCanonicalName());

        // Create a log message of each level. Only messages of levels ERROR
        // and FATAL will be output to the console and file
        logger.log(Level.TRACE, "Trace message");
        logger.log(Level.DEBUG, "Debug message");
        logger.log(Level.INFO, "Info message");
        logger.log(Level.WARN, "Warn message");
        logger.log(Level.ERROR, "Error message");
        logger.log(Level.FATAL, "Fatal message");

        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Flush the postmortem appender. Logs of levels TRACE, DEBUG, INFO,
        // and WARN will be now output to the console and file
        CiiLogManager.flushPostmortemBuffer();
    }
}
```

This example produces the following output:

Listing 4-20 Console output from postmortem example (Java)

```
2020-03-12T22:33:14.524+0100, ERROR, elt.log.examples.PostmortemAppenderExample/
↪main, Error message
2020-03-12T22:33:14.524+0100, FATAL, elt.log.examples.PostmortemAppenderExample/
↪main, Fatal message
2020-03-12T22:33:14.524+0100, TRACE, elt.log.examples.PostmortemAppenderExample/
↪main, Trace message
```

(continues on next page)

(continued from previous page)

```
2020-03-12T22:33:14.524+0100, DEBUG, elt.log.examples.PostmortemAppenderExample/
↪main, Debug message
2020-03-12T22:33:14.524+0100, INFO, elt.log.examples.PostmortemAppenderExample/
↪main, Info message
2020-03-12T22:33:14.524+0100, WARN, elt.log.examples.PostmortemAppenderExample/
↪main, Warn message
```

Async Logging

Log4j supports asynchronous logging, as described in [11]. It's important to note that when using custom configurations**, the includeLocation property of loggers' configuration must be explicitly set to true**. Otherwise, information as the source code file and line references will be lost. It is also necessary for the postmortem buffer to work as intended since it relies on a log event's context data, which would otherwise be lost.

This also needs to be taken into consideration when flushing the buffer, since it's possible that some of the last produced log events might have not been processed yet by the buffer at the time of flushing when asynchronous logging is used.

C++

In C++, the postmortem buffer can be configured using the environment variables mentioned earlier, as well as programmatically. The CiiLogConfigurator class provides methods to initialize a configuration with the postmortem buffer enabled, as well as a method to enable the postmortem buffer once the configuration has been already initialized.

When the postmortem buffered is flushed, log messages indicating the START and END of the flushed logs will be injected, as can be seen in Listing 4-22, helping the user identify the logs that were buffered by the postmortem facility.

More details on how to configure the postmortem buffer can be found in Appendix A.2.

A usage example of the postmortem buffer in C++ is shown in Listing 4-21. In this example, the postmortem buffer is enabled and a log TTL of 30 seconds is set. Then, logging is initialized with the default configuration. A series of log messages of different levels are produced, and finally, the postmortem buffer is flushed to all the configured appenders.

This example can be found in the cii-demo repository: logging-example-postmortem-buffer-app

Listing 4-21 Example postmortem appender (C++)

```
int main(int ac, char *av[]) {
  try {
    //Enable postmortem appender and set the time-to-live of the log events in
    //the buffer to 30s
    std::string postmortem_enable_env = std::string(
```

(continues on next page)

(continued from previous page)

```
      ::elt::log::CiiLogManager::POSTMORTEM_APPENDER_ENABLED_ENV_VAR) +
      "=true";
    putenv(const_cast<char*>(postmortem_enable_env.c_str()));

    std::string postmortem_ttl_env = std::string(
      ::elt::log::CiiLogManager::POSTMORTEM_APPENDER_ENTRYTTL_MS_ENV_VAR) +
      "=30000";
    putenv(const_cast<char*>(postmortem_ttl_env.c_str()));

    // Automatically initialize the logging framework with console and file
    // appender and get the logger with the name
    // elt.log.examples.PostmortemAppenderExample
    log4cplus::Logger logger = ::elt::log::CiiLogManager::GetLogger(
      "PostmortemAppenderExample");

    // Create a log message of each level
    // Only messages of levels ERROR and FATAL will be output to the console
    // and file
    logger.log(log4cplus::TRACE_LOG_LEVEL, "Trace message");
    logger.log(log4cplus::DEBUG_LOG_LEVEL, "Debug message");
    logger.log(log4cplus::INFO_LOG_LEVEL, "Info message");
    logger.log(log4cplus::WARN_LOG_LEVEL, "Warn message");
    logger.log(log4cplus::ERROR_LOG_LEVEL, "Error message");
    logger.log(log4cplus::FATAL_LOG_LEVEL, "Fatal message");
    std::this_thread::sleep_for(std::chrono::seconds(1));

    // Flush the postmortem appender.
    // Logs of levels TRACE, DEBUG, INFO, and WARN will be now output to the
    // console and file
    ::elt::log::CiiLogManager::FlushPostmortemBuffer();

    return 0;
  } catch (const ::elt::error::CiiException& ex) {
    std::cerr << "CiiException occured while executing sample code. What: "
      << ex.what() << '\n';
  }
  return -1;
}
```

This example produces the following output:

Listing 4-22 Console output from postmortem example (C++)

```
2020-04-19T22:27:50.415+0100, ERROR, PostmortemAppenderExample/140577659003584,␣
↪Error message
2020-04-19T22:27:50.415+0100, FATAL, PostmortemAppenderExample/140577659003584,␣
↪Fatal message
2020-04-19T22:27:51.416+0100, TRACE, root/140577659003584, [START] DUMPING␣
↪POSTMORTEM LOGS
```

(continues on next page)

(continued from previous page)

```
2020-04-19T22:27:50.415+0100, TRACE, PostmortemAppenderExample/140577659003584,␣
↪Trace message
2020-04-19T22:27:50.415+0100, DEBUG, PostmortemAppenderExample/140577659003584,␣
↪Debug message
2020-04-19T22:27:50.415+0100, INFO, PostmortemAppenderExample/140577659003584,␣
↪Info message
2020-04-19T22:27:50.415+0100, WARN, PostmortemAppenderExample/140577659003584,␣
↪Warn message
2020-04-19T22:27:51.416+0100, TRACE, root/140577659003584, [END] DUMPING␣
↪POSTMORTEM LOGS
```

Python

The postmortem buffer can be configured by one of the methods provided by the elt.log.CiiLogManager module. Note that once the buffered logger is configured into the system, it can not be disabled.

The standard Python Logging API is unaffected by the use of the postmortem buffer.

The example in Listing 4-23 shows an example usage of the postmortem buffer in Python.

More details on how to configure the postmortem buffer can be found in Appendix A.3.

An usage example of the postmortem buffer in Python is shown in Listing 4-23. In this example, the postmortem buffer is enabled and a log TTL of 30 seconds is set. Then, logging is initialized with the default configuration. A series of log messages of different level are produced, and finally the postmortem buffer is flushed to all the configured handlers.

This example can be found in the cii-demo repository: cii-logging-example-postmortem-buffer-app-py

Listing 4-23 Example postmortem buffer (Python)

```python
def main():
    """
    Main application code
    @return status code, int
    """
    result = 0
    try:
        # Enable postmortem functionality and set time-to-live of the log events
        # in the buffer to 30s
        os.environ[CiiLogManager.POSTMORTEM_BUFFER_ON_ENV_VAR] = 'true'
        os.environ[CiiLogManager.POSTMORTEM_BUFFER_TTL_MS_ENV_VAR] = '30000'

        # Automatically initialize the logging framework with console and file
        # handler and get the logger with name 'PostmortemBufferExample'
        logger = CiiLogManager.get_logger("PostmortemBufferExample")

        # Create a log message of each level
```

(continues on next page)

(continued from previous page)

```
        # Only messages of levels ERROR and FATAL will be outoput to the console
        # and file

        logger.trace("Trace message")
        logger.debug("Debug message")
        logger.info("Info message")
        logger.warning("Warning message")
        logger.error("Error message")
        logger.fatal("Fatal message")

        # Wait a little
        time.sleep(1)

        print('Flushing postmortem buffer')
        # Flush postmortem buffer, trace, debug and info messages should be seen
        # on stdout
        CiiLogManager.flush_postmortem_buffer()

    except Exception as e:
        print('Exception occured while executing sample code: %s', e)
        result = 5
    return result

if __name__ == '__main__':
    sys.exit(main())
```

This example produces the following output:

```
2020-05-06T11:45:02.407+0100, ERROR, cii-logging-example-postmortem-buffer-app-
↪py/MainThread, Error message
2020-05-06T11:45:02.407+0100, CRITICAL, cii-logging-example-postmortem-buffer-
↪app-py/MainThread, Fatal message
2020-05-06T11:45:02.407+0100, TRACE, cii-logging-example-postmortem-buffer-app-
↪py/MainThread, Trace message
2020-05-06T11:45:02.407+0100, DEBUG, cii-logging-example-postmortem-buffer-app-
↪py/MainThread, Debug message
2020-05-06T11:45:02.407+0100, INFO, cii-logging-example-postmortem-buffer-app-py/
↪MainThread, Info message
2020-05-06T11:45:02.407+0100, WARNING, cii-logging-example-postmortem-buffer-app-
↪py/MainThread, Warning message
```

9.5 Logging services administration

This section describes the services used by logging. These services should be managed and maintained by system administrators.

9.5.1 Logging services configuration

It must be noted that running the services is not needed for being able to use the Logging Client API. For details on how to use the CII Log Client API, see section 3.3.

As of CII v4, the log transmission uses these services:

- rsyslog
- logrotate

Host Setup

During cii-postinstall, an adminstrator chooses a cii-role for the host.

In accordance, cii-postinstall configures the rsyslog of the host.

The roles have the following effects:

Role	Effect
own server	writes to local file elt.log
group client	writes to local file elt.log, and pushes logs to a log-collector-host
group server	writes to local file elt.log, and accepts logs pushed from other hosts

Start/Stop Services

The log services can be started using

```
sudo cii-services start log
```

and stopped using

```
sudo cii-services stop log
```

Configuration

The log file is rotated daily, and 21 days will be kept.

To modify the amount of retained logs, see `/etc/logrotate.d/elt-log` (requires privileges)

9.5.2 Logging services monitoring

To check the status of the logging services, do

```
systemctl status --no-pager rsyslog logrotate
```

To see the service logs, do

```
journalctl -u rsyslog
```

```
journalctl -u logrotate
```

9.6 Logging User Tools

This section describes the different user interface tools that enable users to create, query, and browse CII Log events.

9.6.1 ciiLogSend

Notice: As of CII v4, the alias names `logSend` and `cii-services-log-cli` have been removed.

This section describes how to use the ciiLogSendd command line tool. This tool allows a user to write a log entry to the local CII log service, which will add the log to the logsink directory on the local machine, and can also (depending on the CII role of the host) transmit the log entry to a centralized CII log collector via CII distributed log transport.

This tool takes the Message field value and accepts parameters for all the rest of the CII Log format fields as described in Appendix B. For the required fields, the “N/A” value is used if no value is provided, except for the LogType field, for which the INFO level will be used, and for the HostName field, for which the current hostname will be used. The Date field is populated with the current time. Table 5-1 shows the mappings beteen the sendLog command flags and the CII Log fields.

Table 6-1 ciiLogSend CLI tool flags mappings

Flag	Field name
-h, –help	
-lt, –logtype	LogType
-cr, –cref	Cref
-si, –sourceid	SourceID
-h, –hostname	HostName
-au, –audience	Audience
-uli, –userlogid	UserLogID
-ti, –traceid	TraceID
-spi, –spanid	SpanID
-en, –exceptionname	ErrorTypeName
-ed, –exceptiontimestamp	ErrorDateTime
-em, –errormessage	ErrorMessage
-et, errortrace	ErrorTrace

A basic usage example can be seen below:

```
$ ciiLogSend "This is a log message"
```

Which will create the following log:

```
Cref=N/A, Date=2020-03-27T19:44:57.872+0200, HostName=docker-eso-eelt,␣
↪LogType=INFO, SourceID=N/A, LogID=N/A, Message=This is a log message
```

This is a more advanced example, where several of the log fields are passed as arguments:

```
$ ciiLogSend "This is a log message" -lt ERROR -cr example_code_reference -si  \
example_source_id -li example_log_id -au DEVELOPER
```

Which will produce the following log:

```
Cref=example_code_reference, Date=2020-03-27T19:50:28.739+0200, HostName=docker-
↪eso-eelt, LogType=ERROR, SourceID=example_source_id, LogID=example_log_id,␣
↪Audience=DEVELOPER, Message=This is a log message
```

9.6.2 Log Viewer

This section describes usage of the *ciiLogViewer* application, which is a graphical viewer application for logs. It is a stand-alone version of the CII log viewer widget.

Table 6-2 log viewer command line options

Flag	Description
-h, –help	Shows help
-f, –follow	Display (tail) new local logs

In “follow” mode, it detects all newly created log messages on the local host, found in the local log sink directory, and displays them in an automatically refreshing table.

To run it:

```
ciiLogViewer
```

Where it finds logs:

- The tool by default locates the logs in `/var/log/elt`

To show logs from an arbitrary location:

```
ciiLogViewer /path/to/logs
```

View Filtering

As of CII v4, the log viewer can filter over all columns, using regexp-based query expressions.

```
Syntax:

   <COLUMN NAME 1> [NOT] <LIKE STATEMENT 1> <COLUMN NAME 2> [NOT] <LIKE␣
↪STATEMENT 2> ...

   # where the <LIKE STATEMENT N> is of the form:
```

(continues on next page)

(continued from previous page)

```
   LIKE (<REGULAR EXPRESSION FOR THE COLUMN>)


The NOT operator acts as negation, for example:

    # display records where the host column does not contain "test" (case␣
↪insensitive)
    Host NOT LIKE(test)


The OR operator is implicit, for example:

   Host LIKE(\.eso\.org) Log ID LIKE(^Default) Message NOT LIKE(^MyID$)

   # means:
     Host should contain ".eso.org
     or: Log ID should start from "Default"
     or: Message should be exactly "MyID"


More examples:

   Host LIKE(*.eso.org) - an invalid regular expression, no effect

   Host LIKE(.eso.org) will match "?eso?org"

   Host LIKE(\.eso\.org) will match anything that contains ".eso.org"

   Host LIKE(^elt\.eso\.org$) will match "elt.eso.org" exactly
```

Notes:

1. The relation between all column “filters” is OR
2. The NOT and LIKE keywords must be upper-case
3. Column names are matched case-insensitive, e.g. Host, host, HOST
4. Columns names with white space need not be quoted, e.g. Log ID
5. Filters for non-existing columns are ignored, e.g. Host33 NOT LIKE(.*)

9.6.3 Log Viewer Widget (Qt)

This is a Qt widget, meant for integration into an ELT application, allowing to display and filter ELT log data. The widget is the basis for the Log Viewer application described in the previous section.

9.7 Tracing

This section describes the CII Tracing system, including application instrumentation examples as well as the different services used to manage and browse the tracing data produced by the CII distributed systems.

9.7.1 Introduction

Tracing purpose is to aid developers and engineers in the understanding of interactions between components in a distributed system.

It must be noted that logging and tracing are handled separately and there is no automatic connection between logs and traces. This logical connection must be created manually by the user.

It is supported in CII with the OpenTracing API [22], and Jaeger [23] is used as the OpenTracing implementation.

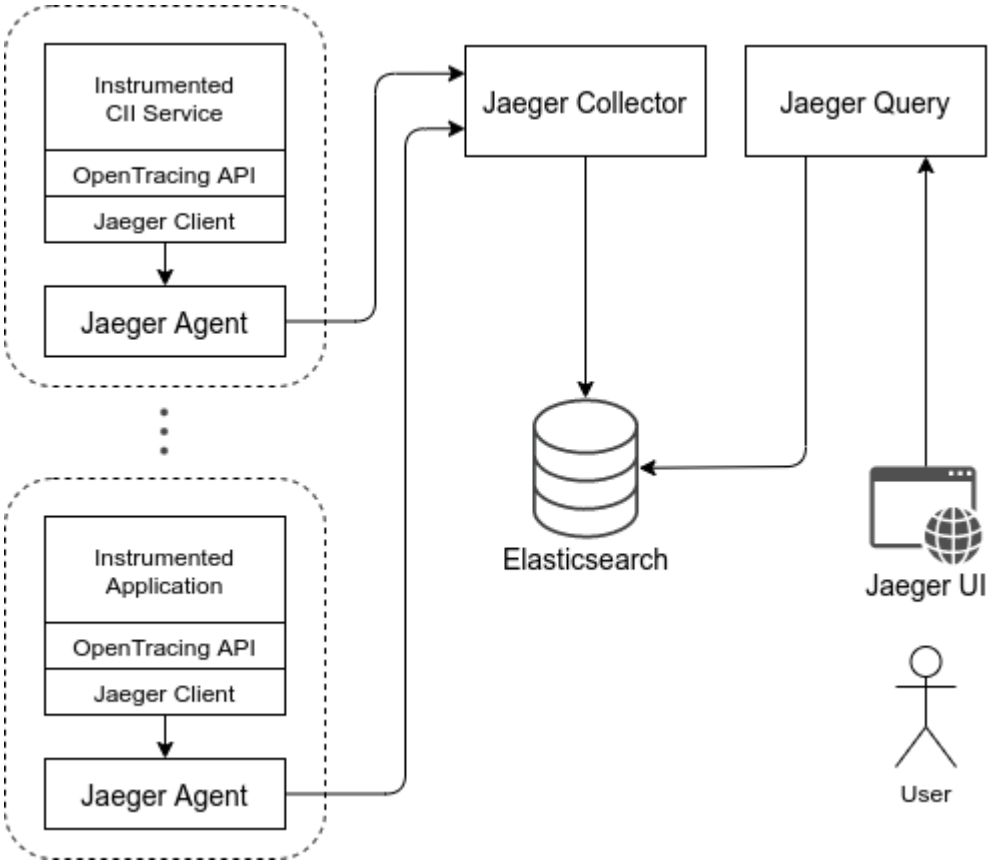


Figure 8-1 CII Tracing architecture

The architecture of the tracing system is shown in the figure above.

It can be divided in:

- Instrumented application using OpenTracing API and Jaeger Client libraries
- Jaeger services
- Elasticsearch persistence backend

The Jaeger services used are:

- Jaeger Agent
- Jaeger Collector
- Jaeger Query

A standard workflow of the tracing system is also shown in Figure 8-1.

Applications and services instrumented with the OpenTracing API produce tracing data, that is sent to a Jaeger Agent by the Jaeger Client library.

The Jaeger Agent collects tracing data from multiple applications and sends batches of data to the Jaeger Collector.

The Jaeger Collector is responsible for persisting the tracing data received from the Jaeger Agents to an Elasticsearch instance.

Finally, the Jaeger Query service provides access to the persisted tracing data, as well as the Jaeger UI, which can be used to browse the tracing data.

Traces and Spans

Distributed tracing relies on traces and spans. A trace is the complete processing of a request. The trace represents the whole execution path of a request as it moves through all of the services or components of a distributed system. All trace events generated by a request share a trace ID that tools use to organize, filter, and search for specific traces.

Each trace is comprised of several spans. A span is an activity or operation that takes place within individual services or components of the distributed system. Each span is another step in the total processing of the overall request. Spans are typically named and timed operations. Spans each carry a unique span ID, and can also carry a reference to their parent span, metadata, or other annotations.

A trace can be seen as a directed acyclic graph, where spans are the nodes and references between spans are the edges.

For example, the following figure illustrates a trace made up of 6 spans:

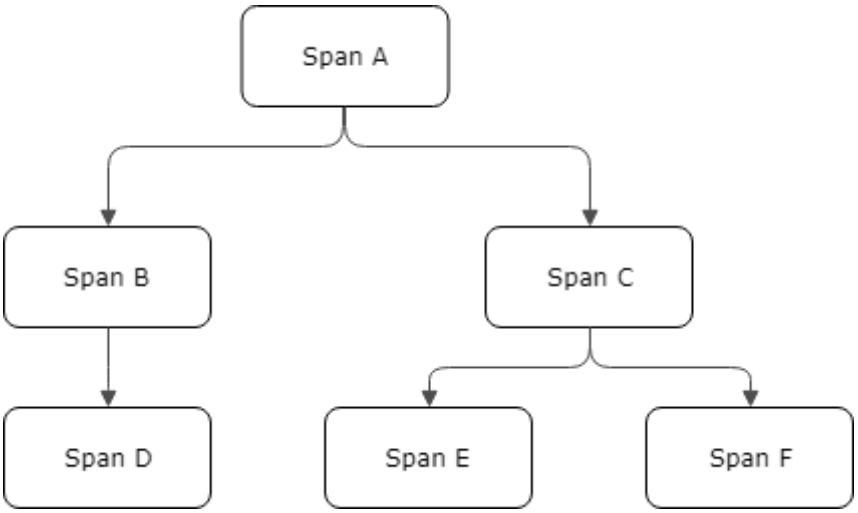


Figure 8-2 Causal relationships between spans in a trace

While this representation allows us to visualize the relationships between the different spans in a trace, it can also be useful to visualize traces in a timeline:

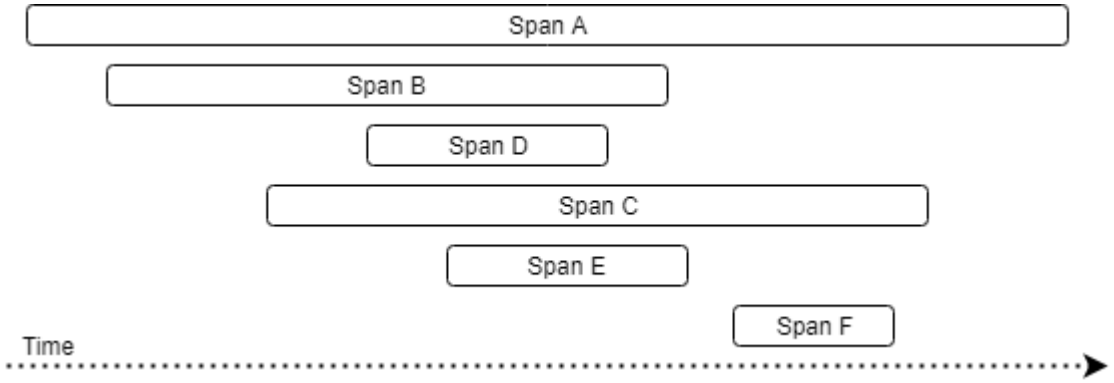


Figure 8-3 Temporal relationships between spans in a trace

Each trace is identified by a unique Trace ID. A traced transaction that spans across multiple hosts, processes, or threads shall have a tracing context propagated between them.

This trace context must contain the following information to be able to trace the call flow of a transaction between services:

- Trace ID: Identifier for a specific operation spanning across one or more distributed systems. When this operation is initiated, a Trace ID is generated and passed to other services that are called as part of the operation.

A trace is composed of one or more spans in one or more distributed hosts. Each span describes a logical unit of work, and contains the following information:

- Operation name: String describing the unit of work represented by the span, e.g. ConfigService::readConfig.
- Span ID: Identifies the span. It must be unique within a trace.

- Parent ID: Identifier that references the ID of the span that triggered the current span.
- Start time: Timestamp indicating when a span was started.
- End time: Timestamp indicating when a span was ended.

The flow of this tracing context information across different services through operation can be seen in Figure 8-4.

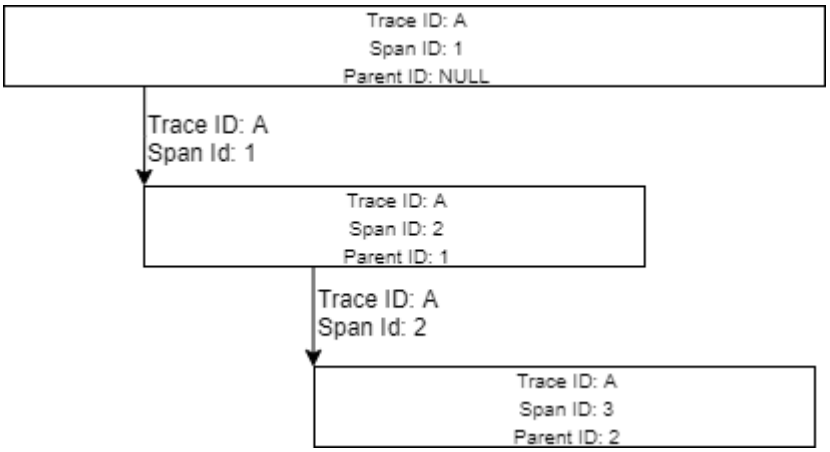


Figure 8-4 Tracing context information flow

In this case, Service 1 would create a tracing context with Trace ID = A, Span ID = 1, and Parent ID = NULL. Together with the call to Service 2, Trace ID and Span ID of the caller are passed. Then Service 2 will create a trace context with Parent ID equal to the Span ID from Service 1 and the Trace ID to the one passed by Service 1. This newly created context will be set as the current context for Service 2. The same procedure applies to the interaction between Service 2 and Service 3 and to subsequent calls to any other service.

9.7.2 Tracing Library Usage

The following sections describe basic tracing instrumentation usage examples.

Advanced usage and configuration of the OpenTracing API and Jaeger are out of the scope of this document. For further information, see [22] and [23].

Tracing request-reply communication

Every request-reply action is instrumented.

When a new request is being issued by the client a new span is created. The MAL middleware implementation must check for existence of an active span using GlobalTracer.get().scopeManager().activeSpan() method. If there is a span active a newly created span Parent Id is set to Span Id of the active span, otherwise Parent Id is to 0. Operation name of the span is set to <instance name>::<method name>, e.g. ConfigService::readConfig. Trace Id and Span Id of

the span are propagated to the server side among other request data. Upon completion or cancelation of the request the Span.finish() method is called on the created span instance.

On the server side a new span is created when request is received. The span shares the same operation name and Trace Id. It is created as a child span of the request span, i.e. Span Id of the span is set to Parent Id the of the request span. Upon completion or cancelation of the reply the Span.finish() method is called on the created span instance. The server-side implementation of a CII request-reply interface can create a child span of the span that CII created using mal tracing utilities by calling elt::mal::util::tracing::startSpan, which will create a child span of an active span if an active span exits, otherwise a childless span will be created.

Tracing publish-subscribe communication

Publish-subscribe communication is not instrumented by default. It needs to be enabled per data entity in ICD, i.e. by setting structure attribute trace to ”true”. This will instruct ICD generator to declare agnostic data entity structure as traceable by extending Traceable interface. The interface provides access to Trace Id and Span Id.

```
interface Traceable {
  int64_getTraceId();
  uint64 getSpanId();
};
```

Figure 8-5 Definition of Traceable interface in pseudo-IDL.

On publisher side the process is the same as on the client side of request-reply communication pattern with exception that the operation name equals “publish(<data entity name>)” and the span finishes as soon as it is published (enqueued).

On the subscriber side it is programmer’s responsibility to create a new span, mal mapping will not create any spans automatically. User needs to extract a tracing context (i.e. sample->getTrace()) and create a span using mal tracing utilities. This decision has been made mostly due to the fact that multiple data entities can be read at once. The Logging API will provide helper method to create and activate a new span from received data entity that implements Traceable, e.g. TracingUtils#createSpan(methodName, traceable).

Includes/Imports

Java

```
import elt.utils.TracingUtils;
import io.opentracing.Span;
import io.opentracing.SpanContext;
import io.opentracing.Tracer;
import io.opentracing.util.GlobalTracer;
```

C++

```
#include <ciiTrace.hpp>
```

Python

```
import elt.trace
```

Java class path dependencies

To support the CII Tracing, the Opentracing and Jaeger libraries need to be added to the Java classpath. It is also important to set the Jaeger service name and sampling type and rate.

Listing 8-1 Java tracing classpath libraries

```
LIBDIR="$CIISRV_ROOT/lib"
SRV_SUPPORT_LIBS="$LIBDIR/srv-support-libs/*"
JAEGER_LIBS="$LIBDIR/srv-support-libs/jaeger/*"
OPENTRACING_LIBS="$MAL_ROOT/lib/opentracing/*"
MALCOMMON_LIB="$MAL_ROOT/lib/mal-common.jar"
export JAEGER_SERVICE_NAME=tracing-example

java -DJAEGER_SAMPLER_TYPE=const -DJAEGER_SAMPLER_PARAM=1 -DJAEGER_SERVICE_NAME=
↪$JAEGER_SERVICE_NAME -cp "$MALCOMMON_LIB:$OPENTRACING_LIBS:$JAEGER_LIBS:$SRV_
↪SUPPORT_LIBS:$LIBDIR/*" elt.trace.examples.TracingBasicExample
```

Basic usage example

In these examples, tracing is initialized using the provided TracingUtils.initializeTracing() helper method, which can be used to initialize the tracing framework and register a global tracer for the current application.

Then a root span is created, and a second span with the root span as its parent.

Finally, after all spans are finished, the tracer is closed.

Java

This example can be found in the cii-demo repository: run-tracing-example-java

Listing 8-2 Tracing basic example (Java)

```
/**
 * CII tracing utility usage
 */
public class TracingBasicExample {

    public static void main(String[] args) {

        /* Initialize tracing framework and register global tracer.
         */
        Tracer tracer = TracingUtils.initializeTracing();

        /* Once the global tracer is registered, it is also
         * possible to obtain an instance from GlobalTracer.
         */
        tracer = GlobalTracer.get();

        /* Start a parent span */
        Span parentSpan = tracer.buildSpan("method1")
                .start();

        /* Start a child span */
        String str;
        Span span = tracer.buildSpan("method2")
                .asChildOf(parentSpan)
                .start();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        /* close the child span */
        span.finish();

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        /* close the parentSpan */
        parentSpan.finish();
```

(continues on next page)

(continued from previous page)

```
        /* close global tracer */
        tracer.close();
    }
}
```

C++

This example can be found in the cii-demo repository: cii-tracing-example-app

Listing 8-3 Tracing basic example (C++)

```
int main(int ac, char *av[]) {
  try {
    /* Initialize tracing framework and register global tracer.*/
    ::elt::trace::InitializeTracing("demo-service-name");

    /* Obtain a tracer (registered in previous step) */
    std::shared_ptr<opentracing::Tracer> tracer = opentracing::Tracer::Global();
    if (!tracer) {
      throw ::elt::error::CiiException("No tracer registered");
    }

    /* Start a parent span */
    std::unique_ptr<opentracing::Span> parentSpan = tracer->StartSpan("method1");

    /* Start a child span */
    std::unique_ptr<opentracing::Span> span = opentracing::Tracer::Global()->
↪StartSpan(
        "method2", { opentracing::ChildOf(&parentSpan->context())});
    std::this_thread::sleep_for(std::chrono::seconds(2));

    /* close the child span */
    span->Finish();
    std::this_thread::sleep_for(std::chrono::seconds(3));

    /* close the parentSpan */
    parentSpan->Finish();

    /* close global tracer */
    tracer->Close();
    return 0;
  } catch (const ::elt::error::CiiException& ex) {
    std::cerr << "CiiException occured while executing sample code. What: "
      << ex.what() << '\n';
  }
  return -1;
}
```

Python

In this example, the use of a span and the tracer as context manager is shown as well.

This example can be found in the cii-demo repository: cii-tracing-example-app-py

Listing 8-4 Tracing basic example (Python)

```
def main():
    """
    Main application code
    @return status code, int
    """
    result = 0
    try:
        # Initialize tracing framework and register global tracer
        elt.trace.initialize_tracing('demo-service-name')

        # Obtain a tracer (registered in previous step)
        tracer = elt.trace.opentracing.Tracer.get_global()
        if tracer is None:
            raise RuntimeError('No tracer registered')

        # Start a parent span
        parent_span = tracer.start_span('method 1')

        # Start a child span
        span = tracer.start_span("method2",
                     [elt.trace.opentracing.child_of(parent_span.context())])

        time.sleep(2)

        # Close the child span
        span.finish()

        # Span as context manager
        with tracer.start_span("method3",
                       [elt.trace.opentracing.child_of(parent_span.context())]):
            time.sleep(1)
        # span.finish() was called on context exit

        time.sleep(1)

        # Close the parent span
        parent_span.finish()

        # Use tracer as context manager that closes itself automatically
        with tracer:
            # make another span and use it as context manager
            with tracer.start_span('another span'):
```

(continues on next page)

(continued from previous page)

```
                time.sleep(1)
        # global tracer is closed a this point
        # to explicitly close the tracer, use: tracer.close()

    except Exception as e:
        print('Exception occured while executing sample code: %s', e)
        traceback.print_exc()
        result = 5
    return result

if __name__ == '__main__':
    sys.exit(main())
```

Mal pub/sub and request/response tracing basic examples

Mal tracing examples are part of icd-demo project and are listed in Table 2.

Table 2: MAL tracing basic examples in icd-demo

Example type	Example location
Mal pub/sub java tracing example	app/java/src/elt/mal/example/ps
Mal pub/sub c++ tracing example	app/cpp/ps/publisher/src app/cpp/ps/subscriber/src
Mal pub/sub python tracing example	app/python/ps/publisher/src app/python/ps/subscriber/src
Mal request/response java tracing example	app/java/src/elt/mal/example/rr
Mal request/response c++ tracing example	app/cpp/rr/client/src app/cpp/rr/server/src
Mal request/response python tracing example	app/python/rr/client/src app/python/rr/server/src

Basic examples for mal pub/sub tracing (based on examples in icd-demo):

Listing 8-5 Tracing mal pub/sub example (C++)

```
/**
  * @brief Initialize tracing (implementation specific to the user
  * environment, mal is using only opentracing API to obtain the tracer)
  *
  *  In this example we are using jaeger implementation of opentracing
  *  standard.
  *
  * @param service_name opentracing service name
  *
  */
```

(continues on next page)

(continued from previous page)

```
void InitializeTracing(const std::string& service_name) {
  const bool log_span = true;
  const bool disabled = false;
  const double always_sample = 1.0;

  /* Configuration specific to jaeger (opentracing API implementation) */

  jaegertracing::Config config = jaegertracing::Config(
    disabled,
    jaegertracing::samplers::Config(
      CONST_SAMPLING_STRATEGY,
      always_sample),
    jaegertracing::reporters::Config(
      jaegertracing::reporters::Config::kDefaultQueueSize,
      jaegertracing::reporters::Config::defaultBufferFlushInterval(),
      log_span,
      ::jaegertracing::reporters::Config::kDefaultLocalAgentHostPort));

  std::shared_ptr<opentracing::Tracer> tracer = jaegertracing::Tracer::make(
    service_name, config, jaegertracing::logging::consoleLogger());

  /* Register jaeger tracer as global opentracing tracer,
     MAL will call opentracing API to obtain this tracer*/
  opentracing::Tracer::InitGlobal(
    std::static_pointer_cast<opentracing::Tracer>(tracer));
}


InitializeTracing("Some service");

/* Create a subscriber and a publisher in usual way and start publishing */

::elt::mal::Uri uri("zpb.ps://127.0.0.1:12404/test");
std::unique_ptr<ps::Subscriber<elt::SampleDataEntity>> subscriber =
  mal::CiiFactory::getInstance().getSubscriber<elt::SampleDataEntity>(
  uri, m_standardQoS, m_malProperties);

std::unique_ptr<ps::Publisher<elt::SampleDataEntity>> publisher =
  mal::CiiFactory::getInstance().getPublisher<elt::SampleDataEntity>(
  uri, m_standardQoS, m_malProperties);

std::shared_ptr<elt::SampleDataEntity> dataEntity = publisher->
↪createDataEntity();

dataEntity->setText("Initial");
dataEntity->setSampleId(0);

const std::size_t  SAMPLE_COUNT = 3;
for (std::size_t i = 0; i < SAMPLE_COUNT; i++) {
```

(continues on next page)

(continued from previous page)

```
  std::stringstream ss;
  ss << "Some text" << i;
  dataEntity->setText(ss.str());
  dataEntity->setSampleId(i);

 /* Since the global tracer is set (InitializeTracing) and
    elt::SampleDataEntity has struct attribute trace="true",
    MAL will start a new span and finish it immediately and propagate the span
    context using sample->setTrace()*/

  publisher->publish(*dataEntity, std::chrono::milliseconds(100));
}

std::this_thread::sleep_for(std::chrono::milliseconds(100));

std::vector<std::shared_ptr<elt::SampleDataEntity>> samples = subscriber->
↪read(0);

std::shared_ptr<elt::SampleDataEntity> sample = samples.at(0);

/* Obtain tracer using opentracing API */
std::shared_ptr<opentracing::Tracer> tracer = opentracing::Tracer::Global();

/* Obtain the span context from sample and create a child span */
std::stringstream contextStrm(sample->getTrace());

std::unique_ptr<::opentracing::SpanContext> spanContext =␣
↪::elt::mal::util::tracing::deserializeSpan(contextStrm);

{
  std::unique_ptr<opentracing::Span> childSpan = opentracing::Tracer::Global()->
↪StartSpan(
    "tracerPublishSubscribeTest",
    {opentracing::ChildOf(spanContext.get())});
}
```

Listing 8-6 Tracing mal pub/sub example (Java)

```
/**
 * @brief Initialize tracing (implementation specific to the user
 * environment, mal is using only opentracing API to obtain the tracer)
 *
 *  In this example we are using jaeger implementation of opentracing
 *  standard.
 *
 * @param serviceName service name
 */
public static void initializeTracing(String serviceName) {
```

(continues on next page)

(continued from previous page)

```
  /* Configuration specific to jaeger (opentracing API implementation) */
  Configuration.SamplerConfiguration samplerConfig =
  Configuration.SamplerConfiguration.fromEnv().withType("const").withParam(1);

  Configuration.ReporterConfiguration reporterConfig =
  Configuration.ReporterConfiguration.fromEnv().withLogSpans(true);

  Configuration config = new
  Configuration(serviceName).withSampler(samplerConfig).withReporter(
    reporterConfig);

  /* Register jaeger tracer as global opentracing tracer,
     MAL will call opentracing API to obtain this tracer */
  GlobalTracer.registerIfAbsent(config.getTracer());
}
initializeTracing("some service");

/* Create a subscriber and a publisher in usual way and start publishing */
URI uri = new URI("zpb.ps://127.0.0.1:12517/Sample");

try (Subscriber<SimpleSample> subscriber =
  CiiFactory.getInstance().getSubscriber(
    uri, QoS.DEFAULT, this.malProperties, SimpleSample.class);
  Publisher<SimpleSample> publisher =
    CiiFactory.getInstance().getPublisher(
      uri, QoS.DEFAULT, malProperties, SimpleSample.class);
) {

  SimpleSample dataEntity = publisher.createDataEntity();

  dataEntity.setValue(42.0);
  dataEntity.setDaqId(0);

  final int sampleCount = 3;
  for (int i = 0; i < sampleCount; i++) {
    dataEntity.setValue(i);
    dataEntity.setDaqId(i);

   /* Since the global tracer is set (initializeTracing) and
    SimpleSample has struct attribute trace="true",
    MAL will start a new span and finish it immediately and
    propagate the span context using sample.setTrace() */

    publisher.publish(dataEntity, 100, TimeUnit.MILLISECONDS);
  }
  /* Get published samples */
  ArrayList<SimpleSample> samples = new ArrayList<>();
  LocalDateTime now = LocalDateTime.now();
  LocalDateTime timeLimit = now.plusSeconds(10);
```

(continues on next page)

(continued from previous page)

```
while (timeLimit.isAfter(now)) {
  List<SimpleSample> readSamples = subscriber.read(0);
  samples.addAll(readSamples);

  if (samples.size() == sampleCount) {
    break;
  }

  now = LocalDateTime.now();
}

SimpleSample simpleSample = samples.get(0);

/* Obtain the span context from sample and create a child span */

Span span = TracingUtil.createSpan("SubscriberDemo",
   simpleSample.getTrace(), Tags.SPAN_KIND_CONSUMER);
```

Context Propagation

Tracing context is automatically propagated through MAL communication. Every request-reply action is instrumented. Upon completion or cancelation of the request, the Span.finish() method is called on the created span.

OLDB messaging automatically propagates the tracing context as well. If there is no active span at the time of call of OLDB API function, then no tracing context is transferred to the OLDB. Span.finish() method is called just before OLDB call completes but only if there was an active span. The OLDB client can obtain the span context from a data point value by calling a getTrace method which will return the context only if an active span exited at the time when write method was called. Otherwise, the context will be empty. It is then the responsibility of the client to create a child span from the context read from the data point value. For the OLDB subscriptions the user is also responsible for obtaining the span context from data point value received trough a subscription listener (NewValue call) and then creating new child spans.

In both cases, if there is an active span, the new span is created with the active span as a parent.

Any other type of tracing context propagation is the responsibility of the developer (see [22]).

9.7.3 Tracing Services

This section describes the services used by tracing. The services described below are used for collection, transmission, persistence, and querying of distributed CII traces data. They must be running and properly configured for the tracing system to work.

These services should be managed and maintained by system administrators. See the Log TRD document on how to set up the tracing services.

Documentation on configuration and deployment of these services can be found in [23].

Jaeger Agent

Network daemon that listens for spans sent over UDP, which it batches and sends to the collector.

Jaeger Collector

Receives traces from Jaeger Agents and runs them through a processing pipeline. It is responsible for persisting traces to the storage backend. It supports Elasticsearch as a storage backend.

Jaeger Query

This service retrieves traces from storage and hosts a UI to display them

9.7.4 Tracing GUI (Jaeger UI)

The Jaeger UI is a web application provided by the Jaeger Query service. It allows browsing the tracing data produced by the different CII services and applications.

By default, it's available on port 16686.

To access it, open to the http://ciijaegerqueryhost:16686 on your browser, where ciijaegerqueryhost is the hostname of the Jaeger Query service. The Jaeger UI search page will be presented to the user, as shown in Figure 8-6.

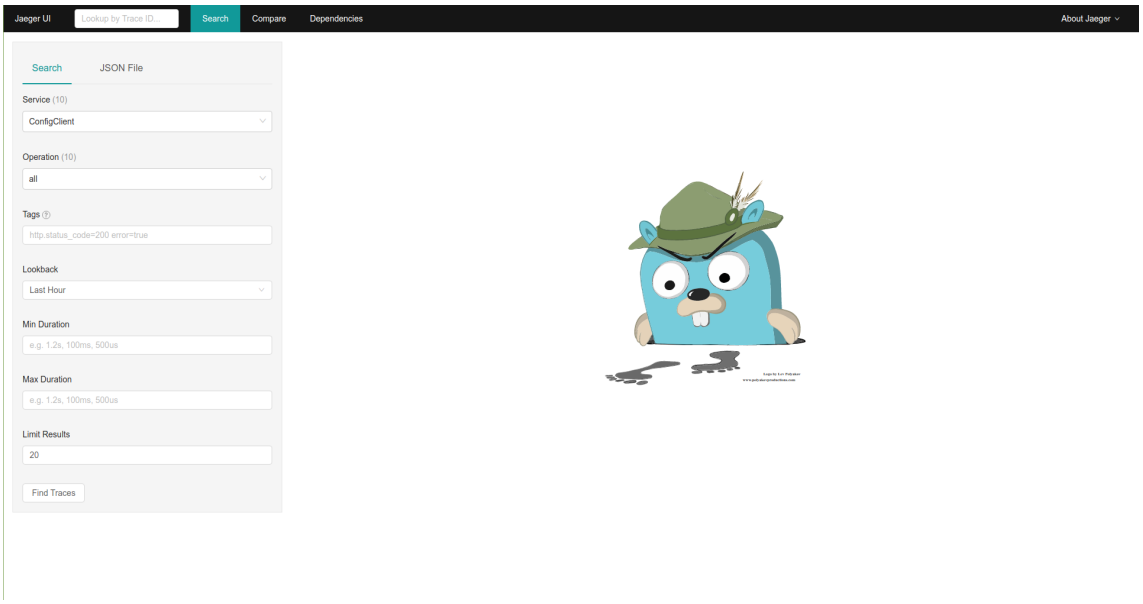


Figure 8-6 Jaeger UI search view

On the left side of this view, the different filtering parameters for the traces search can be defined by the user. To perform the search, click the Find Traces button. When this button is clicked, a list of traces matching the search filtering options will be displayed on the right side of the screen, as shown in Figure 8-7.

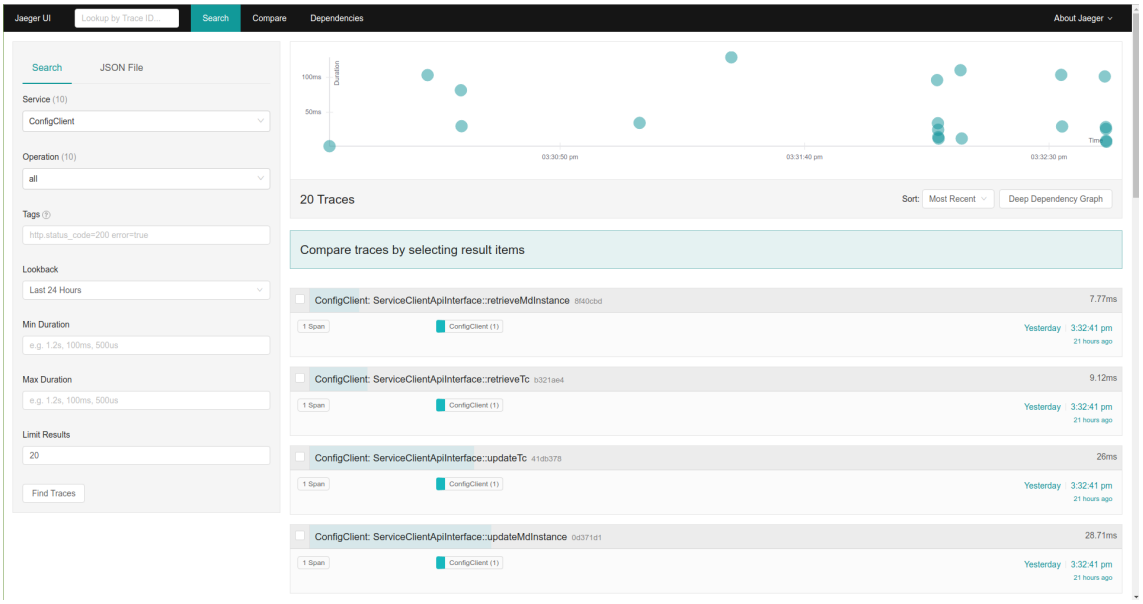


Figure 8-7 Jaeger UI traces search result

It is possible to look for a specific trace using the Lookup by Trace ID input field on the top-left side of the screen. It is also possible to browse a trace directly with the URL

https://ciijaegerqueryhost:16686/trace/<trace_id>[1].

When clicking on a trace on the search results list, or accessing it directly, the trace view will be presented to the user in a trace timeline view, as shown in Figure 8-8. Clicking on each of the spans will display the information about that span.

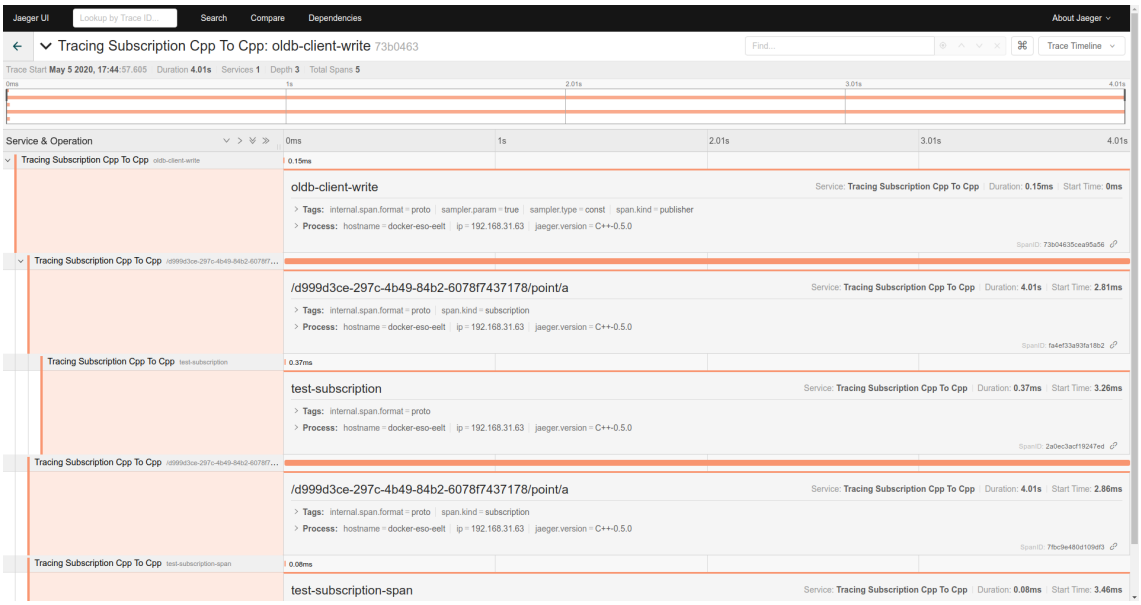


Figure 8-8 Jaeger UI trace view

The trace view offers the following trace visualization modes:

- **Trace Timeline**: Displays the trace and its spans in a timeline visualization (Figure 8-8).
- **Trace Graph**: Displays the trace and its spans as a directed acyclic graph (Figure 8-9).
- **Trace JSON**: Displays the trace and its spans as a JSON object.
- **Trace JSON (Unadjusted)**: Displays the trace and its spans as a JSON object with raw data without any adjustments [25].

[1] https://ciijaegerqueryhost:16686/trace/%3ctrace_id%3e

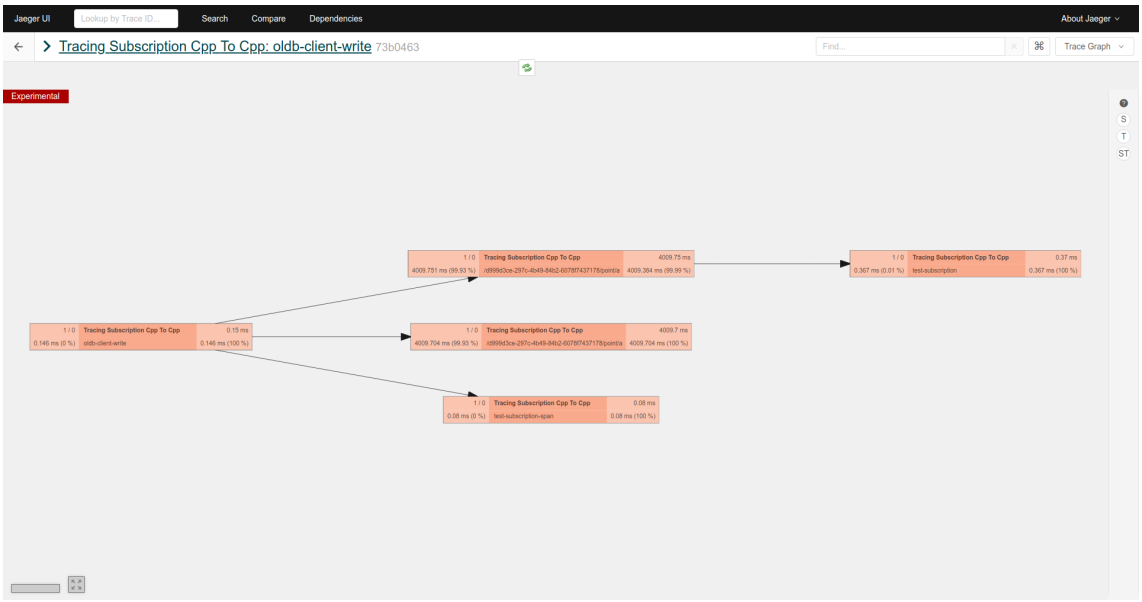


Figure 8-9 Jaeger UI Trace Graph visualization

9.8 CII Log Client API

This are the listings for the CII Log API methods with short descriptions. APIs for Java, C++, and Python are slightly different, therefore a listing for each of the three languages is provided.

9.8.1 Java

Java CII Log API is provided by the CiiLogManager, CiiLogMessageBuilder , CiiLayout, and CiiSimpleLayout.

CiiLogManager

public static Logger getLogger()
CII logger factory method. **Returns**: • A formatter logger object with the name of the calling class. It automatically initializes and configures the logging context if it has not been initialized previously.

public static Logger getLogger(String loggerName)
CII logger factory method. **Returns**: • A formatter logger object with the provided name. It automatically initializes and configures the logging context if it has not been initialized previously.

public static void addAppender(Appender appender, Level level, Filter filter)
This method adds an appender with the provided level and filter to the root logger. **Parameters**: • appender The appender to be added. • level The level threshold for the appender. • filter The filter to be applied to log messages by the appender.

public static void isPostmortemBufferEnabled()
Checks if the postmortem buffer is enabled. **Returns:** • true if the post moretem buffer is enabled, false otherwise.

public static boolean flushPostmortemBuffer()
Flushes buffered logs to appenders that have previously rejected those log messages. **Returns:** • A boolean value indicating if the flushing of the buffer was actually performed. For example, if the method is called when the buffer is disabled, it will return false.

public static void shutdown()
Shuts down current logging context.

public static long elt.log.CiiLogManager.getPostmortemBufferTtlMillis()
Gets the value of the postmortem Time-To-Live (TTL) for buffered log events. **Returns**: • TTL of the postmortem buffer in ms.

CiiLogMessageBuilder

public static CiiLogMessageBuilder create(String message)
Factory method to create a CiiLogMessageBuilder instance with the provided Message field value. **Parameters:** • message The message value. **Returns**: • The message builder.

public static CiiLogMessageBuilder create()
Factory method to create a CiiLogMessageBuilder instance. **Returns**: • The message builder.

public CiiLogMessageBuilder withMessage(String message)
Sets the Message field value. **Parameters:** • message The message value. **Returns**: • The message builder.

public CiiLogMessageBuilder withUserLogId(String userLogId)
Sets the UserLogID field value. **Parameters:** • userLogId The user ID value. **Returns**: • The message builder.

public CiiLogMessageBuilder withAudience(CiiLogAudience audience)
Sets the Audience field value. **Parameters:** • audience The audience value. **Returns**: • The message builder.

public CiiLogMessageBuilder withTracingInfo()
Indicates the CiiLogMessageBuilder to include TraceID and SpanID fields from the current tracing context. **Returns**: • The message builder.

public CiiLogMessageBuilder withTracingInfo(Boolean withTracingInfo)
Indicates the CiiLogMessageBuilder if it should include TraceID and SpanID fields from the current tracing context. **Parameters:** • withTracingInfo Boolean indicating if tracing info should be added to the built message. **Returns**: • The message builder.

public CiiLogMessageBuilder withCiiException()
Sets a CiiException to be formatted to the ErrorTypeName, ErrorDateTime, ErrorMessage, and ErrorTrace and added to the log message. **Returns**: The message builder.

public String build()
Constructs the log message string according to the defined parameters. **Returns**: • The formatted string.

public static String createAndBuildWithException(String message, CiiException exception)
Convenience method that creates a formatted log message with the the provided Message field value and the fields containing the CiiException info. **Parameters:** • message The message value. • Exception The exception to be formatted. **Returns**: • The formatted string.

public static String createAndBuildWithAudience(String message, CiiLogAudience audience)
Convenience method that creates a formatted log message with the the provided Message field value and the provided audience. **Parameters:** • message The message value. • audience The audience value. **Returns**: • The formatted string.

CiiLayout

public static CiiLayout create()
Factory method to create a CiiLayout instance. **Returns:** • A new CiiLayout instance.

CiiSimpleLayout

public static CiiSimpleLayout create()
Factory method to create a CiiSimpleLayout instance. **Returns:** • A new CiiLayout instance.

9.8.2 C++

C++ CII Log Client API is provided by CiiLogManager, CiiLogMessageBuilder, CiiLogConfigurator, CiiLayout, and CiiSimpleLayout classes.

CiiLogManager

void elt::log::CiiLogManager::configure (const std::map< std::string, std::string > & properties, cpp_logging=False)
Configure logging from properties. **Parameters:** • properties Logging configuration written as properties (name and value pairs) • cpp_logging Forward the configuration to the C++ CII Log Manager

void elt::log::CiiLogManager::configure (const std::string & logConfigFile, cpp_logging=False)
Configure logging from a file name. **Parameters:** • logConfigFile Logging configuration file • cpp_logging Forward the configuration to the C++ CII Log Manager

std::string elt::log::CiiLogManager::detailMessage(const std::string & message, const CiiLogAudience audience, const std::string userLogId = “”)
Utility method producing a detailed message with all method’s parameters included. **Parameters:** • message message part of detailed message • audience audience part of detailed message • userLogId userLogId part of detailed message

log4cplus::Logger elt::log::CiiLogManager::getLogger (const std::string loggerName)
Gets a logger instance. **Parameters:** • loggerName logger name

CiiLogMessageBuilder

Provides a way to construct a log message according to the CII log format

elt::log::CiiLogMessageBuilder::CiiLogMessageBuilder ( )
Default constructor.

elt::log::CiiLogMessageBuilder::CiiLogMessageBuilder (const std::string & message)
Constructor, taking message as argument **Parameters**: • message Message value

std::string elt::log::CiiLogMessageBuilder::Build ( ) const
Build formatted log message string from information within builder and return it. **Returns**: • The formatted log message string

static CiiLogMessageBuilder elt::log::CiiLogMessageBuilder::Create ( )
Static method to create CiiLogMessageBuilder (Java API compat) **Returns**: • A new instance of the CiiLogMessageBuilder

static CiiLogMessageBuilder elt::log::CiiLogMessageBuilder::Create (const std::string & message)
Static method to create CiiLogMessageBuilder with message (Java API compat) **Parameters:** • message Message value **Returns:** • A new instance of the CiiLogMessageBuilder

static std::string elt::log::CiiLogMessageBuilder::CreateAndBuildWithAudience (const std::string & message, CiiLogAudience audience)

Static method to create a formatted log message string with provided message and audience

Parameters:

- message Message value
- audience Target log audience

Returns:

- The formatted message string

static std::string elt::log::CiiLogMessageBuilder::CreateAndBuildWithException (const std::string & message, const elt::error::CiiException & exception)

Static method to create a formatted log message string with provided message and CiiException

Parameters:

- message Message value
- exception Reference to the instance of the elt::error::CiiException

Returns:

- The formatted message string

static std::string elt::log::CiiLogMessageBuilder::CreateAndBuildWithMessage (const std::string & message)

Static method to create a formatted log message string with provided message

Parameters:

- message Message value

Returns:

- The formatted message string

CiiLogMessageBuilder& elt::log::CiiLogMessageBuilder::WithAudience (CiiLogAudience audience)

Add audience to the builder and return reference to self (chaining support)

Parameters

- message audience CiiLogAudience value

Returns

- A reference to self

CiiLogMessageBuilder& elt::log::CiiLogMessageBuilder::WithException (const elt::error::CiiException & exception)

Add information about an elt::error::CiiException to the builder and return reference to self (chaining support).

Parameters:

- exception Reference to the exception

Returns:

- A reference to self

CiiLogMessageBuilder& elt::log::CiiLogMessageBuilder::WithException (const std::exception & exception)

Add information about std::exception to the builder and return reference to self (chaining support)

Parameters:

- exception Reference to the exception

Returns:

- A reference to self

CiiLogMessageBuilder& elt::log::CiiLogMessageBuilder::WithMessage (const std::string & message)

Add message to the builder and return reference to self (chaining support)

Parameters:

- message message value

Returns:

- A reference to self

CiiLogMessageBuilder& elt::log::CiiLogMessageBuilder::WithTracingInfo (bool tracing_info)

Enable/disable tracing info flag and return reference to self (chaining support)

Parameters:

- tracing_info boolean flag to enable/disable addition of tracing information into the message

Returns:

- A reference to self

CiiLogMessageBuilder& elt::log::CiiLogMessageBuilder::WithUserLogId (const std::string & user_log_id)
Add user log ID string to the builder and return reference to self (chaining support) **Parameters:** • user_log_id user defined log id string **Returns**: • A reference to self

CiiLogConfigurator

Configures Log4cplus loggers and appenders from configuration file and injects each configured logger with one PostmortemAppender instance to keep all filtered events for some specified time for later usage.

Configure loggers with configuration from log4cplus.properties file and life time of 1 second with example:

```
//configure log4cplus from log4cplus.properties file and then set the
//life time of the stored events for 1 second
CiiLogConfigurator::doConfigure ( 1 , "log4cplus.properties" );
```

If configuration of Log4cplus is initialized without the postmortem buffer enabled, it is possible to enable it once the Log4cplus configuration has been initialized with the following method:

```
//set the life time of the stored events for 1 second
CiiLogConfigurator::inject( 1 );
```

To dump stored events which still have valid lifetimes call this method:

```
//dump all valid events to parent logger
CiiLogConfigurator::postmortemDump();
```

CiiLogConfigurator has an option to reset log4cplus configuration be fully reset when calling doConfigure:

```
CiiLogConfigurator::setResetOnConfigure( shouldReset );
```

static void elt::log::CiiLogConfigurator::doConfigure(uint32_t life_time, const log4cplus::tstring & config_filename, log4cplus::Hierarchy & h = log4cplus::Logger::getDefaultHierarchy(), unsigned flags = 0)

This method eliminates the need to create a temporary PropertyConfigurator to configure log4cplus. It is equivalent to the following:

```
PropertyConfigurator config("filename");
config.configure();
```

Parameters:

- life_time The life time in seconds for an event to be stored internally in the Postmortem appender
- config_filename The configuration file name
- h root hierarchy
- flags flags

static void elt::log::CiiLogConfigurator::inject (uint32_t life_time)

Inject PostMortemAppender into each configured logger instance without loading property file.

Parameters:

- life_time The life time in seconds for an event to be stored internally in the Postmortem appender

static void elt::log::CiiLogConfigurator::postmortemDump ( )

Dump all valid events (events which have life time less than globally specified life time) to all the loggers that previously rejected them.

static void elt::log::CiiLogConfigurator::setResetOnConfigure (bool reset)

Decides whether the configuration should reset all already configured loggers when a new configuration is applied.

Parameters:

- reset true = enable, false = disable

static bool elt::log::CiiLogConfigurator::getResetOnConfigure ( )

Returns:

- Boolean indicating if the existing loggers will be reset when a configuration is applied.

CiiLayout

Log layout for CII Log format.

elt::log::layout::CiiLayout::CiiLayout(const log4cplus::helpers::Properties & properties)
Construct CII layout with properties. **Parameters:** • properties Layout properties.

CiiSimpleLayout

Simple log layout for CII Log format.

elt::log::layout::CiiSimpleLayout::CiiSimpleLayout(const log4cplus::helpers::Properties & properties)
Construct CII simple layout with properties. **Parameters:** • properties Layout properties.

9.8.3 Python

Python CII Log Client API is provided by CiiLogManager, CiiLogConfigurator, and CiiLogMessageBuilder classes.

CiiLogManager

Provides methods to manage and configure logging, as well as methods to obtain logger objects.

log.CiiLogManager.configure(config = None)

Perform logging configuration from provided dictionary/file just like logging.config.dictConfig() function with additional functionality:

- Initialize missing keys in the provided dictionary from CII_DEFAULT_CONFIG, so that the root logger gets initialized with logsink and console handlers.
- Configuration dictionary supports additional top level key cii_log_buffer, which must contain a dictionary with keys enabled and ttl_ms. The value of enabled key must be boolean. When True, a log buffer will be enabled. The value of ttl_ms key must be integer defining captured record Time-To-Live in ms (within the buffer). Note that these settings will be overridden with the values of the environment variables below.
- Environment variables CII_LOG_BUFFER_ON and CII_LOG_BUFFER_TTL_MS are checked for existence. These environment variables override settings for cii_log_buffer provided in the configuration dictionary.
  - If CII_LOG_BUFFER_ON exists AND has value 1 or TRUE (case insensitive), the log buffer will be enabled. The value of 0 or FALSE will disable the log buffer.
  - Environment variable CII_LOG_BUFFER_TTL_MS if exists, must be convertible to int and represents the number of milliseconds that the log buffer retains the messages. When this variable does not exist, the default of 60000ms is assumed.

Parameters:

- config Configuration dictionary or filename or None. When a filename is provided, it must be a valid JSON file that can be parsed into the config dictionary. When None is provided, the configuration dictionary is used from the default CII_DEFAULT_CONFIG dictionary.

Returns:

- Dictionary with the actual configuration used.

log.CiiLogManager.get_logger(name = None)

CII logger factory method.

Parameters:

- name Name of the logger or None to obtain root logger.

Returns:

- Instance of logger object.

log.CiiLogManager.flush_postmortem_buffer()

Flush buffered logs to the handlers that have previosly rejected thos log messages. Cyclic log buffer is reset.

Returns:

- True if flushing was actually performed, False otherwise.

log.CiiLogManager.is_postmortem_buffer_enabled()
Query whether buffered logger is installed. **Returns**: • True when that is the case, False otherwise.

CiiLogConfigurator

log.CiiLogConfigurator.load_configuration(filename)
Load log configuration from provided file. Log configuration must be saved in JSON format. Note that this is not supported with Python logging subsystem. Parsing of log configuration must produce a dictionary that is compatible with CiiLogManager.configure() method **Parameters**: • filename Name of JSON file containing configuration. **Returns**: • Dictionary with log configuration.

log.CiiLogConfigurator.configure(config = None)
Perform logging configuration from provided dictionary/file just like logging.config.dictConfig() function with additional functionality: • Initialize missing keys in the provided dictionary from CII_DEFAULT_CONFIG, so that the root logger gets initialized with logsink and console handlers. • Configuration dictionary supports additional top level key cii_log_buffer, which must contain a dictionary with keys enabled and ttl_ms. The value of enabled key must be boolean. When True, a log buffer will be enabled. The value of ttl_ms key must be integer defining captured record Time-To-Live in ms (within the buffer). Note that these settings will be overridden with the values of the environment variables below. • Environment variables CII_LOG_BUFFER_ON and CII_LOG_BUFFER_TTL_MS are checked for existence. These environment variables override settings for cii_log_buffer provided in the configuration dictionary. – If CII_LOG_BUFFER_ON exists AND has value 1 or TRUE (case insensitive), the log buffer will be enabled. The value of 0 or FALSE will disable the log buffer. – Environment variable CII_LOG_BUFFER_TTL_MS if exists, must be convertible to int and represents the number of milliseconds that the log buffer retains the messages. When this variable does not exist, the default of 60000ms is assumed. **Parameters**: • config Configuration dictionary or filename or None. When a filename is provided, it must be a valid JSON file that can be parsed into the config dictionary. When None is provided, the configuration dictionary is used from the default CII_DEFAULT_CONFIG dictionary. **Returns**: • Dictionary with the actual configuration used.

log.CiiLogConfigurator.check_config( )
Check whether configure() method was already called. If that was not the case, it calls it to configure defaults.

log.CiiLogConfigurator.save_configuration(config, filename)
Save log configuration dictionary as json into provided filename. **Parameters**: • config Dictionary with log configuration. • filename Name of the output file.

log.CiiLogConfigurator.get_formatter(simple = False)

Create and return CII specific formatter

Parameters:

- simple When set to True, return formatter with simple format, otherwise return formatter with CII full pattern.

Returns:

- Instance of formatter object

log.CiiLogConfigurator.flush_postmortem_buffer()

Flush buffered logs to the handlers that have previosly rejected thos log messages. Cyclic log buffer is reset.

Returns:

- True if flushing was actually performed, False otherwise.

log.CiiLogConfigurator.flush_postmortem_buffer_to_string()

Flush content of the cyclic log buffer to string. Cyclic log buffer is reset.

Returns:

- String containing log messages captured by CiiBufferedLogger, or a RuntimeError when CiiBufferedLogger is not installed.

log.CiiLogConfigurator.reset_postmortem_buffer()

Empty the log buffer if buffered logger is installed.

Returns:

- True when log buffer was reset, False when not installed.

log.CiiLogConfigurator.get_effective_logsink_filename()

Return full name of the filename for log sink (if one was created by calling get_logsink_filename).

Returns:

- The filename of the logsink log file or None when get_logsink_filename was not called.

log.CiiLogConfigurator.get_logsink_filename()

Return full name of the filename for log sink.

Returns:

- The filename of the logsink log file.

log.CiiLogConfigurator.enable_postmortem_buffer(ttl_seconds = 60)
Enable postmortem buffer by installing a buffered logger as a root logger. **Parameters**: • ttl_seconds Message Time-To-Live in seconds. **Returns**: • True when logger installed with this invocation, False when already installed. Note ttl_seconds parameter has no meaning when BufferedLogger is already installed.

log.CiiLogConfigurator.is_postmortem_buffer_enabled()
Query whether buffered logger is installed. **Returns**: • True when that is the case, False otherwise.

CiiLogMessageBuilder

Provides a way to construct a log message according to the CII log format.

__init__(self, message=None)
Constructor **Parameters**: • message The message field value.

create(cls, message=None)
Creates a formatted log message string with exception info. **Parameters:** • message The message field value. **Returns:** • A new instance of the CiiLogMessageBuilder.

create_and_build_with_exception(cls, message, exception)
Creates a formatted log message string with exception info. **Parameters:** • message The message field value. • exception The exception instance to be formatted to the appropriate fields the log message. Exception instance should be derived from the elt.config.CiiException. **Returns:** • The formatted string.

create_and_build_with_audience(cls, message, audience)
Creates a formatted log message string with audience field. **Parameters:** • message The message field value. • audience The CiiLogAudience value to be formatted to the Audience field. **Returns:** • The formatted string.

create_and_build_with_args(cls, **kwargs)
Create a formatted log message string from provided arguments. See with_args. The following arguments are supported: • message: value of the message field (string). • audience: value of the audience field (CiiLogAudience). • user_log_id: value of the user log ID field. • exception: exception to be formatted. • with_tracing_info: boolean value for enabling/disabling of tracing info. **Parameters:** • kwargs keyword arguments. **Returns:** • The formatted string.

with_message(self, message)
Sets the mesage field value. **Parameters:** • message The message field value. **Returns:** • The builder instance itself.

with_user_log_id(self, user_log_id)

Sets the user ID field value.

Parameters:

- user_log_id Value for the user log ID field.

Returns:

- The builder instance itself.

with_audience(self, audience)

Sets the audience for the log message.

Parameters:

- audience The intended CiiLogAudience of the log message.

Returns:

- The builder instance itself.

with_tracing_info(self, tracing_info=True)

Sets the log message to include/exclude tracing info fields.

Returns:

- The builder instance itself.

with_cii_exception(self, exception)

Sets the exception to be formatted and included in the log message.

Parameters:

- exception Exception to be formatted and added to the log message.

Returns:

- The builder instance itself.

with_exception(self, exception)

Sets the exception to be formatted and included in the log message. Alias for with_cii_exception.

Parameters:

- exception Exception to be formatted and added to the log message.

Returns:

- The builder instance itself.

with_args(self, message=None, audience=None, user_log_id=None, exception=None, with_tracing_info=None)
Sets the fields from provided keyword arguments. The following arguments are supported: • message: value of the message field (string). • audience: value of the audience field (CiiLogAudience). • user_log_id: value of the user log ID field. • exception: exception to be formatted. • with_tracing_info: boolean value for enabling/disabling of tracing info. **Parameters:** • kwargs keyword arguments. **Returns:** • The builder instance itself

build(self)
Construct the log message string according to the defined parameters. **Returns:** • The constructed log message following the Cii log format.

9.9 CII Log Fields

All fields in a log message, aside from the date/time fields are treated as strings.

Table 8-3 Log fields

Field	ES field name	Description	Optional
Cref	cii.cref	Source filename/line number.	NO
Date	cii.date	Date and time of log creation down to milliseconds in UTC timescale with the 'T' designator between date and time, eg: 2020-03-16T09:32:09.681+0100. It follows ISO 8601 standard.	NO
HostName	cii.hostname	Hostname of the computing node where the log was produced.	NO
LogType	cii.logtype	The type of log message (level).	NO
SourceID	cii.sourceid	Class name, thread name.	NO
LogID	ci.logid	ID of the log message source location.	NO
Message	cii.message	Text message of the log event.	NO
Audience	cii.audience	The intended recipient of the log.	YES
UserLogID	cii.userlogid	A user provided ID.	YES
TraceID	cii.traceid	ID of the trace for the current tracing context.	YES
SpanID	cii.spanid	ID of the span for the current tracing context.	YES
ErrorTypeName	cii.error.typename	Name of the CII Error	YES
ErrorDateTime	cii.error.datetime	Unix timestamp of the error	YES
ErrorMessage	cii.error.message	Message of the error	YES
ErrorTrace	cii.error.trace	Stack trace of the error	YES

CHAPTER

TEN

ALARM

Document ID:	
Revision:	2.1
Last modification:	March 18, 2024
Status:	Released
Repository:	https://gitlab.eso.org/cii/info/cii-docs
File:	alarm.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
1.0	6.5.2020	jstrnisa/ bterpinc	All	Document creation.
1.1	5.7.2020	bterpinc	4	Added plugin tab example
1.2	15.10.2020	mseko-ranja	All	RIX updates
1.3	17.11.2020	mseko-ranja	4.3.2.	Replaced ias-webserver sender with iasWebServer Sender
1.4	13.1.2021	jtepinc	4.2	Alarm Configuration GUI updates
1.5	26.1.2021	jtepinc, mseko-ranja	4.2	Alarm Configuration GUI updates (RIXes PSI 19, PSI 21). Transfer function alarm priority paragraph added (PSI 39).
2.0	14.9.2022	mschilli	all	CII v3: new tooling and workflow for defining and monitoring alarms
2.1	18.03.2024	mschilli	0	CII v4: Public doc

Confidentiality

This document is classified as Public.

Scope

This document is a manual for the Alarm system of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

Glossary of Terms

API	Application Programming Interface
CII	Core Integration Infrastructure
CLI	Command Line Interface
CDB	IAS configuration database
DP	Data Point
GUI	Graphical User Interface
ES	ElasticSearch
JSON	Javascript object notation
OLDB	Online Database
URI	Uniform Resource Identifier
SVN	Subversion
IAS	Integrated Alarm System
DASU	Distributed Alarm System Unit
ASCE	Alarm System Computing Element
IASIO	Integrated Alarm System Input/Output

References

1. ESO, Core Integration Infrastructure Requirements Specification, ESO-192922 Version 7
2. ESO, Integrated Alarm System Architecture, ESO-293482 Version 2
3. ESO, Integrated Alarm System Design, ESO-299387 Version 1
4. ESO, Integrated Alarm System General information, https://github.com/IntegratedAlarmSystem-Group/ias/wiki
5. https://www.eso.org/~eltmgr/CII/latest/manuals/html/docs/services.html
6. https://www.eso.org/~eltmgr/CII/latest/manuals/html/docs/oldb.html
8. https://github.com/IntegratedAlarmSystem-Group/ias/wiki/Transfer-functions-how-to
9. https://github.com/IntegratedAlarmSystem-Group/ias/wiki/Transfer-function-in-python
10. https://github.com/IntegratedAlarmSystem-Group/integration-tools/blob/develop/docs/DEPLOYMENT_GUIDE.md

11. https://github.com/IntegratedAlarmSystem-Group/integration-tools/blob/develop/docs/WEB_APPS_DOCUMENTATION.md
12. https://github.com/IntegratedAlarmSystem-Group/ias/wiki/ConfigurationDatabase

10.1 Overview

This document is a user manual for CII Alarm service. It explains how to set-up, start and stop the alarm system (chapter 3), how to configure alarms (chapter 4), and how to inspect and address alarms (chapter 4), including concrete working examples.

The advanced topics (chapter 5) are mostly only relevant for administrators.

10.2 Introduction

The CII Alarm System is based on the Integrated Alarm System IAS.

It operates on rules that observe the OLDB and decide whether an alarm should be raised. The alarm system is therefore fully configuration-based. An application never directly triggers an alarm. An alarm is represented by an OLDB datapoint as well. These datapoints are owned by the Alarm system, and you cannot write to them. You read these datapoints to find out which alarms are active or not.

IAS is a software whose main purpose is to get alarms and monitor points from different sources and generate alarms to present to the users who can range from operators in the control room up to engineers at their desks. The alarm system is a message-passing facility that routes information about abnormal situations detected from hardware or software to the user. For IAS architecture and design, please refer to the IAS architecture [2] (necessary to read to get to know IAS concepts) and design [3] documents.

IAS is based on pluggable architecture to integrate with external systems. To connect the IAS to the CII, the CII-IAS bridge was developed. The bridge consists of the following components:

- OLDB Plugin - The long-running process that reads from OLDB and writes (via Kafka) to IAS.
- OLDB Sink - The long-running process that reads (via Kafka) from IAS and writes to OLDB.

Changes in CII v3:

The following components were retired

- CII-OLDB Adaptor
- CII-IAS Config Exporter
- Alarm GUI webserver
- WebServer Sender
- Alarm Config GUI

The following features are *not available* in this version of the CII Alarm System

- Acknowledgement
- Shelving

10.3 Prerequisites

This section describes the prerequisites for using the CII Alarm service.

10.3.1 Set-up

An administrator (with root privileges) has to post-install the Alarm system on the host. The procedure is described in #Advanced-Topics/Installation below.

10.3.2 Start-Stop

Start the Alarm system service(s)

```
sudo cii-services start alarm
```

Stop the Alarm system service(s)

```
sudo cii-services stop alarm
```

10.4 Usage

The CII Alarm System operates on rules that observe the OLDB and decide whether an alarm should be raised. An application never directly triggers an alarm. An alarm is represented by an OLDB datapoint as well. These datapoints are owned by the Alarm system, and you cannot write to them. You read these datapoints to find out which alarms are active or not.

The general workflow is this:

I. You define alarm rules

- Step 1: You write an alarm definition file
- Step 2: You pass the definitions to the alarm system
- Step 3: You request the alarm system to read the new definitions

II. You monitor and trigger alarms

- Step 4: You monitor the alarm-datapoints
- Step 5: You write to the input-datapoints

10.4.1 Write Alarm Definitions

Location Alarm rules and CDB are stored in */etc/cii_ias/*

Create/Edit this file:

```
/etc/cii_ias/alarm-rules.yaml
```

Example

```
---
alarmId: /alarm/harmoni/sensor1_alarm
shortDesc: Demo alarm for numerical value (lowOn, highOn)
rule: [ minmax, /harmoni/fcs1/sensor1/value, 30, 50 ]
---
alarmId: /alarm/harmoni/errorstates/tracking_alarm
shortDesc: Demo alarm with a boolean input value
rule: [ bool, /harmoni/fcs1/adc1/trackingfailure ]
---
alarmId: /alarm/harmoni/hitemp/dcs_alarm
shortDesc: Detector High Temperature (highOff, highOn)
rule: [ max, /harmoni/dcs1/detector/temperature1, 70.0, 100.0 ]
---
alarmId: /alarm/harmoni/device/error_alarm
shortDesc: Demo alarm with string input
rule: [ regexp, /harmoni/dcs1/device/error_string, Error.* ]
```

Format

In the alarm definitions file, each rule consists of 4 lines:

```
---             #1 three minus signs (required)
alarmId:        #2 rule name. must begin with "/alarm/"
shortDesc:      #3 free text describing the alarm
rule:           #4 the alarm function name and its arguments, in list format
```

Alarm functions

These are the available functions for use in the alarm definitions.

Hints

- Arguments called “dp” are addresses of OLDB datapoints. Give the absolute path inside the oldb, but without the “cii.oldb://” prefix, i.e. the address shall start with a single slash.
- The DP’s oldb-type must be compatible with the function-arg type. E.g., where an alarm function expects a double, avoid passing a String-DP (OldbString), but you can pass an Int-DP (e.g. OldbInt64Std).

```
* min (double dp, double lowOn)
  Alarm activates while dp value is below lowOn

* min (double dp, double lowOn, double lowOff)
  Alarm activates when dp value goes below lowOn, deactivates above lowOff.

* max (double dp, double highOn)
  Alarm activates while dp value is above highOn

* max (double dp, double highOff, double highOn)
  Alarm activates when dp value goes above highOn, deactivates below highOff.

* minmax (double dp, double lowOn, double highOn)
  Alarm activates while dp value is below lowOn or above highOn
  In other words, this is minmax without hysteresis.

* minmax (double dp, double lowOn, double lowOff, double highOff, double highOn)
  Alarm activates when dp value goes below lowOn, deactivates above lowOff.
  Analogously for highOn/Off.
  In other words, this is minmax with hysteresis.

* bool (bool dp)
  Alarm activates while dp value is "true"

* regexp (string dp, string regexp)
  Alarm activates while dp value matches the regexp
```

10.4.2 Inject Definitions

The ciiAlarmLoader tool reads the alarm definition file, and creates alarm config as well as other config in the format understood by IAS: supervisor config, converter config, etc. The IAS keeps this information in the IAS configuration database (CDB).

```
ciiAlarmLoader  alarm-rules.yaml  .  --check-inputs
```

The alarm loader will output a message if there are alarm rules for which the input-DPs (datapoint-URIs given as arguments to the alarm functions) are currently not existing in the OLDB. These alarm rules will not be able to work, and the affected alarms will never trigger, until the input-datapoints get created by someone. In real scenarios, the input-datapoints get created by the ECS oldb-loader tool, or by control applications.

Alarm rules for which the input-DPs are existing already at the time of reloading the CDB (see next step) will function instantly. For those with missing inputs, there will be a certain delay after the missing input-DPs were created. The alarm system will discover the new inputs within 60 seconds.

Example

To create the input-DPs for the example rules file above, you can use this shell command:

```
/usr/bin/env python - <<EOF

import elt.oldb, elt.config
elt.oldb.CiiOldbGlobal.set_write_enabled(True)
oldb = elt.oldb.CiiOldbFactory.get_instance()

for k,v in {
  "/harmoni/fcs1/sensor1/value" : 34,
  "/harmoni/fcs1/adc1/trackingfailure": False,
  "/harmoni/dcs1/detector/temperature1": 56.0,
  "/harmoni/dcs1/device/error_string": "None"
}.items():

try:
  oldb.create_data_point_by_value (elt.config.Uri("cii.oldb://"+k) , v)
except:
  pass

EOF
```

10.4.3 Request a Reload

Assuming the alarm system is already running, request it to re-read its config.

```
ciiAlarmCtl daemon reload
```

The command will write a warning if the alarm system is not running, to inform you that no-one will handle your request. If so, see the #Prerequisites section on how to start it.

With this, the definition of alarms is completed, and you can start using them.

10.4.4 Monitor Alarms

Since the alarms are OLDB datapoints, you can monitor them with the OLDB API, OLDB GUI, or OLDB tools (all described in [6]), for example:

```
oldb-cli read  </absolute/path/of/alarm-dp>  2>/dev/null | tail -n1

oldb-cli subscribe  </absolute/path/of/alarm-dp>
```

Moreover, the Alarm System comes with 2 dedicated tools:

1. ciiAlarmMon (command line executable)

```
ciiAlarmMon /etc/cii_ias
```

2. Alarm Monitor service (system service)

```
http://<host_where_ias_runs>:5602/
```

The Alarm Monitor service also records all alarm state changes in a (rolling) file

```
/var/tmp/elt/cii-alarm-mon.log
```

10.4.5 Trigger Alarms

Since the inputs are OLDB datapoints, you can read/write them with the OLDB API, OLDB GUI, or OLDB tools (all described in [6]), for example:

```
oldb-cli read  </absolute/path/of/input-dp>   2>/dev/null | tail -n1

oldb-cli write  </absolute/path/of/input-dp>  <value>   2>/dev/null | tail -n1
```

Example 1

This examples uses a built-in demo alarm, so will work on every installation

```
oldb-cli read /alarm/sandbox/cii_diag/basic_alarm  2>/dev/null | tail -n1
Value:  NOMINAL

oldb-cli read /sandbox/cii_diag/alarm_source_bool  2>/dev/null | tail -n1
Value:  false

oldb-cli write /sandbox/cii_diag/alarm_source_bool true 2>/dev/null | tail -n1
Value:  true

oldb-cli read /alarm/sandbox/cii_diag/basic_alarm  2>/dev/null | tail -n1
Value:  ALARM_PRIORITY2

oldb-cli write /sandbox/cii_diag/alarm_source_bool false 2>/dev/null | tail -n1
Value:  false
```

Example 2

This example uses the alarms demonstrated earlier, so it can only work if you have followed the above examples.

In terminal A:

```
ciiAlarmMon /etc/cii_ias
```

In terminal B:

```
# Current temperature
oldb-cli read /harmoni/dcs1/detector/temperature1   2>/dev/null | tail -n1
Value:  56.0

# Increase to 90 - no alarm
oldb-cli write /harmoni/dcs1/detector/temperature1 90   2>/dev/null | tail -n1

# Increase temperature above limit - alarm state will change
oldb-cli write /harmoni/dcs1/detector/temperature1 120.0   2>/dev/null | tail -n1

# Decrease to 90 - alarm still on
oldb-cli write /harmoni/dcs1/detector/temperature1 90.0   2>/dev/null | tail -n1

# Decrease to normal - alarm state will change
oldb-cli write /harmoni/dcs1/detector/temperature1 56   2>/dev/null | tail -n1
```

10.5 Advanced Topics

10.5.1 Administration

Installation

The CII-IAS-Bridge and and Alarm tools are included in the ELT DevEnv distribution, but need to be post-installed before they can be used. The post-install routine will also install additional third-party libraries that are not included in the ELT DevEnv (these libraries are only needed on hosts where the CII Alarm System is running).

To set-up the alarm system on a host, do (as root):

```
/elt/ciisrv/postinstall/cii-postinstall alarm
```

Auto-start

Auto-start the Alarm system on boot (requires root privs)

```
sudo cii-services --beta enable alarm
```

Health Checks

Check status of Alarm system

```
cii-services --beta info
```

The relevant lines in the output are “ias”, “kafka”, “zookeeper”, and “alarm-mon”

Log Configuration

Redefine log levels (requires root privs)

```
# IAS-Bridge
/elt/ciisrv/etc/log/cii-ias-bridge-logging.xml

# Alarm Monitor
/elt/ciisrv/etc/log/cii-alarm-mon-logging.json
```

Logs

Inspect Logs (requires root privs)

```
journalctl -e  -u cii_ias  -u cii_alarm_mon
```

10.5.2 Troubleshooting

1) Alarm-datapoint has value UNKNOWN (instead of NOMINAL etc.):

Try `ciiAlarmCtl daemon reload`, or (as root) `systemctl restart cii_ias`

2) To see the logs from the ias processes (requires root privs):

Use `journalctl -f -u cii_ias -u cii_alarm_mon -u kafka -u kafka-zookeeper`.

The ias core logs are in `/opt/IasRoot/logs`, `/var/log/kafka`, `/var/log/kafka-zookeeper`.

3) For debugging, it is possible to use the internal tools that come with IAS:

Set up the shell environment as described in #Run-IAS-directly. This enables you to run, e.g.:

- `ciiAlarmCtl check heartbeat`
- `iasDumpKafkaTopic`
- Ex.1: `iasDumpKafkaTopic -t plugin`
- Ex.2: `iasDumpKafkaTopic -t core | grep <AlarmId>`
- `ciiAlarmCtl check cdb`
- `iasCdbChecker -jCdb $ALARM_CDB_ROOT`

4) Visit the CII Knowledge Base: https://gitlab.eso.org/ecs/eltsw-docs/-/wikis/KnowledgeBase/CII

10.5.3 Modify CDB directly

The IAS offers a number of additional configuration options, but they require an understanding of the internals of the IAS. Before modifying the IAS CDB directly, read the detailed description of the Alarm system architecture [2], and for all details on alarm configuration attributes see [4] and [12].

Table 1 contains a brief explanation of the configuration elements used in IAS. For the proper explanation of the elements, refer to the IAS documentation [2], [3], [4].

Table 1: Alarm system elements

Entity Name	Description
ASCE	An alarm system computing element that runs inside DASU and contains one transfer function.
DASU	An alarm system element that runs inside Supervisor and contains one or many ASCEs.
IASIO	An alarm system input/output entity.
Supervisor	A standalone process that contains one or many DASUs.
Transfer Function	A transformation function that calculates alarm states. It receives and outputs IASIOs.
Plugin	A plugin for IAS system that interfaces the IAS system with external systems (in CII case OLDB).

General Settings

General settings are configured in `CDB/ias.json`

Set LogLevel, refresh rate, etc. according to the IAS CDB documentation [12].

IASIOs

IASIOs are input and output entities of the Alarm system. To define a new IASIO, go to `CDB/IASIO/iasios.json`

For the IASIO, write the URI of the input data point (e.g. “/alarmtest/device/motor/input_int_dp”, i.e. the relative path of the input data point “cii.oldb:/alarmtest/device/motor/input_int_dp”) and set the type (e.g INT).

Create another IASIO for the output alarm, set the id (e.g. “/motor/input_int_dp_alarm”). The type must be set to “ALARM”. You can define if the operator can suppress/hide the alarm by setting `canShelve` to “True”.

Other values for both IASIOs can be left empty.

The fields `shortDesc`, `docURL` and `emails` are optional, they provide additional information on the IASIO. Also note that the fields `canShelve`, `sound`, and `emails` are only applicable to IASIOs of type ALARM. Template field allows templatization of the IASIO (refer to the IAS documentation on templatization).

Transfer Functions

Alarm system output IASIOs are calculated by Transfer Functions. Transfer functions are in the IAS domain and must be implemented and built there. IAS already provides a small set of Transfer functions. Please refer [8] to [9] and for details on Transfer functions. To import a new transfer function, go to `CDB/TF/tfs.json` and add a new entry.

To register a newly implemented transfer function, write the `className` (e.g. “org.eso.ias.asce.transfer.impls.MinMaxThresholdTF”) and `implLang` (e.g. “SCALA”).

DASUs

DASUs are containers for ASCE entities, that make alarm calculations using the Transfer functions. First DASU needs to be defined followed by its ASCE(s) with Transfer function to be used.

To define a new DASU, go to `CDB/DASU/` and create a new file, or copy an existing one.

Set the name of the DASU file (e.g.”int_alarm_dasu.json”). Leave the other properties empty for the time being. The default logging level will be used and there will be no templatization of the DASU.

Go to `CDB/ASCE/` and define a new ASCE file (e.g. “int_alarm_asce.json”). For the `transferFunctionID` choose one of the transfer function names defined earlier (e.g. “org.eso.ias.asce.transfer.impls.MinMaxThresholdTF”). In the `outputId` field enter the output IASIO defined earlier (e.g. “/motor/input_int_dp_alarm”). For the `dasuId`, write the name of the DASU defined above, (e.g. “int_alarm_dasu”).

For the Inputs, add as an input the IASIO name (e.g. “/alarmtest/device/motor/input_int_dp”) to be the (only) input of the ASCE. In the properties, specify the parameters for the transfer function (e.g. “org.eso.ias.tf.minmaxthreshold.highOn” with value 5, “org.eso.ias.tf.minmaxthreshold.highOff” with value 1)

Note: For the MinMaxThresholdTF transfer function, analogous lower limits minmaxthreshold.lowOff and minmaxthreshold.lowOn can be defined. If both upper and lower limits are defined in the same ASCE, the alarm will not distinguish between the input value going above or below the limits. To distinguish between these two cases, one needs to define two ASCE with the same input, one checking for the high value and the other checking for the low value.

The Alarm priority is determined by the Transfer function. Some implementations allow the priority to be configurable, e.g. org.eso.ias.asce.transfer.impls.MinMaxThresholdTF has a property named org.eso.ias.tf.alarm.priority for that reason. Valid values are: SET_CRITICAL, SET_HIGH, SET_MEDIUM (default), SET_LOW.

To add the ASCE to deploy in the DASU, write the ASCE name into the DASU file (in `CDB/DASU/`) of the respective DASU (i.e. `int_alarm_asce`)

To define the output of the DASU, edit the DASU file, and write the `outputId` as defined in the IASIO file (i.e. “/motor/input_int_dp_alarm”).

Finally, save your configuration changes, and ask the Alarm system to re-read its config, as described in section #Usage above.

Supervisor

To deploy DASUs we need to define a Supervisor first. Supervisor is a stand-alone process that hosts DASUs among with its ASCEs including needed Transfer Function processing.

To define a new Supervisor, go to `CDB/Supervisors` and create a new file, or duplicate an existing one.

Set the supervisor's name (e.g. "test_supervisor"), and assign the hostname (e.g. "localhost"). In the field `DASUs to Deploy`, write the name of the DASU defined above, leave `templateId` and `instance` empty, meaning the DASU will not be processed as a template.

The `template` field defines what template to apply and the `instance` field defines the instance number used to apply when processing the template. For more information on templates refer to the IAS documentation.

Finally, save your configuration changes, and ask the Alarm system to re-read its config, as described in section #Usage above.

Plugin

The OLDB Plugin configuration foremost lists the set of IASIO IDs (each of them corresponding to an OLDB data point) to be monitored by the particular Plugin instance.

To configure how frequently the plugin should check for newly created input-DPs, set the property `dpconnRetrySec`, the default value is 60.

To define a new Plugin, go to `CDB/Plugins/` and create a new file, or duplicate an existing one. Name the new plugin "CiiPlugin" and set the monitored system to "OLDB". The `values` defines a set of data points to monitor. Add a new monitored value, and select the desired input. If needed change the "Refresh time" (for how long a value is valid if not updated) or other attributes.

Finally, save your configuration changes, and ask the Alarm system to re-read its config, as described in section #Usage above.

10.5.4 Run IAS directly

This is **not the recommended way**, because it is more involved and complicated. But if you cannot use the alarm system as a system service for some reason, it is possible to run it "in user space".

When you run the alarm system in this way, you have the freedom to store the alarm rules file(s) and CDB outside the standard location, i.e. not in /etc/cii_ias.

Run these commands as eltdev (or other user with write-permissions on the installation folders).

Set-up

```
mkdir -p $HOME/modulefiles
UPDATES=https://www.eso.org/~eltmgr/CII/latest/install
```

(continues on next page)

(continued from previous page)

```
wget -q -O $HOME/modulefiles/cii_ias.lua  $UPDATES/ias/cii_ias.lua-4.0.0
module load cii_ias
```

Start

```
ciiAlarmCtl  start  all+
```

Check

```
ciiAlarmCtl  is-active  all+
```

Reload

```
ciiAlarmCtl  reload cdb
```

Stop

```
ciiAlarmCtl  stop   all+
```

CHAPTER

ELEVEN

TELEMETRY

Document ID:	
Revision:	1.8
Last modification:	March 18, 2024
Status:	Released
Repository:	https://gitlab.eso.org/cii/srv/cii-srv
File:	telemetry.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
0.1	09.03.2020	amarincek, jpribosek, dsoklic	All	Document creation.
1.0	06.05.2020	dsoklic/btepinc	All	Review updates. Released document.
1.1	17.6.2020	jpribosek	All	Updating manual after receiving support questions.
1.2	7.7.2020	jpribosek	All	Updating manual according to raised LAT issues.
1.3	30.7.2020	jpribosek	All	Updating manual according to raised LAT issues.
1.4	27.8.2020	jpribosek	All	Updating manual according to raised LAT issues.
1.5	3.9.2020	jpribosek	All	Updating manual according to raised LAT issues. Added additional section for setting up OLDB simulator.
1.6	24.9.2020	jpribosek	All	Updating manual according to raised LAT issues.
1.7	06.10.2020	jpribosek	All	Updating manual according to the review comments.
1.8	18.03.2024	mschilli	0	CII v4: public doc

Confidentiality

This document is classified as Public.

Scope

This document is a manual for the Telemetry system of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

Glossary of Terms

API	Application Programming Interface
CII	Core Integration Infrastructure
CLI	Command Line Interface
DP	Data Point
EA	Engineering Archive
ES	ElasticSearch
JSON	Javascript object notation
OLDB	Online Database
SVN	Subversion
YAML	YAML Ain't Markup Language

References

1. Cosylab, ELT CII Configuration user manual, CSL-DOC-19-173189, v1.7
2. Cosylab, Middleware Abstraction Layer design document, CSL-DOC-17-147260 v 1.6
3. Elasticsearch website. https://www.elastic.co/
4. Cosylab, ELT CII – ICD Generation, User's Manual, CSL-MAN-19-170402, v1.4
1. Cosylab, CII Service Management Interface, CSL-DOC-18-163377, v1.2
2. Cosylab, ELT CII Online Database user manual, CSL-DOC-19-176283, v1.3
3. Cosylab, ELT CII Log user manual, CSL-DOC-20-181435, v1.3
4. Telemetry examples project: https://svnhq8.hq.eso.org/p8/trunk/CONTROL-SYSTEMS/CentralControlSystem/CII/CODE/telemetry/telemetry-examples/

11.1 Overview

This document is a user manual for usage of the Telemetry Service system. It explains how to use the Telemetry Archiver API, how to use the CLI tools, and how to configure the Telemetry Archiver.

All examples in this manual will be also presented in the telemetry-examples module.

11.2 Introduction

The purpose of the CII Telemetry Service is archiving the periodic, event-based and ad-hoc data that will be used for calculation of statistics, and the long-term trends and performance of the ELT telescope.

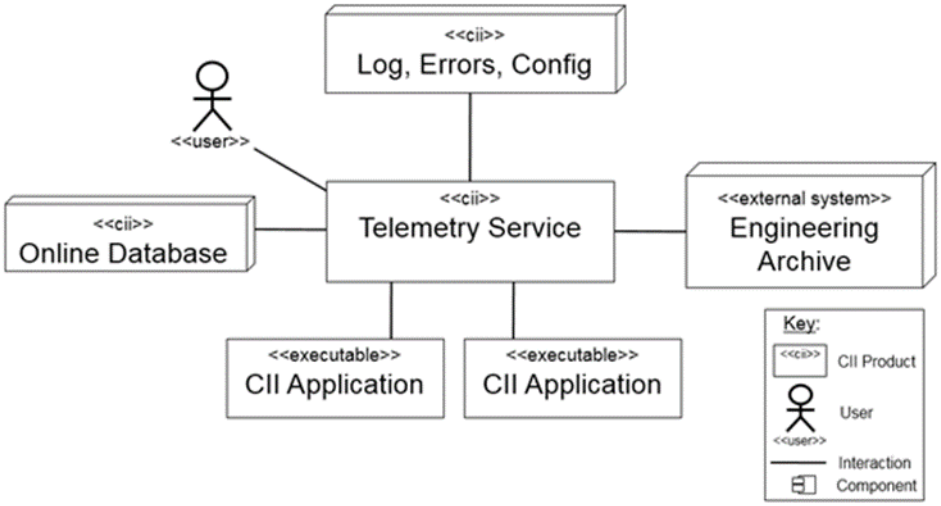


Figure 2-1: Telemetry Service API interactions

CII Telemetry Service is a central archiving service of the CII infrastructure. It is reliant on the other CII services:

- CII Online Database (OLDB) – the main source of data for archiving.
- CII Configuration Service (Config) – storing and retrieving data capture configurations.
- CII Log Service (Log) – used for storing Telemetry Service logs.
- CII Errors Service (Errors) – used for reporting Telemetry Service errors.

The data captured by the CII Telemetry service is stored into the Engineering Archive (EA).

The CII Telemetry Service consists of multiple CII Telemetry Archivers that are responsible for archiving the incoming telemetry data. The Telemetry Archiver is split into two parts:

1. **OLDB Subscription**: this part of the archiver subscribes to data point changes that are coming from the online database and archives them accordingly to the specified values in the data capture configurations. It also handles periodic archiving of the data point values according to the data capture configuration.
2. **Telemetry Client API**: this part of the archiver handles communication between the service and CII Applications. The Client API is split into two endpoints:

- Management endpoint – management of archiver (stop archiver, restart archiver, refresh archiver's configuration) and archiver statistics (number of data points that are being archived, data archive rate)

- Archive API endpoint – API used for storing ad-hoc data, querying, and downloading data from the Engineering Archive.

A high-level overview of a single CII Telemetry Archiver can be seen in Figure 2-2.

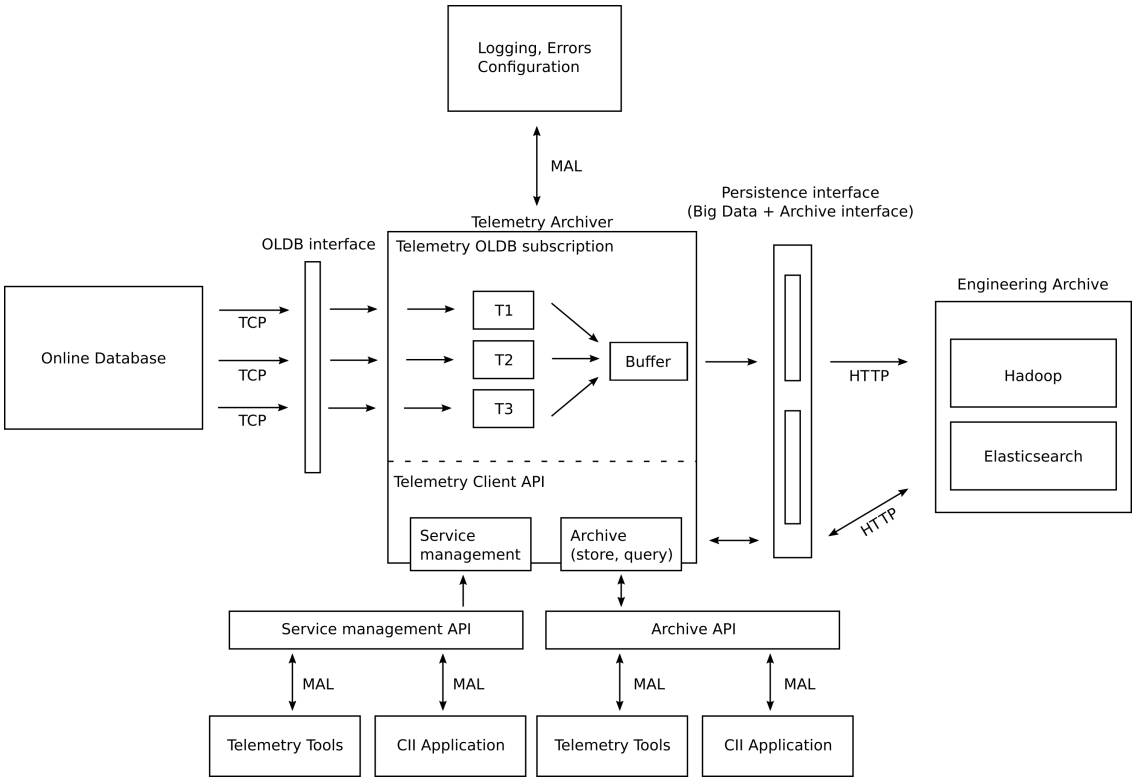


Figure 2-2: High-level overview of one Telemetry Archiver

The CII Telemetry Archiver accesses the CII Configuration service to retrieve its configuration. This configuration is used when the CII Telemetry Archiver receives data points to determine which should be archived and which should be discarded. When a data point that should be archived is received, it is stored in the Engineering Archive.

The Engineering Archive consists of two parts:

- Data Storage, which is used to archive smaller data (up to 2 GB); and
- Big Data Storage, which is used to archive large data.

11.2.1 Telemetry OLDB Subscription

CII Telemetry Archiver subscribes to OLDB data points via OLDB client and archives the data point values according to the data capture configurations that are stored in configuration service. Each data capture configuration is used to provide archiving options for one data point. It is possible to define multiple data capture rules for the same data point URI on the same CII Telemetry Archiver – e.g. data point could be archived on both value change and quality flag change on the same Telemetry Archiver.

Each Telemetry Archiver has defined its unique specified range in the form of a URI (i.e. *serviceRange*) that defines the location in configuration service from where the data capture configurations will be loaded from. For example, if the *serviceRange* field in Telemetry Archiver configuration is set to cii.config:///*telemetry/service1, then all data capture configurations placed under /telemetry/service1/datacapture/ URI on configuration service will be used (an example of valid uri would then be cii.config:///*/telemetry/service1/datacapture/c722835e-cb22-4819-adb6-7cfe60c1ff39). Extra note: data capture configurations should be placed under <SERVICE RANGE>/datacapture/ uri.

For each data point that will be archived a data capture configuration must be created. Since the data capture configurations are going to be stored in the CII Configuration service a CIIBasicDataTypes can be used for configuration fields. The data capture settings are described in Appendix B.

The changes of the data capture configurations are only propagated when the end-user calls the refreshConfiguration method on the Service Management API (see section 5.4). This will reload the specified configuration (or all configurations) without interrupting the ongoing archiving tasks except as required by the modified configuration. The configuration service will not be continuously polled for data capture configuration changes. When the Telemetry Archiver is started (initialized), the data capture configurations are downloaded via CII config client from the location specified in *serviceRange*. The Telemetry OLDB Subscription subscribes to data point changes via OLDB client. Internally the OLDB client maintains a pool of data point subscriptions.

Telemetry Archivers do not know about one another and it's the end user's responsibility to distribute data points across them depending on the expected service workload. This allows easy horizontal scaling of the Telemetry Archivers depending on the amount of data that we would like to archive. If more throughput is needed, additional Telemetry Archivers can be deployed.

Each Telemetry Archiver contains its configuration stored in the CII Configuration Service and connects to the CII Configuration Service using the CII Configuration Client. The client by default uses the ciiconfservicehost as the host location of CII Configuration Service (ciiconfservicehost is the hostname of the CII Configuration Service that should be resolved either by local DNS or custom /etc/hosts file). The default Telemetry Archiver configuration is stored under the cii.config://*/telemetry/config/defaultarchiverconfig.

Multiple service range entries can be defined per one Telemetry Archiver configuration.

To specify the custom configuration for another Telemetry Archiver, a custom Telemetry Archiver configuration YAML must be provided. Appendix C contains a description of all configuration fields and an example of a custom YAML configuration.
Figure 2-3 shows an example of 2 Telemetry Archivers. Telemetry Archiver 1 uses the default configuration and the Telemetry Archiver 2 uses custom configuration. They both communicate with the configuration service to retrieve the service range and data capture configurations. Each Telemetry Archiver configuration specifies service ranges under which the data capture configurations are found. The data capture configuration, which is in subpath of specified service range, specifies the details about the behavior of the data capture (OLDB data point URI, delta value, . . .).

When the Telemetry Archiver starts up, the data point values (CiiOldbDpValue) of the subscribed data points (as defined in data capture configurations) are downloaded from OLDB and stored in memory. Every time the data point value of the subscribed data point changes, they are checked against the data capture configurations to determine if the data point is suitable for archiving.

The subscribed data point will be archived depending on the following cases:

- **Value change**: if the data point value changed and if the difference between the old and new data point value has changed for more than the specified delta value and if duration since the last archive is bigger than the minimum interval duration (set in data capture configuration) the data point will be archived.
- **Metadata change**: if the data capture configuration is capturing based on the metadata change, the data point value will be archived whenever the metadata has changed
- **Quality change**: if the data capture configuration is capturing based on the quality change and the data point quality flag has changed, the data point value will be archived.
- **Maximum interval**: the purpose of the maximum interval is to archive the data point value in the absence of a data point value change event. If the maximum interval is set, a periodic task will be submitted to Telemetry Archiver's scheduled executor that will try to archive the last data point value according to the specified period. Timestamp since the last archive event for the given data capture configuration will be stored in the memory of the archiving worker.
- **Minimum interval**: the purpose of the minimum interval is to not archive too many data point value changes in a short timeframe. If the minimum interval is set and the duration between now and time, since the last value was archived, is bigger than the minimum interval duration, the data point value will be archived

It is possible to combine the above rules to make more fine-grained archiving behavior. For example, one can create two data capture configurations:

1. Data capture configuration with value archiving behavior and maximum time interval set to 10 seconds.
2. Data capture configuration with quality archiving behavior.

In such a case, the data point will be archived whenever the quality flag of the data point has changed

or whenever the value has changed or every 10 seconds if the value was not archived in the last 10 seconds.

If the data point is valid for archiving according to the rules described above, the archiving worker thread will transform a data point configuration into a data packet described and insert it into the archiving buffer. This operation preserves the data of the data point (such as quality flags and source timestamps). If no data point source timestamp exists, a current wall clock time is set with microsecond resolution. If no quality flag is associated with the data point data, an Unknown quality flag is assigned.

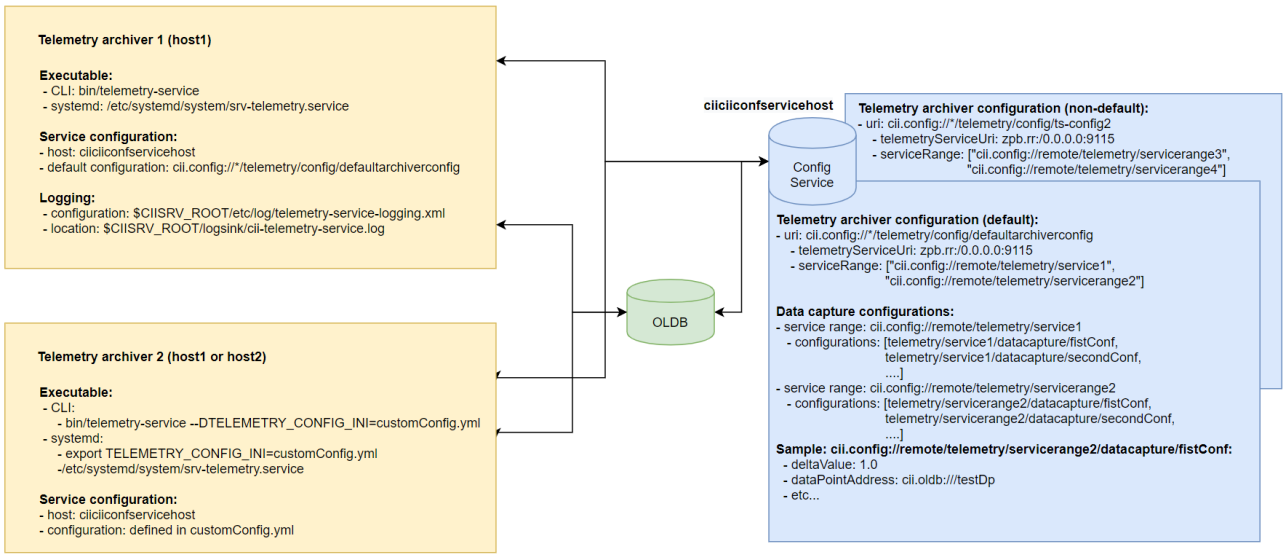


Figure 2-3: Communication and configuration of Telemetry Archivers

Data Packet

The Data Packet is a data transfer object used for transferring data point values and ad-hoc data between Telemetry Archiver and its clients. The Telemetry Archiver clients could be written in one of the three supported languages (Java, C++, Python). For this reason, the data packet is generated with the MAL ICD [4]. The Data Packet contains the following fields:

Table 2-1: Data packet fields description

Field	Type	Description	Example
Timestamp (automatically assigned)	Double	Representing the point in time when the data packet was archived (in UNIX time)	1571914537.0
Data (required)	String	Data is stored as a valid JSON string	{“temperature”: 30, “comment”: “ELT motor temperature”}
Big Data File Reference (optional)	String	A reference / location of the big data file in the large storage service. The client has to store a large file into the large storage service before archiving the data packet	

11.2.2 Telemetry Client API

Telemetry Client API provides an interface for easier interaction with the archiving system. The API could be used by the command line tools and other CII Applications. Telemetry Client API consists of three parts:

- **Archive API** – API used for storing, querying, and retrieving data from the Engineering Archive.
- **Service Management API** – API used for managing Telemetry Archiver and providing important statistics about the Telemetry Archiver internals.
- **Telemetry Tools** – CLI tools that make interaction with the Telemetry Archiver easier.

In order to provide an API to CII Applications that are written in different languages, a MAL ZPB service is used internally for handling the client requests. Client API service exposes two ICD interfaces:

- Archive API interface, the service is exposed on:
  zpb.rr://server:9115/telemetry/service/archive
- Service Management interface, the service is exposed on:
  zpb.rr://server:9115/telemetry/service/management

Archive API

Archive API shall be used by CLI tools and other CII Applications that need to interact with the Engineering Archive. Internally an Archive API is communicating with the archiving system through the Persistence Interface using the Telemetry Persistence Implementation.

Archive API will receive ad-hoc data from the clients in the form of data packets which will be stored irrespective of the age of the data into the EA.

Archiving API does not support storing data packets that contain large data blobs directly as that would result in a lot of unnecessary network traffic. The clients that will be using the Archiving API will

have to first store the large files into the Large Storage Service and then store the data packet with the location reference of the blob into the Engineering Archive.

To make storing large data files to Large Storage Service easier, a library implementing the Large Storage Service interface is provided. The provided implementation will interact with the Hadoop storage system.

Archiving API handles any valid data package irrespective of the sampling frequency of the data. The actual speed of storing large big data blobs depends on the used hardware.

Service Management API

The CII Telemetry Archiver is a *systemd* service and should be started, stopped or restarted via *systemd* commands:

- **Start** - start the service if it's not already running
- **Restart** - restart the Telemetry Archiver. All currently ongoing tasks will finish before the service will be restarted.
- **Stop** - shutdown the Telemetry Archiver. All currently ongoing tasks will finish before the service is stopped

The instructions for manipulating the CII Telemetry Archiver with *systemd,* please check the CII Telemetry Transfer document.

The Service Management Client API exposes a *RefreshConfiguration* command which refreshes the specified data capture configurations from the configuration service without interrupting the ongoing archiving tasks.

As part of the Service Management API, metrics about the current state of the Telemetry Archiver are exposed via the JMX interface. For more information about the JMX interface see the design document for Service Management Interface [5]. For the Telemetry Archiver, the following metrics are exposed:

General metrics:

- Current time.
- Telemetry Archiver uptime.
- Current Memory consumption.
- Current CPU consumption.
- The current number of archiving worker threads.
- Engineering Archive health (periodic health checks reporting if the archiving system is still online).

Archiving worker threads:

- The number of currently subscribed data points.

- **The number of current data points that are being periodically** archived.
- **The number of dropped data point values (archiving buffer was full** and a data point value was dropped).

Archiving buffer statistics:

- Buffer fullness - the percentage of the buffer that is full.
- Archiving throughput - number of archived data points in the last hour.
- The number of errors that occurred while storing packets into the Engineering Archive in the last hour.
- The total number of archived data points for the current day.

Telemetry CLI tools

The Telemetry CLI tools are separated into 2 modules:

- **CLI Manager is a tool that allows a user to send the Telemetry** Archiver a notification to trigger a refresh of its configurations.
- **CLI Archiver is a tool enabling users to store, download, and query** data packets from and to the Engineering Archive. The tool provides the user with three different commands: store, download, and query.

11.3 Prerequisites

This section describes the prerequisites for using the Telemetry Service, API, and applications. Note that the preparation of the environment (i.e. installation of required modules, configuration, running of services) is not in the scope of this document. Consult your administrator and the Telemetry Service TRD document for information on how to establish the environment for usage of the Telemetry Archiver, API, and applications. Before using the CII Telemetry make sure that telemetry configuration is initialized with *telemetry-initEs* command.

11.3.1 Services

Engineering Archive

The Engineering Archive (EA) stores the data that the Telemetry Archiver receives and decides to archive. The Engineering Archive is, in fact, a set of services. The EA has been initially designed to use Persistence interface to access Elasticsearch and Hadoop, but can be modified to use other forms of storage.

Configuration Service

The CII Telemetry Service uses CII Configuration Service to store its configuration. The Configuration service must be accessible for the Telemetry Archiver to successfully initialize.

OLDB Service

The Online Database service is the main source of data that provides data point updates which are checked against the capturing rules defined by the Telemetry Archiver (TA) and later archived in the archiving system. For testing purposes, the OLDB service could be replaced with the OLDB simulator that generates the data point value updates based on the capturing rules.

11.4 Data Capture Configuration

11.4.1 Create Data Capture Configuration

Before the Telemetry Archiver can archive data packets, it must be configured. The data capture configuration can be done using the Configuration GUI.

The following steps are necessary for configuration:

1. In the GUI click on Options and tick the Write enabled checkbox.
2. In the Target Config tab click on the Add button.
3. The dialog shown in Figure 4-1 will open. In this dialog the new configuration can be specified.

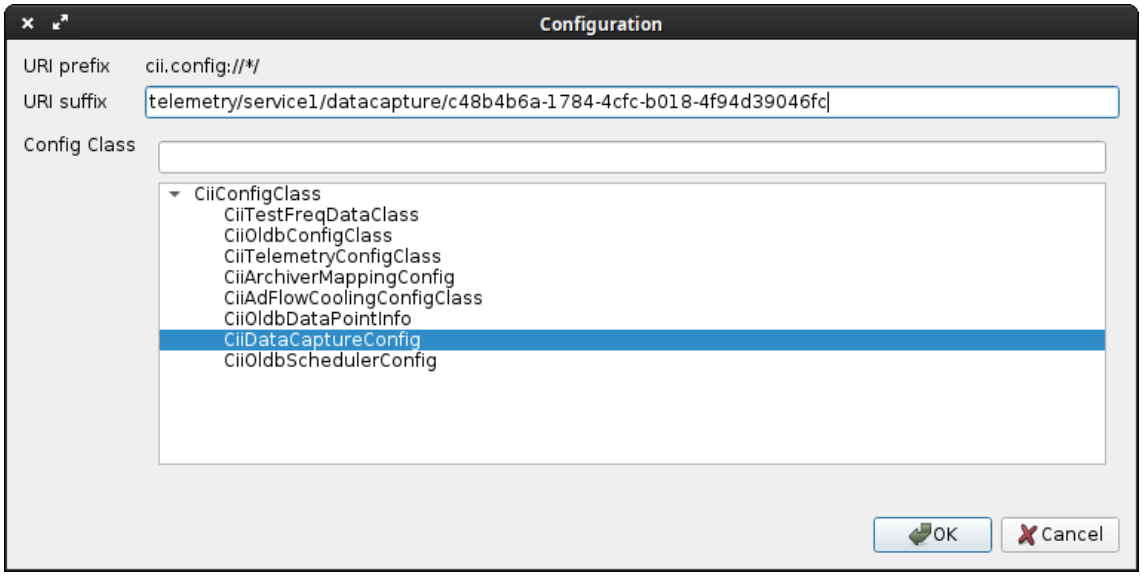


Figure 4-1: Creation of new configuration

4. Specify the following:
   a. URI suffix: The suffix needs to follow the following format: <SERVICE_RANGE**>**/datacapture/<UUID>. The first part needs to be the range under which the Telemetry Archiver is running (e.g. telemetry/service1 in the example). The second part needs to be a unique ID. A tool such as uuidgen can be used to generate this UID.

      NOTE: when generating a unique ID with the uuidgen tool, use only the generated ID that starts with a letter, regenerate an ID if it starts with a number.

   b. Config Class: Select CiiDataCaptureConfig.
5. Click Ok. The data capture configuration should now be stored on the configuration service.
6. Edit the created data capture configuration fields by following the steps in the next section 4.2.

11.4.2 Edit Data Capture Configuration

1. Select the data capture configuration in the tree menu in the Target Config tab.
2. Fill in the configuration parameters. The example in Figure 4-2 shows a capture configuration that is storing value changes. The individual fields are listed and documented in Table 9.4.

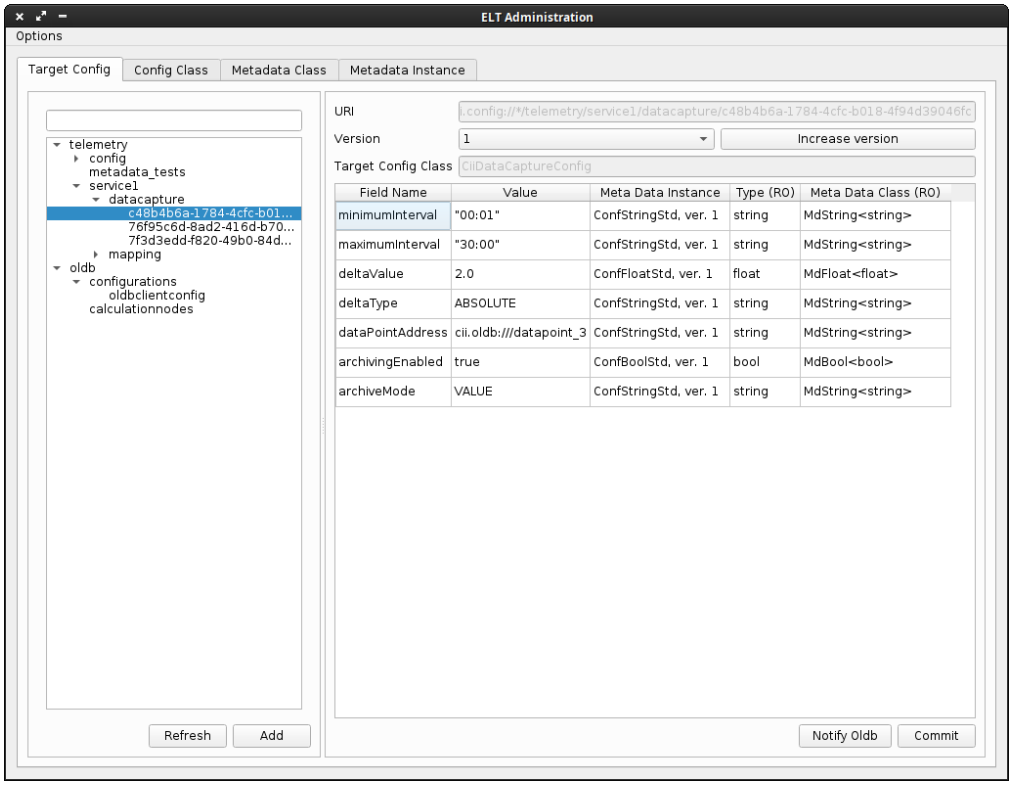


Figure 4-2: Configuring a data capture configuration

3. Click on the Commit button.
4. Before the Telemetry Archiver can use the new configuration, Telemetry Archiver must be notified/refreshed about the updated configuration. For that, one can use the CLI tool described in section 6.1. The ID that the CLI tool expects for refreshing configuration is the URI of the configuration that is selected in the tree view on the left side of the GUI.

11.5 Telemetry Archiver API Usage

This section provides examples on how to use the Archive API and Service management API in an application. The Archive API provides users with the functionality to store, query, and download data as well as write, read, and delete big data that is saved in the Engineering Archive. The Service management API provides users with the functionality to refresh configurations.

All examples assume that CII Telemetry Archiver is running on localhost. This can be changed to the *ciiarchivehost* accordingly.

11.5.1 Basic Example

The following section demonstrates a basic step-by-step example workflow of using the Telemetry API.

The basic example starts by first configuring the data capture. Once the configuration service has processes and saved the new configuration, we refresh the configuration on the Telemetry Archiver. After a short while, the Telemetry Archiver will subscribe to the data point.

At this point, the Telemetry Archiver is archiving data points that match the provided configuration. The basic example continues by connecting to the Archive API and queries it to retrieve the archived data point IDs. The retrieved data point IDs are used to download the data packets. Upon downloading the packets, the data points data is displayed.

The APIs are documented in sections 5.2, 5.3, and 5.4.

Listing 5-1 shows this basic example in Java (TelemetrySubscriptionExample.java). This can be done in CPP and Python as well.

Listing 5-1: The basic example in Java

```
import java.net.URI;
import elt.mal.CiiFactory;
import elt.mal.rr.qos.QoS;
import elt.mal.rr.qos.ReplyTime;
import elt.mal.zpb.ZpbMal;
import elt.telemetry.archive.DataPacket;
import elt.telemetry.archive.TelemetryArchiveApiSync;

public class TelemetrySubscriptionExample {
  public static void main(String[] args) throws Exception {
    final URI telemetryServiceRange = URI.create("cii.config://*/telemetry/
↪service1");
    final URI uniqueCaptureLocation = URI
        .create(String.format("%s/%s", telemetryServiceRange, UUID.randomUUID().
↪toString()));
    final URI oldbDatapointUri = URI.create("cii.oldb:///datapoint_1");
    final URI telemetryManagementEndpoint = URI
        .create("zpb.rr://localhost:9115/telemetry/service/management");
```

(continues on next page)

(continued from previous page)

```
    final URI telemetryArchiveEndpoint = URI
        .create("zpb.rr://localhost:9115/telemetry/service/archive");
    // start of the actual program
    CiiConfigClient configClient = CiiConfigClient.getInstance();
    configClient.setWriteEnabled(true);
    DataCaptureConfiguration captureConfig =␣
↪createCaptureConfig(oldbDatapointUri);
    configClient.saveTargetConfig(uniqueCaptureLocation, captureConfig);
    // wait for the configuration service to process and save request
    Thread.sleep(2_000);
    // refresh specific data capture configuration
    refreshConfiguration(Arrays.asList(uniqueCaptureLocation.toString()),
        telemetryManagementEndpoint);
    // wait for the telemetry archiver to subscribe to data point (if it's not␣
↪already)
    Thread.sleep(2_000);
    // query the stored data point from the archive
    TelemetryArchiveApiSync archiveApi =␣
↪connectToArchiveApi(telemetryArchiveEndpoint);
    List<String> ids = archiveApi.queryData("data.uri:*datapoint_1*");
    assert ids.size() > 0 : "At least one data point should be present";
    // download queried ids
    List<DataPacket> packets = archiveApi.downloadData(ids);
    for (DataPacket packet : packets) {
      // process packets
      System.out.println(packet.getData());
    }
    // close resources
    configClient.close();
    archiveApi.close();
    System.exit(0);
  }
}

public static void refreshConfiguration(List<String> refreshIds, URI␣
↪managementEndpoint) {
  CiiFactory factory = null;
  TelemetryServiceManagementSync client = null;
  try {
    factory = CiiFactory.getInstance();
    factory.registerMal("zpb", new ZpbMal());
    client = factory.getClient(managementEndpoint,
        new QoS[]{new ReplyTime(10, TimeUnit.SECONDS)}, new Properties(),
        TelemetryServiceManagementSync.class);
    if (refreshIds.isEmpty()) {
      client.refreshAllConfigurations();
    } else {
      client.refreshConfigurations(refreshIds);
    }
```

(continues on next page)

(continued from previous page)

```
  } catch (CiiSerializableException e) {
    String msg = String.format("Failed to refresh configurations %s", e.
↪getCiiMessage());
    System.out.println(msg);
  } catch (Exception e) {
    System.out.println("Failed to refresh configuration: " + e);
  } finally {
    if (client != null) {
      client.close();
    }
    if (factory != null) {
      factory.close();
    }
  }
}
public static TelemetryArchiveApiSync connectToArchiveApi(URI␣
↪telemetryClientEndpoint) throws Exception {
  CiiFactory factory = CiiFactory.getInstance();
  factory.registerMal("zpb", new ZpbMal());
  TelemetryArchiveApiSync client = factory.getClient(telemetryClientEndpoint,
      new QoS[]{new ReplyTime(10, TimeUnit.SECONDS)}, new Properties(),
      TelemetryArchiveApiSync.class);
  CompletableFuture<Void> connectionFuture = client.asyncConnect();
  connectionFuture.get(5, TimeUnit.SECONDS);
  return client;
}
public static DataCaptureConfiguration createCaptureConfig(URI oldbDatapointUri)

    throws CiiConfigDataLimitsException, CiiInvalidTypeException {
  DataCaptureConfiguration captureConfig = new DataCaptureConfiguration(
      "dataCaptureConfigTest", // name
      "",                          // id is automatically set on store
      oldbDatapointUri.toString(),   // to which data point we want to subscribe␣
↪to
      1.0f,                 // delta value change
      "ABSOLUTE",           // delta type set to absolute changes
      "VALUE",            // archive mode set to value changes
      "00:01",         // minimum interval (at most once per second)
      "10:00",           // maximum interval (at least once every 10 minutes)
      true            // archiving enabled
  );
  return captureConfig;
}
```

11.5.2 Archive API: storing, querying and downloading data

A short description of the part of the Archive API used for storing, querying and downloading data can be found in A.1.

Imports/includes

Usage of the Telemetry Archive API requires the following imports to establish a MAL connection: CiiFactory, QoS, ReplyTime, and ZpbMal. The TelemetryArchiveApiSync interface defines the Telemetry Archive API methods.

An example of an application that uses the Telemetry Archive API can be found in the main method of the *TelemetryArchiveAPIExample.java* file in the telemetry-examples project (see [8]). Similar examples can be found in cpp/archive_api and python/archive_api folders.

For a list of necessary libraries in order to connect to the API make sure to check the wscript files in cpp, java and python folders of the telemetry-examples project.

Java

```
import java.net.URI;
import elt.mal.CiiFactory;
import elt.mal.rr.qos.QoS;
import elt.mal.rr.qos.ReplyTime;
import elt.mal.zpb.ZpbMal;
import elt.telemetry.archive.DataPacket;
import elt.telemetry.archive.TelemetryArchiveApiSync;
import com.fasterxml.jackson.databind.ObjectMapper;
```

CPP

```
#include <mal/Cii.hpp>
#include <mal/utility/LoadMal.hpp>
#include <mal/rr/qos/ReplyTime.hpp>
#include <CiiTelemetryArchiveApi.hpp>
```

Python

```
import elt.pymal as mal
from ModCiiTelemetryArchiveApi.Elt.Telemetry.Archive import DataPacket
from ModCiiTelemetryArchiveApi.Elt.Telemetry.Archive.
↪TelemetryArchiveApi import TelemetryArchiveApiSync
```

Connection to Telemetry Archiver using MAL

To establish a MAL connection between our application and the Archive API endpoint, we first get an instance of CiiFactory, register a MAL connection on it and get a TelemetryArchiveApiSync client that connects to the URI **zpb.rr://localhost:9115/telemetry/service/archive**. We then wait until the connection is established.

Listing 5-2, Listing 5-3 and Listing 5-4 demonstrate how to establish a MAL connection between an application and the Archive API endpoint.

Connection in Java

Listing 5-2: Example of connection to the Telemetry Archiver using Java

```
CiiFactory factory = CiiFactory.getInstance();
Mal mal = new ZpbMal();
factory.registerMal("zpb", mal);
final URI endpointUri = URI.create("zpb.rr:// localhost:9115/telemetry/service/
↪archive");
TelemetryArchiveApiSync client = factory.getClient(endpointUri,
        new QoS[]{new ReplyTime(10, TimeUnit.SECONDS)}, new Properties(),
        TelemetryArchiveApiSync.class);
CompletableFuture<Void> connectionFuture = client.asyncConnect();
connectionFuture.get(10, TimeUnit.SECONDS);
```

Connection in CPP

Listing 5-3: Example of connection to the Telemetry Archiver using CPP

```
int status = 0;
const elt::mal::Uri archiveApiEndpoint("zpb.rr://localhost:9115/telemetry/
↪service/archive");

// Register MAL and obtain client for archive API
std::shared_ptr<::elt::mal::Mal> malInstance = elt::mal::loadMal("zpb", {});
::elt::mal::CiiFactory &factory = elt::mal::CiiFactory::getInstance();
factory.registerMal("zpb", malInstance);
```

(continues on next page)

(continued from previous page)

```
std::unique_ptr<elt::telemetry::archive::TelemetryArchiveApiSync> client =
   factory.getClient<elt::telemetry::archive::TelemetryArchiveApiSync>(
      archiveApiEndpoint,
      {std::make_shared<elt::mal::rr::qos::ReplyTime>(std::chrono::seconds(10))},
      {});

if (client == nullptr) {
  std::cerr << "Could not obtain client for: " << archiveApiEndpoint.string() <<␣
↪std::endl;
  status = 1;
} else {
  ::elt::mal::future<void> connectionFuture = client->asyncConnect();
  std::future_status futureStatus = connectionFuture.wait_
↪for(boost::chrono::seconds(10));
  if (futureStatus != boost::future_status::ready) {
    std::cerr << "Connection timeout to " << archiveApiEndpoint.string() <<␣
↪std::endl;
    status = 2;
  }
}
```

Connection in Python

Listing 5-4: Example of connection to the Telemetry Archiver using Python

```
archive_api_endpoint = mal.Uri('zpb.rr://localhost:9115/telemetry/service/archive
↪')

# Initialize Mal
factory = mal.CiiFactory.getInstance()
mal_instance = mal.loadMal('zpb', {})
if mal_instance is None:
    raise RuntimeError('Could not load zpb mal')
factory.registerMal('zpb', mal_instance)
qos = [mal.rr.qos.ReplyTime(datetime.timedelta(seconds=10))]

# Obtain client and connect to Telemety service
client = factory.getClient(archive_api_endpoint, TelemetryArchiveApiSync,
                           qos, {})
connection_future = client.asyncConnect()
if connection_future.wait_for(datetime.timedelta(seconds=10)) == mal.
↪FutureStatus.TIMEOUT:
    raise RuntimeError('Could not to connect to %s, timeout' % (archive_api_
↪endpoint.string(),))
_ = connection_future.get()
```

Storing a data packet

The data packet should contain the following fields:

- archived: timestamp when the data packet was archived
- bigDataLocation: reference to the big data file stored in Hadoop. This is an optional field.
- data field that contains:
  - timestamp: data point timestamp in microsecond resolution
  - uri: data point URI
  - value: data point value
  - quality: data point quality flag
  - metadataReference: data point metadata reference

When storing a data packet via the Archive API, the data point URI could be an arbitrary string and the URI protocol does not have to start with cii.oldb.

Here is an example of how the subscribed data point value update will be serialized in ElasticSearch (fields prefixed with _ are autogenerated ElasticSearch internal fields and are not fetched when retrieving data):

Listing 5-5: Archived metadata

```
{
"_id" : "5HEb9m8Ba7LLWT1elmN_",
"_source" : {
"archived" : 1580382019114000,
"data" : {
"@type" : "MdOldbNumber",
"value" : 2.0,
"comment" : "",
"checked" : false,
"metadataInstanceVersion" : 1,
```

```
"@name" : "telemetry_meta1",
"@genType" : "SINGLE",
"metadataInstance" : "telemetry_meta1"
}
}
}
```

Listing 5-6: Archived data point value

```
{
"_id" : "5XEb9m8Ba7LLWT1emGP8",
"_source" : {
"archived" : 1580382019737000,
"data" : {
"uri" : "cii.oldb:///datapoint_1",
"metadataReference" : "5HEb9m8Ba7LLWT1elmN_",
"quality" : "OK",
"value" : 0.5,
"timestamp" : 1580382018598000
}
"bigDataLocation" : "4a9a3810-1efb-4d39-b8fb-751f09c91ea8"
}
}
```

Java client

To store data, we first create a DataPacket object containing metadata. In the provided example, the static method TelemetryArchiveAPIExample.createTestMetadata serves this purpose. It first creates the data packet class and then sets its fields: a timestamp is applied, the location of the data in the big data database of the Engineering Archive is specified and the data is added in JSON format.

The following sample code is an excerpt from the code file named TelemetryArchiveAPIExample.java, stored in the java/sample-app folder.

Listing 5-7: Example of data packet metadata creation in Java

```
public static DataPacket createTestMetadata(Mal mal) {
    DataPacket dataPacket = mal.createDataEntity(DataPacket.class);
    dataPacket.setArchived(Instant.now().toEpochMilli());
    dataPacket.setBigDataLocation("");

    Map<String, Object> map = new HashMap<>();
    map.put("@type", "MyOldbNumber");
    map.put("value", 2.0);
    map.put("comment", "");
    map.put("checked", false);
    map.put("metadataInstanceVersion", 1);
    map.put("@name", "telemetry_meta1");
    map.put("@genType", "SINGLE");
    map.put("metadataInstance", "telemetry_meta1");
    JSONObject json = new JSONObject(map);
    dataPacket.setData(json.toString());

    return dataPacket;
}
```

To store the created data packet, we call the storeData method of the TelemetryArchiveApi interface. It returns a string specifying the assigned ID of the data packet containing metadata.

Listing 5-8: Example of storing of metadata data packet in Java

```
DataPacket metadata = TelemetryArchiveAPIExample.createTestMetadata(mal);
String metadataReference = client.storeData(metadata);
```

We proceed by creating another DataPacket object representing the data point value by passing the ID of the data packet containing the corresponding metadata (metadataReference) to the static method TelemetryArchiveAPIExample.createTestDataPointValue.

Listing 5-9: Example of creating a data packet in Java

```
public static DataPacket createTestDataPointValue(Mal mal, String␣
↪metadataReference) {

    DataPacket dataPacket = mal.createDataEntity(DataPacket.class);
```

(continues on next page)

(continued from previous page)

```
    dataPacket.setArchived(Instant.now().toEpochMilli());
    dataPacket.setBigDataLocation("bigDataLocation");

    Map<String, Object> map = new HashMap<>();
    map.put("uri", "cii.client:///datapoint_1");
    map.put("metadataReference", metadataReference);
    map.put("quality", "OK");
    map.put("value", 0.5);
    map.put("timestamp", Instant.now().toEpochMilli() - 10_000);
    JSONObject json = new JSONObject(map);
    dataPacket.setData(json.toString());

    return dataPacket;
}
```

To store the created data packet, we call the storeData method of the TelemetryArchiveApi interface. It returns a string specifying the assigned ID of the data packet in the Engineering Archive.

Listing 5-10: Example of storing of data packet in Java

```
DataPacket dataPointValue =
    TelemetryArchiveAPIExample.createTestDataPointValue(mal, metadataReference);
String id = client.storeData(dataPointValue);
```

CPP client

The following listings contain a sample CPP client that stores a datapacket. The sample code is an excerpt from the code file named app.cpp stored in the cpp/archive-api folder. The code follows the same flow as the Java code.

First, a metadata DataPacket is created.

Listing 5-11: Example of data packet metadata creation in CPP

```
std::shared_ptr<elt::telemetry::archive::DataPacket> createTestMetadata(
    std::unique_ptr<elt::telemetry::archive::TelemetryArchiveApiSync> &client) {
  std::shared_ptr<elt::mal::Mal> malInstance = client->getMal();
  std::shared_ptr<elt::telemetry::archive::DataPacket> dataPacket =
      malInstance->createDataEntity<elt::telemetry::archive::DataPacket>();
  long timestamp = timestampMillis();
  dataPacket->setArchived(timestamp);
  dataPacket->setBigDataLocation("");
  boost::property_tree::ptree tree;
  tree.put("@type", "MyOldbNumber");
  tree.put("value", 2.0);
  tree.put("comment", "");
  tree.put("checked", false);
```

(continues on next page)

(continued from previous page)

```
  tree.put("metadataInstanceVersion", 1);
  tree.put("@name", "telemetry_meta1");
  tree.put("@genType", "SINGLE");
  tree.put("metadataInstance", "telemetry_meta1");
  std::stringstream data;
  boost::property_tree::json_parser::write_json(data, tree);
  dataPacket->setData(data.str());
  return dataPacket;
}
```

To store the data packet, we call the storeData method of the client.

Listing 5-12: Example of storing of metadata data packet in CPP

```
std::shared_ptr<elt::telemetry::archive::DataPacket> metadata =␣
↪createTestMetadata(client);
std::string metadataReference = client->storeData(metadata);
```

We proceed by creating another DataPacket object representing the data point value.

Listing 5-13: Example of creating a data packet in CPP

```
std::shared_ptr<elt::telemetry::archive::DataPacket> createTestDatapointValue(
    std::unique_ptr<elt::telemetry::archive::TelemetryArchiveApiSync> &client,
    const std::string &metadataReference) {
  std::shared_ptr<elt::mal::Mal> malInstance = client->getMal();
  std::shared_ptr<elt::telemetry::archive::DataPacket> dataPacket = malInstance->
↪createDataEntity<elt::telemetry::archive::DataPacket>();
  long timestamp = timestampMillis();
  dataPacket->setArchived(timestamp);
  dataPacket->setBigDataLocation("bigDataLocation");
  boost::property_tree::ptree tree;
  tree.put("uri", "cii.client:///datapoint_1");
  tree.put("metadataReference", metadataReference);
  tree.put("quality", "OK");
  tree.put("value", 9.5);
  tree.put("timestamp", timestamp - 10000.0);
  std::stringstream data;
  boost::property_tree::json_parser::write_json(data, tree);
  dataPacket->setData(data.str());
  return dataPacket;
}
```

To store the data packet, we call the storeData method of the client.

Listing 5-14: Example of storing of the data packet in CPP

```
std::shared_ptr<elt::telemetry::archive::DataPacket> datapoint =␣
↪createTestDatapointValue(client, metadataRef);
std::string dataReference = client->storeData(datapoint);
```

Python client

The following listings contain a sample Python client that stores a data packet. The sample code is an excerpt from the code file named cii-telemetry-examples-archive-py.py stored in the python/archive_api folder. The code follows the same flow as the Java code.

First, a metadata DataPacket is created.

Listing 5-15: Example of data packet metadata creation in Python

```
def create_test_metadata(client):
    mal_instance = client.getMal()
    data_packet = mal_instance.createDataEntity(DataPacket)
    data_packet.setArchived(datetime.datetime.now().timestamp() * 1000.0)
    data_packet.setBigDataLocation('')
    data = {'@type': 'MyOldbNumber',
            'value': 2.0,
            'comment': '',
            'checked': False,
            'metadataInstanceVersion': 1,
            '@name': 'telemetry_meta1',
            '@genType': 'SINGLE',
            'metadataInstance': 'telemetry_meta1'}
    data_packet.setData(json.dumps(data))
    return data_packet
```

To store the data packet, we call the storeData method of the client.

Listing 5-16: Example of storing of metadata data packet in Python

```
metadata = create_test_metadata(client)
metadata_reference = client.storeData(metadata)
```

We proceed by creating another DataPacket object representing the data point value.

Listing 5-17: Example of creating a data packet in Python

```
def create_test_datapoint_value(client, metadata_reference):
    mal_instance = client.getMal()
    # timestamp must be in millis
    timestamp = datetime.datetime.now().timestamp() * 1000.0
    data_packet = mal_instance.createDataEntity(DataPacket)
    data_packet.setArchived(timestamp)
    data_packet.setBigDataLocation('bigDataLocation')
    data = {'uri': 'cii.client:///datapoint_1',
            'metadataReference': metadata_reference, # WATCH OUT
            'quality': 'OK',
            'value': 0.5,
            'timestamp': timestamp - 10000.0}
    data_packet.setData(json.dumps(data))
    return data_packet
```

To store the data packet, we call the storeData method of the client.

Listing 5-18: Example of storing of data packet in Python

```
datapoint_value = create_test_datapoint_value(client, metadata_reference)
packet_id = client.storeData(datapoint_value)
```

Storing a list of data packets

To store a list of data packets, we first create a list of DataPacket objects representing data point values. Then we call the storeDataList method of the TelemetryArchiveApi interface. It returns a list of strings specifying the assigned IDs of the data packets in the Engineering Archive.

Listing 5-19, Listing 5-20 and Listing 5-21 contain a simple client that stores a list of data packets in the Java, CPP and Python programming languages respectively.

Java client

Listing 5-19: Example of storing a list of data packets in Java

```
List<DataPacket> dataPackets = new ArrayList<>();
metadata = TelemetryArchiveAPIExample.createTestMetadata(mal);
metadataReference = client.storeData(metadata);
for (int i = 0; i < 2; i++) {
    dataPointValue = TelemetryArchiveAPIExample.
↪createTestDataPointValue(metadataReference);
    dataPackets.add(dataPointValue);
}
List<String> ids = client.storeDataList(dataPackets);
```

CPP client

Listing 5-20: Example of storing a list of data packets in CPP

```
std::vector<std::shared_ptr<elt::telemetry::archive::DataPacket>> dataPackets;
std::shared_ptr<elt::telemetry::archive::DataPacket> metadataInstance =␣
↪createTestMetadata(client);
std::string metadataRef = client->storeData(metadataInstance);
for (std::size_t i = 0; i < 2; i++) {
  std::shared_ptr<elt::telemetry::archive::DataPacket> datapoint =␣
↪createTestDatapointValue(client, metadataRef);
  dataPackets.push_back(datapoint);
}
std::vector<std::string> ids = client->storeDataList(dataPackets);
```

Python Client

Listing 5-21: Example of storing a list of data packets in Python

```
data_packets = []
metadata = create_test_metadata(client)
metadata_reference = client.storeData(metadata)
for i in range(2):
    datapoint_value = create_test_datapoint_value(client, metadata_reference)
    data_packets.append(datapoint_value)
ids = client.storeDataList(data_packets)
```

Querying data packets

To query data packets in the Engineering Archive we call the queryData method of the TelemetryArchiveApi interface. It takes a string representing an ES Query DSL [3] query and returns a list of strings specifying the assigned IDs of the data packets in the Engineering Archive. In the provided example, we query all data packets that have the *data.metadataReference* field set to the ID of the data packet containing metadata that was stored last. When querying the EA using the ES Query DSL, the current limitation is that it currently does not support sorting the results.

Note: When the Elasticsearch is used as the Engineering Archive, some time needs to elapse between storing the data and querying it. This time can be up to 1 second depending on the settings of the Elasticsearch.

Listing 5-22, Listing 5-23 and Listing 5-24 contain a sample Java client that queries the Archive API for a data packet.

Java client

Listing 5-22: Example of querying data packets in Java

```
List<String> queriedIDs = client.queryData("data.metadataReference:" +␣
↪metadataReference);
```

CPP client

Listing 5-23: Example of querying data packets in CPP

```
std::stringstream query;
query << "data.metadataReference:" << metadataReference;
std::vector<std::string> queriedIds = client->queryData(query.str());
```

Python client

Listing 5-24: Example of querying data packets in Python

```
queried_ids = client.queryData("data.metadataReference:%s" % (metadata_reference,
↪))
```

If the search query contains Elasticsearch reserved characters such as + - = && || > < ! ( ) { } [ ] ^ " ~ * ? : \ /., they have to be escaped e.g: \. Additional resources on Elasticsearch queries: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html

Additional Query Examples

Querrying data for a specific data point in a specified time interval (please note the quotes around the data point URI). When querying for subscribed URIs the URI protocol should be set to cii.oldb. When querying for data stored via Archiver API, the client might have used a different protocol:

```
data.uri:"cii.oldb:///trk/ctrl/my_data_point" AND data.timestamp:(>=␣
↪1593786692807000 AND <= 1593786892807000)
```

Searching for data point with an exact data point value

```
data.uri:"cii.oldb:///trk/ctrl/my_data_point" AND data.value:0.0
```

Searching for all data points with a quality flag set to BAD in specified time interval

```
data.quality:"BAD" AND data.timestamp:(>= 1593786692807000 AND <=␣
↪1593786892807000)
```

Searching for all data points with a quality flag set to BAD or OK and data point value >= 1.0

```
(data.quality:"BAD" OR data.quality:"OK") AND data.value:>=1.0
```

Searching for all data points uris that start with a specific path:

```
data.uri:"cii.oldb:///trk/ctrl/*"
```

The limitation in this case is that one cannot use wildcard in the middle of the path (such as: cii.oldb:///trk/c* as Elasticsearch splits the term into subpaths by splitting on the '/' character).

Downloading data packets

To download data packets from the Engineering Archive we call the downloadData method of the TelemetryArchiveApi interface. It takes a list of strings representing the IDs of the data packets and returns a list of DataPacket objects. In the provided example, we download the data packets specified by the IDs that we acquired above.

There exists a separate API intended for downloading big data. The examples for big data can be found in section 5.3.

Java client

The following sample code demonstrates how to download data packets. The code is an excerpt from the code file named TelemetryArchiveAPIExample.java stored in the java/sample-app folder.

Listing 5-25: Example of downloading data packets in Java

```
List<DataPacket> downloadedDataPackets = client.downloadData(queriedIDs);
downloadedDataPackets.forEach(TelemetryArchiveAPIExample::displayDataPacket);
```

The static method TelemetryArchiveAPIExample.displayDataPackets takes care of printing the downloaded data packets to the standard output.

Listing 5-26: Example of displaying data packet data in Java

```
public static void displayDataPacket(DataPacket dataPacket) {

    Map<String, Object> map = new HashMap<>();

    map.put("archived", dataPacket.getArchived());
    map.put("bigDataLocation", dataPacket.getBigDataLocation());
    map.put("data", dataPacket.getData());

    JSONObject json = new JSONObject(map);
    System.out.println("\t" + json);
}
```

CPP client

In a similar manner to the example using Java, the packets can be downloaded in CPP as follows:

Listing 5-27: Example of downloading data packets in CPP

```
std::vector<std::shared_ptr<::elt::telemetry::archive::DataPacket>>␣
↪downloadedDataPackets = client->downloadData(queriedIds);
for (auto &packet : downloadedDataPackets) {
  displayDataPacket(packet);
}
```

The data packets can be displayed using the following function:

Listing 5-28: Example of displaying data packet data in CPP

```
void displayDataPacket(std::shared_ptr<elt::telemetry::archive::DataPacket> &
↪dataPacket) {
  std::cout << "... downloaded data packet: " << std::endl;
  std::cout << "    archived: " << dataPacket->getArchived() << std::endl;
  std::cout << "    bigDataLocation: " << dataPacket->getBigDataLocation() <<␣
↪std::endl;
  std::cout << "    data: " << dataPacket->getData() << std::endl;
}
```

Python client

In a similar manner to the example using Java, the packets can be downloaded in Python as follows:

Listing 5-29: Example of downloading data packets in Python

```
downloaded_data_packets = client.downloadData(queried_ids)
_ = [display_data_packet(data_packet) for data_packet in downloaded_data_packets]
```

The data packets can be displayed using the following function:

Listing 5-30: Example of displaying data packet data in Python

```
def display_data_packet(data_packet):
    print(' '*10, 'Data packet:')
    print(' '*15, 'archived: ', data_packet.getArchived())
    print(' '*15, 'bigDataLocation: ', data_packet.getBigDataLocation())
    print(' '*15, 'data: ', data_packet.getData())
    print()
```

Data packet deserialization

The following examples are presenting one way of deserializing data packet data field that contains the serialized OLDB data point.

Deserializing in Java

First, we have to define the deserialization structure (getters and setters are omitted for clarity):

```
static class ExampleDataPoint {
    private URI uri;
    private String metadataReference;
    private String quality;
```

(continues on next page)

(continued from previous page)

```
    private double value;
    private double timestamp;
}
```

Once we have the structure defined, we can deserialize the data packet:

Listing 5-31: Example of deserializing data packet containing data point in Java

```
ObjectMapper mapper = new ObjectMapper();
```

```
double archived = packet.getArchived();
```

```
String bigDataLocation = packet.getBigDataLocation();
```

```
ExampleDataPoint dataPoint = mapper.readValue(packet.getData(), ExampleDataPoint.
↪class)
```

Deserializing in CPP

First, we have to define the deserialization structure:

```
// structure that was stored in the data field
struct ExampleDataPoint {
  std::string uri;
  std::string metadataReference;
  std::string quality;
  double value;
  double timestamp;

  // Serialize itself to json string
  std::string serialize() const {
    boost::property_tree::ptree tree;
    tree.put("uri", uri);
    tree.put("metadataReference", metadataReference);
    tree.put("quality", quality);
    tree.put("value", value);
    tree.put("timestamp", timestamp);
    std::stringstream data;
    boost::property_tree::json_parser::write_json(data, tree);
    return data.str();
  }

  // Method to construct ExampleDataPoint from json string
  static ExampleDataPoint deserialize(const std::string &json) {
    boost::property_tree::ptree tree;
    // Parse json and store its contents in the property tree
```

(continues on next page)

(continued from previous page)

```
    std::istringstream stream(json);
    boost::property_tree::json_parser::read_json(stream, tree);
    // Move the values from the tree into new ExampleDataPoint instance
    ExampleDataPoint newInstance;
    newInstance.uri = tree.get<std::string>("uri");
    newInstance.metadataReference = tree.get<std::string>("metadataReference");
    newInstance.quality = tree.get<std::string>("quality");
    newInstance.value = tree.get("value", 0.0);
    newInstance.timestamp = tree.get("timestamp", 0.0);
    return newInstance;
  }
}
```

Once we have the deserialization structure defined, we can deserialize the data field:

Listing 5-32: Example of deserializing data packet containing data point in CPP

```
// deserialize the downloaded data packet
double archived = packet->getArchived();
std::string bigDataLocation = packet->getBigDataLocation();
ExampleDataPoint dataPoint = ExampleDataPoint::deserialize(packet->getData());
```

Deserializing in Python

Listing 5-33: Example of deserializing data packet containing data point in Python

```
archived = data_packet.getArchived()
bigDataLocation = data_packet.getBigDataLocation()
data = json.loads(data_packet.getData())
for key, value in data.items():
      print(key, '=', value)
```

Closing the MAL connection

Once we are finished storing, querying, and downloading data packets, we should close the MAL and CII Config client connection by calling the close method.

Closing the connection in Java

The close method needs to be called on both on the TelemetryArchiveAPISync object client as well as the CiiFactory object factory.

```
client.close();
factory.close();
```

Additionally, both objects are also auto-closeable, so they can be used in try-with-resources statements if that is practical for your application.

Closing the connection in CPP

In CPP the connections can be closed as follows:

```
client->close();
```

Closing the connection in Python

In Python the connections can be closed as follows:

```
client.close()
```

11.5.3 Archive API: Big Data Storage

Normally the Engineering Archive is agnostic of the underlying databases used for storage. However, to have higher performance, the Hadoop storage must be accessed directly when working with big data. The provided example uses an agnostic Big Data interface, but a specific implementation of that interface for Hadoop.

An example of an application that uses Big Data Storage library can be found in the telemetry-examples project. The examples project shows the usage of the Big Data Storage in the:

- *HadoopStorageExample.java* file in the java/sample-app directory for Java,
- *app.cpp* file in the cpp/storage directory for CPP, and the
- *cii-telemetry-examples-storage-py.py* file in the python/storage directory for Python.

The examples can also be found in Appendix D.

A short description of the part of the Archive API used for writing, reading and deleting big data can be found in Appendix A.2.

Imports/Includes

Usage of the HadoopStorage library requires importing the Hadoop Storage class that implements the methods of the BigDataStorage interface.

Java

```
import java.net.URI;
import elt.storage.HadoopStorage;
import elt.storage.BigDataStorage;
```

CPP

```
#include <ciiHadoopStorage.hpp>
#include <ciiException.hpp>
```

Python

```
import elt.storage
```

Creating the client instance

First, we need to create an instance of the Hadoop Storage class by providing the URI of the Hadoop database of the Engineering Archive.

Java example

The following listing shows how to create an instance of the Big Data Storage in Java. The sample code is an excerpt from the code file named TelemetryArchiveAPIExample.java stored in the java/sample-app folder.

Listing 5-34: Example of Big Data Storage client creation in Java

```
final BigDataStorage hadoopStorage =
        new HadoopStorage(new URI("http://ciihdfshost:9870"), "/esoLs");
// contains path to the archive
final URI uri = new URI("cii.config://*/archive");
```

CPP example

The following listing shows how to create an instance of the Big Data Storage in CPP. The sample code is an excerpt from the code file named app.cpp stored in the cpp/storage folder. The code follows the same flow as the Java code.

Listing 5-35: Example of Big Data Storage client creation in CPP

```
const elt::mal::Uri hadoopLocation("http://ciihdfshost:9870");
const std::string hadoopEndpoint("/esoSample");
elt::storage::CiiHadoopStorage storage(hadoopLocation, hadoopEndpoint);
```

Python example

The following listing shows how to create an instance of the Big Data Storage in Python. The sample code is an excerpt from the code file named cii-telemetry-examples-storage-py.py stored in the python/storage folder. The code follows the same flow as the Java code.

Listing 5-36: Example of Big Data Storage client creation in Python

```
hadoop_location = 'http://ciihdfshost:9870'
hadoop_endpoint = 'esoSample'
storage = elt.storage.CiiHadoopStorage(hadoop_location, hadoop_endpoint)
```

Writing data

To write data to the big data database of the Engineering Archive we call the write method of the HadoopStorage class. It takes the big data archive URI and the data as parameters. Once the big data file is stored, the write method returns the automatically generated UUID file name.

If one would like to store the big data file reference into the Engineering Archive, one has to create the data packet (see section 5.2.3), set created big data file reference into the bigDataLocation data packet field and store the data packet into the Engineering Archive as described in section 5.2.3.

Java example

The following listing shows how to write data into the Big Data Storage in Java. The sample code is an excerpt from the code file named TelemetryArchiveAPIExample.java stored in the java/sample-app folder.

Listing 5-37: Example of writing data to the Big Data Storage in Java

```
byte[] data = "data".getBytes();
String filename = hadoopStorage.write(uri, data);
```

CPP example

The following listing shows how to write data into the Big Data Storage in CPP. The sample code is an excerpt from the code file named app.cpp stored in the cpp/storage folder. The code follows the same flow as the Java code.

Listing 5-38: Example of writing data to the Big Data Storage in CPP

```
const std::string remoteDirectory = "archive";
const std::string remoteFilename = "file.bin";
const std::string contentToStore(generateContent());
std::stringstream stream(contentToStore);
storage.write(remoteDirectory, remoteFilename, stream);
```

Python example

The following listing shows how to write data into the Big Data Storage in Python. The sample code is an excerpt from the code file named cii-telemetry-examples-storage-py.py stored in the python/storage folder. The code follows the same flow as the Java code.

Listing 5-39: Example of writing data to the Big Data Storage in Python

```
remote_directory = 'archive'
remote_filename = 'file.bin'
content_to_store = generate_content()
stream = io.BytesIO(content_to_store.encode('utf-8'))
storage.write_from(stream, remote_directory, remote_filename)
```

Finding UUIDs of Archived Big Data Files

The big data file could be stored in the big data database either through OLDB-TA subscription mechanism or through the client code (see section 5.3.3). At some point after storing the data in the big data database, one might want to read that file for further analysis. In order to do that a UUID of the big data file must be known.

There are two ways to get the UUID:

- When you store the big data file via Big Data Storage library, an UUID of the file is returned which you can use for further actions.
- When you are trying to retrieve a historic big data file (data that was archived some time ago), you have to find it by querying the Engineering Archive. Big data files are usually associated with other data points values/metadata, so you have to form your query according to the data point you are interested in; see section 5.2.5 on how to query the data. From the queried data you have to extract the big data file reference (UUID that you are looking for) and use that UUID for downloading or deleting big data file.

Note on extracting: if you are using the Archiving API for querying, the API will return you all the data packets relevant to your query. To get the big data file UUIDs you have to iterate over received data packets and call getBigDataLocation() method.

Reading data

To read the data from the big data database of the Engineering Archive we call the read method of the HadoopStorage class. It takes the big data archive URI and the UUID file name as parameters, and returns data encoded into a sequence of bytes. In the provided example we read the data that we have previously written. If one would like to find the big data file UUID that was archived in the past, see section 5.3.4 on how to do that.

Listing 5-40, Listing 5-41, and Listing 5-42 show examples of reading data from the Big Data Storage in Java, CPP, and Python respectively.

Java example

Listing 5-40: Example of reading data from the Big Data Storage in Java

```
byte[] readDataBytes = hadoopStorage.read(uri, filename);
```

CPP example

Listing 5-41: Example of reading data from the Big Data Storage in CPP

```
std::ostringstream ostream;
storage.read(remoteDirectory, remoteFilename, ostream);
```

Python example

Listing 5-42: Example of reading data from the Big Data Storage in Python

```
ostream = io.BytesIO()
storage.read_into(ostream, remote_directory, remote_filename)
```

Deleting files

To delete a file from the big data database of the Engineering Archive we call the delete method of the HadoopStorage class. It takes the big data archive URI and the UUID file name as parameters. In the provided example we delete the file containing the data that we have previously written and read. If one would like to find the big data file UUID that was archived in the past, see section 5.3.4 on how to do that.

Listing 5-43, Listing 5-44 and Listing 5-45 show examples of deleting data from the Big Data Storage in Java, CPP and Python respectively.

Java example

Listing 5-43: Example of deleting files in the Big Data Storage in Java

```
hadoopStorage.delete(uri, filename);
```

CPP example

Listing 5-44: Example of deleting files in the Big Data Storage in CPP

```
storage.remove(remoteDirectory, remoteFilename);
```

Python example

Listing 5-45: Example of deleting files in the Big Data Storage in Python

```
storage.remove(remote_directory, remote_filename)
```

11.5.4 Telemetry Archiver Management API

After changing configurations on the Configuration service, the Service Management API must be used to refresh the configuration on the Telemetry Archiver.

A short description of the part of the Telemetry Archiver Management API can be found in A.3.

An example of an application that uses the Telemetry Archiver Management API to refresh a configuration can be found in the *TelemetryServiceManagementAPIExample.java* file in the telemetry-examples project.

Imports/includes

Usage of the Telemetry Archiver Management API requires the following imports to establish a MAL connection: CiiFactory, QoS, ReplyTime, and ZpbMal. The TelemetryServiceManagementSync interface defines Telemetry Archiver Management methods.

Java

```
import java.net.URI;
import elt.mal.CiiFactory;
import elt.mal.rr.qos.QoS;
import elt.mal.rr.qos.ReplyTime;
import elt.mal.zpb.ZpbMal;
import elt.telemetry.management.TelemetryServiceManagementSync;
```

CPP

```
#include <mal/Cii.hpp>
#include <mal/utility/LoadMal.hpp>
#include <mal/rr/qos/ReplyTime.hpp>
#include <CiiTelemetryManagement.hpp>
```

Python

```
import elt.pymal as mal
from ModCiiTelemetryManagement.Elt.Telemetry.Management.
↪TelemetryServiceManagement import TelemetryServiceManagementSync
```

Connection to Telemetry Archiver using MAL and closing the connection

The connection can be created in the same way as described in section 5.2.2. The MAL connection needs to be closed as shown in section 5.2.8.

Refreshing configurations

To refresh selected configurations, we call the refreshConfigurations method of the TelemetryServiceManagement interface. It takes a list of strings specifying the URIs of the configurations that we want to refresh as parameter. The URIs of the relevant data capture configurations could be obtained:

- Via configuration GUI, for more information see section 4.2.
- When the data capture configuration is saved in the config-service: in order to save the data capture configuration in the config-service, the location/URI where the data capture configuration will be stored must be specified. In order to inform the Telemetry Archiver of the new/updated configuration one has to call refreshConfigurations method with the same URI.
- Programmatically by querying the config-service for the relevant data capture configuration. An example of programmatic querying in Java for all data captures stored under Telemetry Archiver's service range could be seen below. For more info on how to query the config-service in other languages, see [1].

```
CiiConfigClient configClient = CiiConfigClient.getInstance();
String archiverServiceRange = ElasticSearchUtilities.createElasticSearchId(
  URI.create("cii.config://*/telemetry/service1/datacapture/"));
List<String> dataCaptureConfigIds =
  configClient.remoteIndexSearch("configuration_instance",
  String.format("id:%s*", archiverServiceRange));

// dataCaptureConfigIds could now be as an input for refreshConfigurations
// method
```

In order to refresh all the configurations, we call the refreshAllConfigurations method of the TelemetryServiceManagement interface.

Listing 5-46, Listing 5-47 and Listing 5-48 show examples refreshing configurations in Java, CPP and Python respectively. Note that the example code assumes that a configuration with the URL *"cii.config:///*telemetry/service1/48fe831c-436a-401b-ab27-18ee401735af"* already exists.

Java example

Listing 5-46: Example of refreshing configurations in Java

```
// refresh selected configurations
List<String> ids = new ArrayList<>();
ids.add("cii.config://*/telemetry/service1/48fe831c-436a-401b-ab27-18ee401735af
 ↪");
client.refreshConfigurations(ids);

// refresh all configurations
client.refreshAllConfigurations();
```

CPP example

Listing 5-47: Example of refreshing configurations in CPP

```
// Refresh selected configurations
std::vector<std::string> dataCaptureIds
 {"cii.config://*/telemetry/service1/48fe831c-436a-401b-ab27-18ee401735af"};
client->refreshConfigurations(dataCaptureIds);
// Refresh all configurations
client->refreshAllConfigurations();
```

Python example

Listing 5-48: Example of refreshing configurations in Python

```
# Refresh selected configurations
ids = ['cii.config://*/telemetry/service1/48fe831c-436a-401b-ab27-18ee401735af']
client.refreshConfigurations(ids)

# Refresh all configurations
client.refreshAllConfigurations()
```

11.6 Telemetry CLI tools

This section describes how to use Telemetry Archiver CLI tools. These command line tools simplify basic operations without needing to write custom applications for these tasks.

11.6.1 CLI Manager

CLI Manager allows a user to send the Telemetry Archiver a notification to trigger a refresh of its configurations.

The default syntax for the CLI Manager is:

```
telemetry-manager refresh [-c <config> <config>] [-h] [-s <service endpoint>]
```

The -h argument prints a help description to the standard output.

The refresh command options are:

```
-c,--configurations <config> <config>    specifies list of configuration
                                         ids, based on which telemetry
                                         service configurations are
                                         refreshed. If this option is
                                         omitted, all configurations are
                                         refreshed
 -h,--help                               displays this help message
 -s,--service <service endpoint>         specifies custom service endpoint
                                         of telemetry archiver for manager
                                         to connect to.
```

Usage example

```
$ telemetry-manager refresh -c config_1 config_2
```

Here only specific configurations are refreshed (configurations with ids config_1 and config_2).

```
$ telemetry-manager refresh
```

Here all existing configurations are refreshed.

11.6.2 CLI Archiver

CLI Archiver is a tool enabling users to store, download, and query data packets from and to the Engineering Archive. The CLI Archiver is not optimized for speed and is meant to be used for one-time slow archive related tasks. A single instance of the telemetry-tool doesn't support full required rate of telemetry service.

The telemetry-tool provides the user with three different commands: store, download, and query. All commands have specific options available, further specifying the behavior of the command. The help

and service options can be used with all commands. The help option is used for display the help message. The service option enables the user to specify a custom address of the Telemetry Archiver.

The default syntax for CLI archiver is:

```
telemetry-tool [command] [-h | specific options] | [-h] [command]
```

The -h argument prints a help description to the standard output.

Each command has a list of options that can be used with it. All options can be viewed with the -h option.

Storing data packets

Storing of data packages is possible by using the store command. Additionally, by using different provided options the user can store different files (data packets containing metadata and/or big binary files), as well as specifying different service endpoints for the archiver to connect to (service endpoint option).

The Store command syntax is as follows:

```
telemetry-tool store [-b <file path> <file path>] -d <file path> <file path> | -
↪h [-s <service endpoint>]
```

The Store command options are:

```
options:
 -b,--bigdata <file path> <file path> specifies file path to
                                       bigData binary file to be stored
 -d,--data <file path> <file path>    specifies file path that
                                       contains data for creation of new
                                       dataPacket
 -h,--help                             displays help message
 -s,--service <service endpoint>       specifies custom service endpoint
                                       of telemetry archiver to connect to.
```

Multiple data packets can be created and stored with a single store command. If multiple file paths are provided as parameters to the option. The bigdata option is optional, but can also specify multiple file paths. Additionally, specified big data file paths correspond to their respective data packets provided with the data option (this means, that not all data packets necessarily have to contain corresponding big data files attached to them and that both types of packets are paired based on the order they were passed as parameters: 1 data packet goes with 1 big data file, etc.).

The data file is meant to be any kind of JSON based file up to 2 GB in size. While the data file does not have to follow the data packet structure (any valid json will do), it is highly recommended that it follows it (the data packet structure), otherwise analyzing and operating on such highly dynamic data will be extremely problematic.

The bigdata file could be anything (text file, image, video, compiled program) and doesn't have a file size limit.

Storing metadata

The example below shows how to store the data point metadata into the Engineering Archive:

```
$ telemetry-tool store -d metaFile1.json
mcx7jHQBRBKWEuv9m9Gc
```

The output from the telemetry tool represents the stored metadata data packet ID (mcx7jHQBRBKWEuv9m9Gc), which the user could use later on to download the archived metadata or reference the metadata when archiving data point values (see section 6.2.1.2). The metaFile1.json is following the structure of the CII Config metadata and it is displayed below:

Listing 6-1: an example of metaFile1.json file

```
{
  "@type": "MdOldbNumber",
  "value": 2.0,
  "comment": "",
  "checked": false,
  "metadataInstanceVersion": 1,
  "@name": "telemetry_meta1",
  "@genType": "SINGLE",
  "metadataInstance": "telemetry_meta1"
}
```

Storing data point values

The data point values are stored via the following command:

```
$telemetry-tool store -d data/dpValue1.json data/dpValue2.json -b BigPicture.jpg
j_HWn3ABUTxPCroFrXwo n8yTjHQBRBKWEuv9ltGT
```

The dataPacket1.json and dataPacket2.json are files that should follow the structure below (the structure is not enforced and the end user can archive any valid JSON):

Listing 6-2: structure of the dpValue1.json file

```
{
  "uri": "cii.client:///my_datapoint_id_1",
  "metadataReference": "mcx7jHQBRBKWEuv9m9Gc",
  "quality": "OK",
  "value": 0.5,
  "timestamp": 1600084745793000
}
```

Before storing the data point value (e.g., dpValue1.json) into the Engineering Archive, make sure that the metadata in the metadataReference field is already present in the Engineering Archive. The metadata reference can be obtained by either searching in the Engineering Archive (see section 6.2.3)

or storing the metadata first (see section 6.2.1.1) and then storing the data packet with the reference of the archived metadata.

If the -b (–bigdata) flag is present, the referenced big data file is stored to the Big Data Storage and its ID is stored to corresponding data packet's bigDataLocation field. Big data files are paired with data packets in order they are provided. In the provided example above, the dpValue1.json data packet has a reference to big data file attached (reference to archived BigPicture.jpg), while the dpValue2.json data packet does not have a reference to big data file.

The telemetry tool returns the IDs of the archived data point data packets, which are used for downloading the archived data packets from the Engineering Archive (see section 6.2.2).

Downloading data packets

Data packets and their corresponding big data files can be retrieved from the Engineering Archive using the download command. Data packets can be either downloaded to local disk or displayed based on options provided.

The Download command syntax is as follows:

```
telemetry-tool download -h | -p <packet id> <packet id> [-o <file path>]
[-s <service endpoint>]
```

The Download command options are:

```
options:
 -h,--help                                  displays help message
 -o,--output <folder path>                    specifies the path to folder where
                                            data packets will be stored. If
                                            this option is ommited, downloaded
                                            packets data is displayed instead.
 -p,--packets <packet id> <packet id>       specifies list of packet IDs that
                                            will be downloaded
 -s,--service <service endpoint>            specifies custom service endpoint
                                            of telemetry archiver to connect
                                            to.
```

Multiple packets can be displayed/downloaded using a single download command. The output option (-o,–output) controls if the data packets will be displayed or downloaded. This option is not required, so by default, the specified data packets will only be displayed to the user. If this option is present, packets will be stored inside the folder denoted by this option's parameter, specifying the location on the local disk where the download folder will be created. If the data packet that should be downloaded contains the location (URI denoting the location in the EA) of the big data file, that file is also retrieved and stored in the same folder.

Below two examples show the usage of both modes (display and download) for the download command.

Usage example

Displaying data packet downloaded from the Engineering Archive:

```
$ telemetry-tool download -p j_HWn3ABUTxPCroFrXwo
{"archived":1.583229611294E15,
 "data":"{\"key\":\"testValue1\"}",
 "bigDataLocation":"cii.config:\/\/*\/archive\/0a383f2f-8b37-4e11-b6c5-
↪1d798e278da0"}
```

Here the specified data packet is retrieved from the EA and its content is displayed. It is not stored on the local disk.

Usage example

Downloading a data packet from Engineering Archive to local disk:

```
$ telemetry-tool download -p j_HWn3ABUTxPCroFrXwo -o ~/downloadFolder
```

Here a new folder named downloadFolder is created inside the user's home directory. A new file is created inside containing the information stored in the data field of the downloaded data packet. If the packet has a big data file location defined, another binary file will be created. The data packet ID also specifies the name of the newly created file, where the big data binary file additionally has .bin appended to the end.

Querying data packets

The user can search for specific data packets using the query command. The ES Query DSL [3] is used for querying, allowing the user to find packets matching certain criteria specified by the query request (e.g. all packets having certain field set to specified value). The result of the query command is a list of IDs representing data packets that match the sent query.

The Query command syntax is as follows:

```
telemetry-tool query -h | -r <request definition>  [-s <service endpoint>]
```

The Query command options are:

```
options:
 -h,--help                          displays help message
 -r,--request <request definition>  Elasticsearch query based on which
                                    the returned packet IDs are selected
 -s,--service <service endpoint>    specifies a custom service endpoint of
                                    the telemetry archiver to connect to.
```

When specifying options, either request option (-r,–request) or help options (-h,–help) must be provided. The example below displays the usage of the query command.

Usage example

```
$ telemetry-tool query -r "data.metadataInstance: telemetry* AND data.value: 2"
j_HWn3ABUTxPCroFrXwo jvHWn3ABUTxPCroFrXwo
```

Here query requests data packets where the *data.value* field equals *2* and *data.metadataInstance* matches all *telemetry** entries. As a result, IDs of two data packets (j_HWn3ABUTxPCroFrXwo and jvHWn3ABUTxPCroFrXwo in the above example), that match the set query are returned.

11.7 Telemetry Subscription

This section describes how to correctly prepare a Telemetry Archiver (TA) for archiving data point changes that are published via OLDB subscription. Telemetry Archiver subscribes to the data point changes depending on the data capture configurations which are stored in the config-service.

Each data capture configuration represents a set of rules that determine whether or not a specific data point change should be captured. It is possible to define multiple data capture configurations for one data point in order to make a more fine-grained archiving behaviour.

Each Telemetry Archiver has its own specified range that defines the location in the Configuration service from which the data capture configurations are being downloaded from. An example on how this works is shown below:

Say we have a Telemetry Archiver (TA1) and we would like it to archive data point (DP1) value changes. In order to do that we have to:

1. Create a data capture configuration (CiiDataCaptureConfig) with desired archiving rules set (mentioned below in the examples section).
2. Store data capture configuration into config-service under Telemetry Archiver's service range.
3. Reload the data capture configurations of TA1 by calling one of the refreshConfiguration methods on the TA1's Service Management API that will seamlessly reload the in-memory capture configurations with the new ones coming from the config-service.

The data capture configuration fields are documented in Appendix B.

The data capture configurations could be saved/updated with both the config-client and config-gui. See provided manuals [1] on how to effectively use config-client or config-gui. An example written in the Java programming language is shown below (similarly one can do the same via C++ or Python):

Listing 7-1: Example of creating and saving a data capture configuration

```
final URI serviceRange = URI.create("cii.config;//*/telemetry/service1");
final URI oldbDatapointUri = URI.create("cii.oldb:///datapoint_1");
DataCaptureConfiguration captureConfig = new DataCaptureConfiguration(
    oldbDatapointUri, // oldb datapoint uri
    1.0f,           // delta value change
    "ABSOLUTE",     // delta type set to absolute changes
    "VALUE",        // archive mode set to value changes
    "00:01",        // minimum interval (at most once per second)
    "10:00",        // maximum interval (at least once every 10 minutes)
    true            // archiving enabled
);
// data capture configuration id should be unique
captureConfig.setId(UUID.randomUUID().toString());


// save data capture configuration to config-service
try {
```

(continues on next page)

(continued from previous page)

```
    CiiConfigClient client = CiiConfigClient.getInstance();
    client.saveTargetConfig(serviceRange, captureConfig);
} catch (CiiSerializableException e) {
    System.out.println("Error while storing data capture configuration: "
                       + e.getCiiMessage());
}
```

11.7.1 Data Capture Configuration Examples

We assume the Telemetry Archiver (TA1) which will subscribe to data point (DP1) changes with a service range set to cii.config://*/telemetry/archiver1:

Archive data point on absolute value change or quality change

In this example we would like to archive data point (DP1) in the following situation:

- Data point should be archived on absolute value change (+/- 2.0)
- Data point should be archived at least once every 5 minutes
- Data point should be archived at most every second
- Data point should be archived if the quality flag has changed

We have to create 2 capture configurations:

1. Data capture (DC1) that will define the rules for archiving absolute value change. The data capture configuration should have the following fields set:

Field	Value
dataPointAddress	cii.oldb:///datapoints/dp1
archiveMode	VALUE
deltaType	ABSOLUTE
deltaValue	2.0
minimumInterval	1
maximumInterval	05:00
archivingEnabled	true

2. Data capture (DC2) that will define the rules for archiving quality change:

Field	Value
dataPointAddress	cii.oldb:///datapoints/dp1
archiveMode	QUALITY
deltaType	“”
deltaValue	0
minimumInterval	0 (minimum interval when capturing quality changes is not used)
maximumInterval	0 (maximum interval for quality change can be set as well, but in this example this is already handled in the DC1 config example)
archivingEnabled	true

Archive data point on relative value change or metadata change

We would like to archive the data point based on the following rules:

- Data point should be archived if the value changes by 4% of the current value
- Data point should be archived if the metadata has changed

Again, we create two data capture configurations:

1. Data capture configuration (DC1) that will define the rules for relative value change. The process is the same as in section 7.1.1, except the delta type field should be set to “RELATIVE”.

Field	Value
dataPointAddress	cii.oldb:///datapoints/dp1
archiveMode	VALUE
deltaType	RELATIVE
deltaValue	0.04
minimumInterval	0 (all updates will be stored)
maximumInterval	0 (maximum interval is not set)
archivingEnabled	true

2. Data capture configuration (DC2) that will define the rule for metadata change. The data capture configuration that handles metadata changes does not respect the set minimum and maximum interval and will archive the data point on every metadata change. It is expected that metadata will be rarely changed.

Field	Value
dataPointAddress	cii.oldb:///datapoints/dp1
archiveMode	METADATA
deltaType	“” (irrelevant for metadata change)
deltaValue	0 (irrelevant for metadata change)
minimumInterval	0 (minimum interval is not respected; metadata change is always archived)
maximumInterval	0 (maximum interval is not set)
archivingEnabled	true

Stop archiving the data point

There are two option for TA1 to stop archiving changes of the specified data point (DP1):

1. Set the archivingEnabled field to false for all data capture configurations that are referencing a DP1 (via dataPointAddress field) and call one of the refreshConfiguration methods on the TA1’s Service Management API.
2. Delete the data capture configuration referencing a DP1 (via dataPointAddress field) from the configuration service and call one of the refreshConfiguration methods on the TA1’s Service Management API.

11.7.2 OLDB Simulation

For testing purposes of the Telemetry Archiver, it might be useful to simulate the data point value updates. This could be done by:

- Using OLDB mock.
- Using OLDB simulator.

OLDB Mock

For fine grained testing purposes, it might be useful to use your own OLDB mock service instead of the production OLDB service. Internally the Telemetry Archiver is using OLDB client, which subscribes to the OLDB service for data point value updates. Consult the 3.3.1 OLDB Client Configuration section of CII Online Database user manual [6], on how to configure the OLDB client to connect to your own OLDB mock service which you are going to use as part of the testing purposes.

OLDB Simulator

For performance testing purposes it might be useful to use the OLDB simulator which allows you to continuously simulate data point value updates depending on the data capture configurations that are stored in the CII Configuration Service. The OLDB data point value simulation works in the following fashion:

- The user starts the Telemetry Archiver with the TA's simulateCapture configuration field set to true.
- The OLDB simulator that is started within the Telemetry Archiver downloads the data capture configurations from the CII Config Service from the serviceRange specified in the TA's configuration.
- The data point info is parsed from the downloaded data capture configurations. The URI from the data capture configuration defines the URI of the data point for which the updates will be simulated.
- The OLDB simulator starts generating data point updates based on parsed data point info.

The OLDB simulator is currently supporting 2 archive modes from the data capture configuration:

- Value change
- Quality change

If the data capture configuration archiveMode is set to VALUE_CHANGE, the ABSOLUTE value of the data point will be continuously generated with the minimum interval based the data capture configuration deltaChange field (delta change represents the value change necessary for data point to be archived). This ensures that every generated data point value is suitable for archiving. The data point RELATIVE value updates are currently not supported by the simulator.

If the data capture configuration archiveMode is set to QUALITY_CHANGE, the quality flag of the data point will be flipped (from true to false and vice versa) with the minimum interval period as defined in the data capture configuration.

The URIs in the data capture configurations could be anything as long as they follow the OLDB URI convention (cii.oldb:///uri), however the OLDB simulator supports a special behavior in case the data point URI is one of the following:

Datapoint URI	Behavior
cii.oldb:///simulated_datapoint_1	Generate data point updates based on the data capture configuration, without any side effects. This is the regular behaviour of the OLDB simulator.
cii.oldb:///simulated_datapoint_2	Generate data point updates based on the data capture configuration, but omit the data point timestamp field (timestamp field of the generated data point is null).
cii.oldb:///simulated_datapoint_3	Generate data point updates based on the data capture configuration, but omit the data point quality flag field (quality flag of the generated data point is null).

11.8 Telemetry Archiver Deployment

Telemetry Archivers (TA) are designed to be horizontally scalable. In order to achieve this, they do not know about one another and the end user has to manually specify which TA will be archiving which data points under which data capture rules.

This data point archiving specification is done through the Telemetry Archiver's configuration (see Appendix C). Every Telemetry Archiver has to have its own configuration file, because the Telemetry Archiver configurations have to differ in at least two configuration fields (both fields have to be unique when running two services on the same machine):

- **telemetryServiceURI**: location that defines the network interface and port on which the TA will be bound to (e.g. zpb.rr://0.0.0.0:9115/).
- **serviceRange**: location in configuration service where the specific Archiver's data capture configurations are located (e.g: cii.config//*/telemetry1/archiver1/)

Each Telemetry Archiver contains its configuration stored in the CII Configuration Service. The configuration file itself could be deployed in either local or remote configuration service database. The location of the configuration file is determined by either command line flag or environment variable (TELEMETRY_CONFIG). The variable should be following the CII Configuration Service URI rules (cii.config//location/path/to/config).

The order of configuration file location is the following:

1. Command line flag (-DTELEMETRY_CONFIG)
2. Environment variable (TELEMETRY_CONFIG)
3. If none of the above is set, default configuration will be used (cii.config://*/telemetry/config/defaultArchiverConfig)

If the configuration is stored on remote Configuration service, a CONFIG_SERVICE_LOCATION variable has to be set as well (e.g: zpb.rr://hostname:port/).

The order of the Configuration service location flag is the following:

1. Command line flag (-DCONFIG_SERVICE_LOCATION)
2. Environment variable (CONFIG_SERVICE_LOCATION)
3. If none of the above is set, default Configuration service location will be used (tcp://localhost:6379)

If one would like to run multiple TAs on the same server (or virtual machine), one has to use command line flag (-DTELEMETRY_CONFIG) as otherwise two services will try to bind on the same port and only one will succeed.

Make sure that the serviceRange field in the TAs configuration is unique, otherwise different archivers will subscribe to the same data points with the same capturing rules and you will end up with duplicated data in the archive.

11.8.1 Example

Say we have our custom TA configuration located in the CII configuration service under the cii.config://*/telemetry/config/custom location.

- Start TA with -DTELEMETRY_CONFIG="cii.config://*/telemetry/config/custom/" command line flag.
- Store data capture configurations in configuration service under location that is specified in the serviceRange (see also section 4). If serviceRange is specified as cii.config///*telemetry/archiver1, the data capture configuration should be stored under cii.config*///telemetry/archiver1/datacapture/<data capture UUID>

11.8.2 Systemd Deployment

If you are running multiple services via systemd, you have to deploy multiple systemd unit files that will start the Telemetry Archiver executables with different -DTELEMETRY_CONFIG flag set.

11.8.3 Scaling Telemetry Archiver

The actual load that one Telemetry Archiver can handle entirely depends on the hardware on which the TA is running and the desired performance needs (number of subscribed data points and their update frequency). The TA was designed in a way to support horizontal scaling, meaning if more throughput is necessary you can easily deploy more archivers.

For maximum performance of the TA it is recommended to deploy it on a different machine than the one where CII OLDB Service or Engineering Archive are deployed on. Nevertheless, if your desired performance needs are lower than one server can handle then you are free to run an entire infrastructure on one server.

The only way to get the actual performance numbers of one TA node is to run performance tests on the production hardware and measurements. This is usually done by slowly increasing the load of the TA and observing the metrics (CPU, RAM, JVM metrics and TA metrics through JMX). The following are some of the general problems and solutions that are likely to occur during performance measurements:

- If archiving buffer starts filling up in the TA (check JMX metrics), that means you have to increase the performance of the Engineering Archive. Before you start tinkering with the Engineering Archive, you might want to try and increase or decrease the archiving buffer batch size through the TA's configuration. It's possible that decreasing or increasing the batch size might improve the archiving throughput. During the performance tests that we performed, the Engineering Archive was always a bottleneck.
- If CPU is the bottleneck and fully utilized add more cores or share the load by moving some of the subscribed data points to another TA node. For maximum performance make sure the TA does not share CPU resources with another virtual machine that is running on the same server.

- If RAM is the bottleneck, add more RAM or share the load with another TA. A rule of thumb is to size the heap in a way that after a garbage collection, cca. 30% of the heap is still occupied. You can try to modify the heap size via JVM flags (-Xmx, -Xms). For maximum performance ensure that the system is not swapping (on high performance servers swap is usually disabled).
- If the network interface can't handle the incoming data, then you have to act accordingly to your infrastructure capabilities.
- Running the TA on a more recent JVM (11) might improve performance by 10-15%, due to the improvements made in G1GC garbage collector. If you notice long stop the world GC pauses, you can try tuning the GC via JVM flags. If long GC pauses are proven as problematic, you might want to try switching to one of the newer concurrent garbage collectors (Shenandoah, ZGC) that promise garbage collector pauses lower than 10ms.

To get a better feeling of what is really going on with the system, one might want to use the CLI tools that are usually installed on a Linux machine:

- vmstat - reports information about processes, memory, paging, block IO, traps, disks and CPU activity.
- iostat - reports Central Processing Unit (CPU) statistics and input/output statistics for devices, partitions and network filesystems (NFS).
- nicstat - prints out network statistics for all network cards (NICs), including packets, kilobytes per second, average packet sizes and more.

11.8.4 Logging

For logging purposes, the Telemetry Archivers are using CiiLogManager which is configured through logging configuration file. For more information about logging and possible configuration options, see the CII Log user manual [7].

11.9 Advanced Topics

11.9.1 Telemetry Archiver statistics

Telemetry Archiver exposes basic statistic using the Java MBeans JMX interface. Statistics can be obtained with the usage of jconsole. The user should start the jconsole with the same user account that started Telemetry Archiver and running the following command:

```
jconsole&
```

Telemetry Archiver can be chosen from the list of processes and connected to by clicking on the Connect button (Figure 9-1).

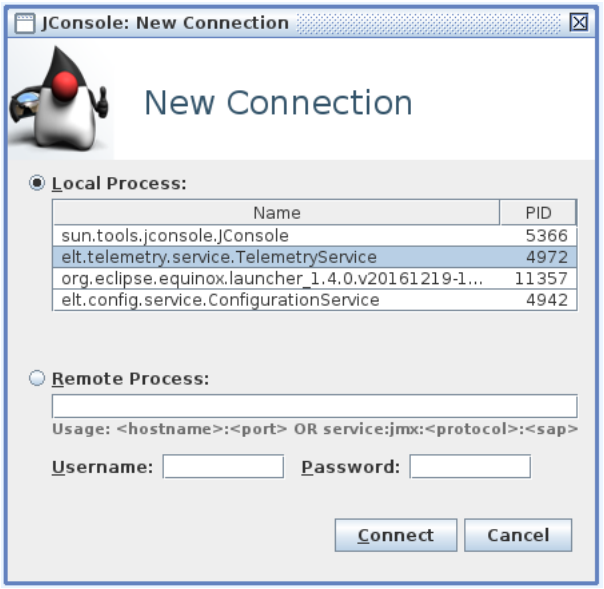


Figure 9-1: Connect to Telemetry Archiver with Jconsole

After the connection is made, the console with statistics is displayed. The console displays heap usage, CPU usage, number of threads, and similar java statistics. The specific Calculation service statistics are exposed under the elt.telemetry.service.stat namespace.

The statistics show multiple metrics:

- current time,
- service up-time,
- number of working threads,
- EA status,

- number of subscribed data points,
- number of dropped data points,
- buffer fullness,
- number of archived data points in last hour,
- number of archiving errors,
- number of archived data points in a day,
- number of transferred data packets,
- number of data packets not inserted,
- number of failed data point subscriptions,
- number of invalid data packets.

To obtain statistics, the user should click on the MBeans tab and browse the elt.telemetry.service.stat namespace (Figure 9-2).

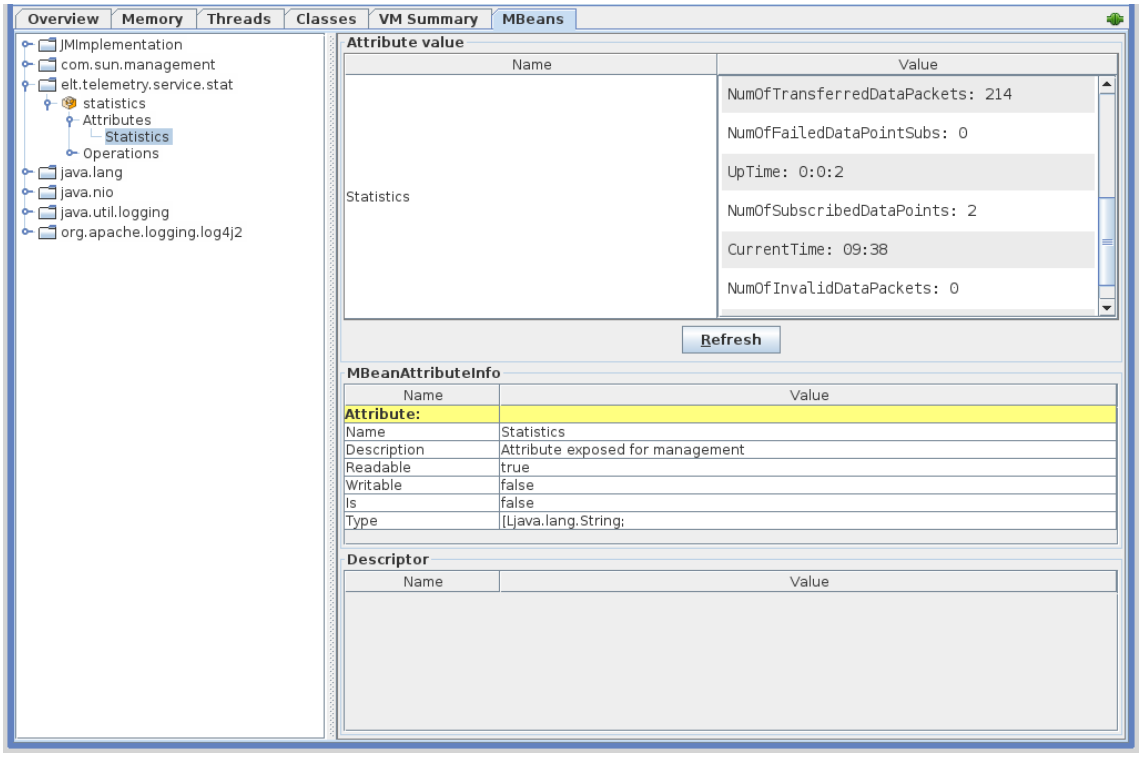


Figure 9-2: Telemetry Archiver Statistics

11.10 Archive API

11.10.1 Archive API: storing, querying and downloading data

Telemetry Archive API provides users with the functionality to store, query and download data that is saved in the Engineering Archive. The methods, specified in the TelemetryArchiveApi interface are implemented in the TelemetryArchiveApiSyncImpl class.

Table 9.1: Archive API

Return Type	Method and Description
String	storeData(DataPacket dataPacket) Stores a data packet (containing a timestamp, location of the data in the big data database and data in JSON format) to the Engineering Archive. Returns a string specifying the assigned ID of the data packet.
List<String>	storeDataList(List<DataPacket> dataPackets) Stores a list of data packets to the Engineering Archive. Returns a list of strings specifying the assigned IDs of the data packets.
List<String>	queryData(String queryString) Queries data packets in the Engineering Archive based on the provided query string (ES Query DSL should be used). Returns a list of strings specifying the IDs of the corresponding data packets.
List<DataPacket>	downloadData(List<String> dataPacketIDs) Downloads data packets with provided IDs from the Engineering Archive.

11.10.2 Archive API: Big Data Storage

The Hadoop Storage library provides users with the functionality of writing data to and reading data from the big data database of the Engineering Archive. The HadoopStorage class implements the methods of the BigDataStorage interface.

Table 9.2: Big Data Storage API

Return Type	Method and Description
String	write(URI uri, byte[] data) Writes the data to the file on the location, specified by the big data archive URI and the automatically generated UUID file name. Returns the file name.
String	write(URI uri, InputStream stream) Writes the data from the stream to the file on the specified location and automatically generates UUID file name. Returns the file name.
void	write(URI uri, int version, String filename, byte[] data) Writes the data to the file on the location, specified by the big data archive URI, version and file name.
void	write(URI uri, String filename, InputStream stream) Writes the data to the file on the location, specified by the big data archive URI and file name.
void	write(URI uri, int version, String filename, InputStream stream) Writes the data to the file on the location, specified by the big data archive URI, version and file name.
byte[]	read(URI uri) Reads data from the file on the location specified by the URI.
byte[]	read(URI uri, String filename) Reads data from the file on the location, specified by the big data archive URI and file name.
byte[]	read(URI uri, int version, String filename) Reads data from the file on the location, specified by the big data archive URI, version and file name.
byte[]	read(URI uri, OutputStream stream) Reads data from the file on the location specified by the URI. The data is written to the output stream.
void	read(URI uri, String filename, OutputStream stream) Reads data from the file on the location specified by the URI and file name. The data is written to the output stream.
void	read(URI uri, int version, String filename, OutputStream stream) Read data from the file on the location specified by the URI, the version and file name. The data is written to the output stream.
void	delete(URI uri, String filename) Deletes file on the location, specified by the big data archive URI and file name.
void	delete(URI uri, int version, String filename) Deletes file on the location, specified by the big data archive URI, version and file name.

11.10.3 Telemetry Archiver Management API

The Telemetry Archiver Management API provides users with the functionality to refresh configurations. The methods, specified in the TelemetryServiceManagement interface are implemented in the TelemetryServiceManagementSyncImpl class.

Table 9.3: Service Management API

Return Type	Method and Description
void	refreshConfigurations(List<String > ids) Refreshes configurations, specified by their IDs. The IDs are the UUIDs of the data capture configuration that can be obtained either through querying the config-service or GUI, for more info see section 5.4.3
void	refreshAllConfigurations() Refreshes all configurations.

11.11 Data Capture Configuration Options

The data capture configurations contain the following fields, which could be tweaked in order to get the desired archiving behaviour:

Table 9.4: Data capture configuration options

Field	Description	Example
dataPointAddress	Address of data point in OLDB	cii.oldb:///datapoints/dp1/
archiveMode	Defines on which on-change event the data point is archived (data point value change, quality flag change, time based – e.g. once per day)	VALUE_CHANGE or QUALITY_CHANGE or METADATA_CHANGE
deltaValue	Delta value change required to archive (in the case of data value change event)	+/- 1.0
deltaType	Type that the delta value field represent (absolute change or relative change) If RELATIVE delta type is chosen, the delta value represents the percentage of the data point value for which the data point value should change in order to be archived. The value 1 represents 100% change and value 0.01 represents 1% change. If ABSOLUTE delta type is chosen, the delta value represents the actual value for which the data point value should have changed in order to be archived.	ABSOLUTE or RELATIVE
minimumInterval	Minimum time between specific data point archiving in the *hh:mm:ss* format. If the data point value has changed multiple times in a short interval, it ensures we don't store too many values. If minimum interval is set to 0, every data point value change will be archived according to the delta field.	00:00:10
maximumInterval	Maximum interval between two data point archiving events in the *hh:mm:ss* format. If the data point value has not changed in the specified maximum interval, the data point value will still be archived. Setting it to 0 will disable it.	01:15:00 (data point should be archived at least on every 1 hour and 15 minutes)
archivingEnabled	Enable archiving (True) or disable archiving (False) for this specific data capture configuration	True/False

11.12 Telemetry Archiver Configuration

Listing 9-1 Telemetry Archiver configuration YAML

```
engineeringArchiveBackupURI: "http://ciiarchivehost:9200/"
engineeringArchiveURI: "http://ciiarchivehost:9200/"
largeStorageServiceURI: "http://ciihdfshost:9870/"
serviceRange:
  - "cii.config://remote/telemetry/serviceRange2"
```

Each Telemetry Archiver contains its configuration stored in the CII Configuration Service.

Table 9.5 contains a description of all configuration fields. If Telemetry Archiver needs to use custom configuration, a java parameter DTELEMETRY_CONFIG_INI or environment variable TELEMETRY_CONFIG_INI must be specified with the location to the custom configuration YAML. An example of configuration YAML is shown in Listing 9-1.

Table 9.5: Telemetry Archiver Configuration settings

Field	Description	Example
telemetry-ServiceURI	Defines an IP address and port on which the Telemetry Archiver is bound to	zpb.rr://0.0.0.0:9115/ Default is zpb.rr://0.0.0.0:9115/
simu-lateArchive	If set to true, all data archive operations will be simulated. If set to false, data will be archived to Engineering archive	True/False Default is False
simulate-Capture	If set to true, all data capture is simulated (using simulator instead of real OLDB)	True/False Default is False
archive-Quality-Mask	Permits selective archiving of a data points based on data points quality flag Possible values: UNKOWN, INVALID, VALID, SIMULATED, ALL, NONE	UNKOWN, VALID Default is ALL
engi-neeringArchiveURI	Engineering Archive location	Default is https://ciiarchive:9200/
engi-neeringArchive-Bac kupURI	Engineering Archive backup URI	Default is https://ciiarchive2:9200/
maxSearch-Size	The maximum number of data packets stored in the EA that can be retrieved at once	5000 Default is 10000
authentica-tionToken	Authentication token that allows service to interact with the archiving system	Default is empty string
ser-viceRange (required)	Defines the configuration service URI location from where the data capture configuration will be loaded. Multiple service ranges are allowed here	[cii.config://*/telemetry/service1]
archiving-BufferSize	Maximum number of data packets that the archiving buffer can contain	50000 Default is 100000
bufferBatch-Size	Maximum number of data packets that one archiving batch can contain	150 Default is 400
maximum-BatchWait	Maximum number of seconds that the archiving buffer will wait for data before sending a batch to the Engineering Archive (even if the batch is not full).	5 Default is 1

11.13 Big Data Storage Examples

The following listings show examples of the usage of the Big Data Storage for the Java, CPP and Python programming languages.

Listing 9-2: Example usage of the Big Data Storage in Java

```
package sampleapp;

import java.io.IOException;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.net.URI;
import java.net.URISyntaxException;
import elt.storage.HadoopStorage;
import elt.storage.BigDataStorage;

/**
 * Hadoop Storage example.
 */
public class HadoopStorageExample {

    /**
     * Private method to generate content to be written to the remote file
     */
    private static String generateContent() {
      String s = "AAAA";
      for (int i = 0; i < 100; i++) {
        switch(i % 5) {
          case 0:
            s += "cccc";
            break;
          case 1:
            s = s + s;
            break;
          case 2:
            s += "XXXXXXXXXX";
            break;
          case 3:
            s += "!!!!!!";
            break;
          case 4:
            s += "<<<>>>";
            break;
        }
      }
      return s;
    }

    /**
```

(continues on next page)

(continued from previous page)

```
     * HadoopStorage main method.
     *
     * @param args Command line arguments (none are required).
     */
    public static void main(String[] args) throws URISyntaxException,␣
↪IOException, InterruptedException {

        int status = 0;
        final BigDataStorage hadoopStorage =
                new HadoopStorage(new URI("http://ciihdfshost:9870"), "/esoSample
↪");
        // contains path to the archive (remote directory)
        final URI uri = new URI("cii.config://*/archive");
        // remote filename
        final String remoteFilename = "file1.bin";
        // content to store
        final String contentToStore = generateContent();

        try {
          // save data to Hadoop, use binary stream
          ByteArrayInputStream stream = new ByteArrayInputStream(contentToStore.
↪getBytes());
          System.out.println("Writing blob to hadoop storage...");
          hadoopStorage.write(uri, remoteFilename, stream);

          // read data from Hadoop
          ByteArrayOutputStream ostream = new ByteArrayOutputStream();
          System.out.println("Reading blob from hadoop storage...");
          hadoopStorage.read(uri, remoteFilename, ostream);
          ostream.close();

          if (contentToStore.equals(ostream.toString())) {
            System.out.println("OK, saved and read content are the same");
          } else {
            System.err.println("FAIL: data from remote not same as source");
            status = 1;
          }

          // delete a file from Hadoop
          hadoopStorage.delete(uri, remoteFilename);
      } catch (Throwable th) {
          System.err.println(String.format("Exception during remote operation: %s",
↪ th));
        status = 5;
      }
      try {
        // try to read from non existent file
        System.out.println("Access to non existing remote file (should fail)...
↪");
```

(continues on next page)

(continued from previous page)

```
        byte[] data = hadoopStorage.read(uri, remoteFilename);
      } catch (Throwable th) {
        System.err.println(String.format("REMOTE READ FAILURE (expected): %s",␣
↪th));
      }
      System.exit(status);
    }
}
```

Listing 9-3: Example usage of the Big Data Storage in CPP

```
#include <iostream>
#include <sstream>
#include <string>
#include <cstdlib>
#include <ciiHadoopStorage.hpp>
#include <ciiException.hpp>

/**
 * Generates sample string
 * @return string, generated content
 */
static std::string generateContent() {
  std::string s = "AAAA";
  for (std::size_t i = 0; i < 100; i++) {
    switch (i % 5) {
      case 0:
        s += "BBBB";
        break;
      case 1:
        s = s + s;
        break;
      case 2:
        s += "XXXXXXXXXX";
        break;
      case 3:
        s += ".....";
        break;
      case 4:
        s += "<<<>>>";
        break;
    }
  }
  return s;
}


/**
 * main method
```

(continues on next page)

(continued from previous page)

```
 */
int main(int argc, char **argv) {
  int status = 0;
  const elt::mal::Uri hadoopLocation("http://ciihdfshost:9870");
  const std::string hadoopEndpoint("/esoSample");
  const std::string remoteDirectory = "archive";
  const std::string remoteFilename = "file.bin";
  const std::string contentToStore(generateContent());

  // Initialize hadoop storage,
  // First parameter is Uri of the webhdfs server
  // Second parameter is name of the top directory under which all files for this
  // instance of hadoop storage will be stored.
  elt::storage::CiiHadoopStorage storage(hadoopLocation, hadoopEndpoint);

  // In this example, stream is used
  std::stringstream stream(contentToStore);
  // Create receiving stream
  std::ostringstream ostream;

  try {
    // Write content of the stream to the remote file
    std::cout << "Writing blob to hadoop storage..." << std::endl;
    storage.write(remoteDirectory, remoteFilename, stream);

    // Read content of the stream from the remote file
    std::cout << "Reading blob from hadoop storage..." << std::endl;
    storage.read(remoteDirectory, remoteFilename, ostream);

    // Remove remote file
    storage.remove(remoteDirectory, remoteFilename);

    if (ostream.str() == contentToStore) {
      std::cout << "OK, saved and read content are the same" << std::endl;
    } else {
      std::cerr << "FAIL: data from remote not same as source" << std::endl;
      status = 1;
    }
  } catch (const elt::error::CiiException &e) {
    std::cerr << "Cii Exception during remote operation: " << e.what() <<␣
↪std::endl;
    status = 2;
  } catch (const std::exception &e) {
    std::cerr << "Std Exception during remote operation: " << e.what() <<␣
↪std::endl;
    status = 5;
  }

  // Read from non existing remote file
```

(continues on next page)

(continued from previous page)

```
  try {
    std::cout << "Access to non existing remote file (should fail)..." <<␣
↪std::endl;
    storage.read(remoteDirectory, remoteFilename, ostream);
  } catch (const std::exception &e) {
    std::cerr << "REMOTE READ FAILURE (expected): " << e.what() << std::endl;
  }
  return status;
}
```

Listing 9-4: Example usage of the Big Data Storage in Python

```
#!/usr/bin/env python

#pylint: disable=E1101,C0103,R0914,C0330,W0612
#This script demonstrates usage of HadoopStorage

import sys
import io
import traceback
import elt.storage

def generate_content():
    """Helper function that generates content string"""
    s = 'AAAA'
    for i in range(100):
        modulo = i % 5
        if modulo == 0:
            s += 'bbbb'
        elif modulo == 1:
            s += s
        elif modulo == 2:
            s += 'XXXXXXXXXX'
        elif modulo == 3:
            s += ',,,,,,'
        else:
            s += '<<<>>>'
    return s

# Define URI for remote webhdfs server to be used
hadoop_location = 'http://ciihdfshost:9870'
# Define top level remote directory
hadoop_endpoint = 'esoSample'
# Define remote directory
remote_directory = 'archive'
# Define remote filename
remote_filename = 'file.bin'
```

(continues on next page)

(continued from previous page)

```
def _main():
    """main method implementation"""
    status = 0
    # Initialize hadoop storage.
    # First parameter is string containing Uri of the webhdfs server
    # Second parameter is name of the top directory under which all files
    # for this instance of hadoop storage will be stored.
    storage = elt.storage.CiiHadoopStorage(hadoop_location, hadoop_endpoint)

    # In this example, stream is used
    content_to_store = generate_content()
    # Content to write must be raw binary values. Use io.BytesIO!
    stream = io.BytesIO(content_to_store.encode('utf-8'))
    try:
        print('Writing blob to hadoop storage...')
        # Write content of the stream to the remote file
        storage.write_from(stream, remote_directory, remote_filename)

        # Read content of the remote file into the stream
        ostream = io.BytesIO()
        print('Reading blob from hadoop storage...')
        storage.read_into(ostream, remote_directory, remote_filename)

        # Delete remote file
        storage.remove(remote_directory, remote_filename)

        if content_to_store == ostream.getvalue().decode('utf-8'):
            print("OK, saved and read content are the same")
        else:
            print("FAIL: data from remote not same as source", file=sys.stderr)
            status = 1
    # pylint: disable=W0703
    except Exception as e:
        traceback.print_tb(sys.exc_info()[2])
        print("Exception during remote operation: ", e, file=sys.stderr)
        status = 5

    # try to read from non existing remote file
    try:
        print('Access to non existing remote file (should fail)...')
        ostream = io.BytesIO()
        storage.read_into(ostream, remote_directory, remote_filename)
    # pylint: disable=W0703
    except Exception as e:
        print("REMOTE READ FAILURE (expected): ", e, file=sys.stderr)
    return status

def main():
    """main method wrapper"""
```

(continues on next page)

(continued from previous page)

```
    result = 0
    try:
        result = _main()
    #pylint: disable=W0703
    except Exception as e:
        print(e)
        traceback.print_tb(sys.exc_info()[2])
        result = 5
    return result

if __name__ == '__main__':
    sys.exit(main())
```

CHAPTER

TWELVE

SERVICE MANAGEMENT

Document ID:	
Revision:	1.1
Last modification:	February 29, 2024
Status:	Released
Repository:	https://gitlab.eso.org/cii/info/cii-docs
File:	services.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
1.0	04.10.2021	mschilli	All	Created.
1.1	29.02.2024	mschilli	All	CII v4: updated, Public doc

Confidentiality

This document is classified as Public.

Scope

This document is a user manual for the service management of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

12.1 Overview

This document is a user manual for CII Services management tooling.

12.2 Introduction

This document describes the tools available for the management of CII Services

- cii-postinstall - for setting up CII services on a host
- cii-services - for starting / stopping / monitoring of the CII Services

12.3 cii-postinstall

You run the CII-PostInstall on a freshly installed DevEnv machine, or after an upgrade of the DevEnv. You choose a role for the host, and the utility will apply the corresponding configuration steps. For most configuration steps, root privileges are needed.

Usage:

```
$ /elt/ciisrv/postinstall/cii-postinstall <role>
```

Run the command without arguments to see its help.

12.3.1 Roles

The role “ownserver” configures the services for local development on a single machine.:

```
$ /elt/ciisrv/postinstall/cii-postinstall  role_ownserver
```

The role “groupserver” does a few more things than the one above. It runs the services with more resources and makes the services/databases accessible from other hosts.:

```
$ /elt/ciisrv/postinstall/cii-postinstall  role_groupserver
```

The role “groupclient” configures the host to use the services provided by a groupserver. For example, assuming host “elthlccd20” has been post-installed with the role “groupserver”, then, to let applications on your host use the CII Services on that groupserver via the groupserver’s non-deterministic network interface (alternatively, you can specify the IP-Address of the server’s network interface), you do:

```
$ /elt/ciisrv/postinstall/cii-postinstall  role_groupclient  elthlccd20-cnd
```

Switching Roles?

It is possible to run the utility again later, but your mileage will vary: some role-switches will work, others not.

Redo post-install

It is possible to re-apply/repair/update the already selected role, by doing:

```
$ /elt/ciisrv/postinstall/cii-postinstall  knownrole
Loading 'known role' for this host: role_groupclient elthlccd20-cnd
Update/Installation started, please wait....
```

12.3.2 Subcommands

A “role” is a collection of configuration steps, and those configuration steps can also be applied one-by-one, by executing them as a “subcommand”. To see the available subcommands, just pass “help”.:

```
$ /elt/ciisrv/postinstall/cii-postinstall commands
Available subcommands: role_ownserver  role_groupserver  role_groupclient ␣
↪knownrole  logsink  oldb  rsyslog  rsyslog_collect  rsyslog_forward ␣
↪serverlocation  help
```

12.4 cii-services

The *cii-services* utility lets you start/stop/monitor the CII services. For some operations, it requires root-privileges -> unless you are running it with root-privileges, it will show a password prompt when needed.

Usage:

```
$ cii-services
CII Services Tool (20240220)
Available subcommands: info logs start stop restart enable disable␣
↪status(deprecated)
```

12.4.1 Monitoring/Status Check

info install

A deployment-centric view showing which services are running on the current host:

```
.. code-block:: ruby
```

$ cii-services info install CII Services Tool (20240220)

install .... [oldb] config-client-ini |install: [oldb] cii-oldb-default-redis |active:no |boot:n |install: [oldb] cii-oldb-calc-daemon |active:no |boot:n |install: [oldb] cii-oldb-calc-scheduler |active:no |boot:n |install: [log] rsyslog |active:yes |boot:y |install:/usr/sbin/rsyslogd [log] systemd-journald |active:yes |boot:y |install:/usr/lib/systemd/systemd-journald [log] logrotate |active:yes |boot:y |install:/usr/sbin/logrotate [trace] jaeger-all |active:no |boot:n |install:/usr/local/bin/jaeger-all-in-one [telem] srv-telemetry |active:no |boot:n |install: [alarm] cii_ias |active:no |boot:n |install: [alarm] kafka |active:no |boot:n |install: [alarm] kafka-zookeeper |active:no |boot:n |install: [alarm] cii_alarm_mon |active:no |boot:n |install: [deprec] filebeat |active:no |boot:n |install: [deprec] logstash |active:no |boot:n |install: [deprec] kibana |active:no |boot:n |install: [deprec] elasticsearch |active:no |boot:n |install: [deprec] minio |active:no |boot:n |install:

info access

A deployment-centric view showing which CI services are being used:

```
.. code-block:: ruby
```

$ cii-services info access CII Services Tool (20240220)

discovering ..... # access ... [oldb] config-redis |access:no |host:localhost |IP:127.0.0.1 |Port:6379 [oldb] redis-server |host: |IP: |Port:0 [oldb] pubsub-server |host: |IP: |Port:0 [oldb] calc-node [trace] jaeger |access:no |host:localhost |IP:127.0.0.1 |Port:14269 [telem] telem |host:localhost |IP:127.0.0.1 |Port:9115 [alarm] alarm-ias

|host:localhost |IP:127.0.0.1 [alarm] alarm-mon |access:n/a |host:localhost |IP:127.0.0.1 |Port:5602

info stats

Details about the usage of the services:

```
.. code-block:: ruby
```

$ cii-services info stats CII Services Tool (20240220)

stats .... config-redis |total_connections_received:81 |rejected_connections:0 redis-server |total_connections_received:82 |rejected_connections:0 jaeger telem

info function

A feature-centric view, that tells you which features you have available:

```
.. code-block:: ruby
```

$ cii-services info function CII Services Tool (20240220)

function ... Log |functional:yes OLDB DP |functional:yes OLDB CE |functional:yes IntCfg |functional:yes

logs

Shows journal logs of CII Services. Privileged Operation. Run the command as root:

```
.. code-block:: ruby
```

$ su -c “cii-services logs” Password: CII Services Tool (20240220)

Service Logs ....

[cii-oldb-default-redis] – No entries –

[rsyslog] Jan 29 20:20:23 eltmal28 rsyslogd[946]: imjournal: journal files changed, reloading... [v8.2306.0-1.fc38 try https://www.rsyslog.com/e/0] Jan 29 20:20:24 eltmal28 rsyslogd[946]: imjournal: journal files changed, reloading... [v8.2306.0-1.fc38 try https://www.rsyslog.com/e/0] Feb 04 00:00:23 eltmal28 systemd[1]: rsyslog.service: Sent signal SIGHUP to main process 946 (rsyslogd) on client request.

12.4.2 Start / Stop

Privileged Operations. You will be asked for the root password.:

```
.. code-block:: ruby
```

$ cii-services start CII Services Tool (20240220) Possible args: one or more groups: oldb, log, trace, all, or any unit names from the all group

$ cii-services start all CII Services Tool (20210706) Password:

$ cii-services stop all CII Services Tool (20210706) Password:

12.5 Example

How to read the tool's output:

```
$ cii-services info
```

Section 1 reports some info about the tool itself:

```
CII Services Tool (20240220)

# install .................. ..............
```

Tool version and progress dots. If an error occurs while collecting information, run the command with "-v" or "-vv".

Section 2 reports which CII services are installed on your host, whether they are running, and whether they are automatically started at machine boot-up:

```
.. code-block:: ruby
```

> # install .... [oldb] config-client-ini |install: [oldb] cii-oldb-default-redis |active:no |boot:n |install:/usr/bin/redis-server [oldb] cii-oldb-calc-daemon |active:no |boot:n |install:/elt/ciisrv/bin/srv-oldb-calculation [oldb] cii-oldb-calc-scheduler |active:no |boot:n |install:/elt/ciisrv/bin/srv-oldb-scheduler [log] rsys-log |active:yes |boot:y |install:/usr/sbin/rsyslogd [log] systemd-journald |active:yes |boot:y |install:/usr/lib/systemd/systemd-journald [log] logrotate |active:yes |boot:y |in-stall:/usr/sbin/logrotate [trace] jaeger-all |active:no |boot:n |install:/usr/local/bin/jaeger-all-in-one [telem] srv-telemetry |active:no |boot:n |install: [alarm] cii_ias |active:no |boot:n |install: [alarm] kafka |active:no |boot:n |install: [alarm] kafka-zookeeper |active:no |boot:n |install: [alarm] cii_alarm_mon |active:no |boot:n |install:

Several services (not: Telemetry and Alarms) are installed on your host. Only the log services are running on your host. Note the log services are recommended to run always. No other CII services are running on your host. If you plan to use CII services running on another host, this is fine. Otherwise, you need to start the services, see above.

Section 3 reports where applications on your host will find the CII services, and whether the cii-services tool was able to access them (= talk to them). The services may be running on your host (previous section) or on another host.:

```
.. code-block:: ruby
```

> # access ... [oldb] config-redis |access:yes |host:ciiconfservicehost |ip: |port:6379 [oldb] redis-server |host: |ip: |port:0 [oldb] pubsub-server |host: |ip: |port:0 [oldb] calc-node [trace] jaeger |access:no |host:localhost |ip:127.0.0.1 |port:14269 [telem] telem |host:localhost |ip:127.0.0.1 |port:9115 [alarm] alarm-ias |host:localhost |ip:127.0.0.1 [alarm] alarm-mon |access:n/a |host:localhost |ip:127.0.0.1 |port:5602

The empty ip fields tell us that hostname resolution is not configured. This means applications running

on your host are not able to locate the CII services. You need to configure your host, see cii-postinstall

Section 4 reports some statistics from the services.:

```
.. code-block:: ruby
```

    # stats .... config-redis redis-server jaeger telem

An empty output like this again indicates that the the services are not accessible, but you knew that already.

Sections 5 reports whether the services are working:

```
.. code-block:: ruby
```

    # function .... Log |functional:yes OLDB DP |functional:no OLDB CE |functional:no IntCfg |functional:yes

Explanation:

- Log : syslog logging (to /var/log/elt/elt.log)
- OLDB DP: datapoint storage
- OLDB CE: calculation engine
- IntCfg : internal config-storage

Preview: After running post-install and cii-services start, the output will change to something like this:

```
$ cii-services info
CII Services Tool (20240220)

# install .................. ..............
[oldb]    config-client-ini                          |install:
[oldb]    cii-oldb-default-redis   |active:yes  |boot:y  |install:/usr/bin/redis-
↪server
[oldb]    cii-oldb-calc-daemon     |active:yes  |boot:y  |install:/elt/ciisrv/
↪bin/srv-oldb-calculation
[oldb]    cii-oldb-calc-scheduler  |active:yes  |boot:y  |install:/elt/ciisrv/
↪bin/srv-oldb-scheduler
[log]     rsyslog                  |active:yes  |boot:y  |install:/usr/sbin/
↪rsyslogd
[log]     systemd-journald         |active:yes  |boot:y  |install:/usr/lib/
↪systemd/systemd-journald
[log]     logrotate                |active:yes  |boot:y  |install:/usr/sbin/
↪logrotate
[trace]   jaeger-all               |active:no   |boot:y  |install:/usr/local/bin/
↪jaeger-all-in-one
[telem]   srv-telemetry            |active:no   |boot:n  |install:
[alarm]   cii_ias                  |active:no   |boot:n  |install:
[alarm]   kafka                    |active:no   |boot:n  |install:
[alarm]   kafka-zookeeper          |active:no   |boot:n  |install:
```

(continues on next page)

(continued from previous page)

```
[alarm]   cii_alarm_mon            |active:no   |boot:n  |install:
[deprec]  filebeat                 |active:no   |boot:n  |install:/usr/share/
↪filebeat/bin/filebeat
[deprec]  logstash                 |active:no   |boot:n  |install:/usr/share/
↪logstash/bin/logstash
[deprec]  kibana                   |active:no   |boot:n  |install:/usr/share/
↪kibana/bin/kibana
[deprec]  elasticsearch            |active:no   |boot:n  |install:/usr/share/
↪elasticsearch/bin/elasticsearch
[deprec]  minio                    |active:no   |boot:n  |install:/usr/local/bin/
↪minio

# discovering ........
# access ...
[oldb]   config-redis   |access:yes  |host:ciiconfservicehost  |ip:127.0.0.1 ␣
↪|port:6379
[oldb]   redis-server   |access:yes  |host:localhost           |ip:127.0.0.1 ␣
↪|port:6379
[oldb]   pubsub-server  |access:yes  |host:localhost           |ip:127.0.0.1 ␣
↪|port:6379
[oldb]   calc-node      |access:yes  |host:localhost           |ip:127.0.0.1 ␣
↪|port:9117
[trace]  jaeger         |access:no   |host:localhost           |ip:127.0.0.1 ␣
↪|port:14269
[telem]  telem                       |host:localhost           |ip:127.0.0.1 ␣
↪|port:9115
[alarm]  alarm-ias                   |host:localhost           |ip:127.0.0.1
[alarm]  alarm-mon      |access:n/a  |host:localhost           |ip:127.0.0.1 ␣
↪|port:5602

# stats ....
config-redis   |total_connections_received:6  |rejected_connections:0
redis-server   |total_connections_received:7  |rejected_connections:0
jaeger
telem

# function ...
Log      |functional:yes
OLDB DP  |functional:yes
OLDB CE  |functional:no
IntCfg   |functional:no
```

CHAPTER

THIRTEEN

INTERNAL CONFIG SYSTEM

Document ID:	
Revision:	1.11
Last modification:	March 18, 2024
Status:	Released
Repository:	https://gitlab.eso.org/cii/info/cii-docs
File:	config.rst
Project:	ELT CII
Owner:	Marcus Schilling

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
0.1	10.04.2019	bterpinc	All	Document creation
0.2	15.04.2019	manovak	GUI	GUI part added
0.9	26.4.2019	bterpinc	All	Update after internal review, release
1.0	3.6.2019	bterpinc	6	Package support update
1.1	19.7.2019	bterpinc	3	Added sample application section.
1.2	11.9.2019	bterpinc/jpribosek	All	Updated after internal review. Added sample application workflow and search examples.
1.3	25.9.2019	bterpinc	Appendix C	Added sample code produced by the generators.
1.4	09.10.2019	bterpinc	Introductio n 4	Added introductio n section Added simple custom metadata sample application
1.5	12.02.2020	bterpinc	All	General rewrite of the manual. Sections added and restructure d.
1.6	20.02.2020	kstampar	All	Update of cpp file names and added flags when generating python bindings.
1.7	13.07.2020	bterpinc	Appendix C Appendix G Appendix H 5.4 2.1	Removed the MdInt64 duplicate. Added client settings Added service settings Added cache example Additional note on class versioning.
1.8	17.03.2022	mschilli	Doc Title Overview	Module is now only for CII-internal usage
1.9	24.08.2023	jrepinc	7	Added YAML config type mapping
1.10	05.09.2023	mschilli	7	Removed GUI and out- dated info
1.11	18.03.2024	mschilli	0	CII v4: Public doc

Confidentiality

This document is classified as Public.

Scope

This document is a manual for the internal configuration system of the ELT Core Integration Infrastructure software.

Audience

This document is aimed at Users and Maintainers of the ELT Core Integration Infrastructure software.

References

1. https://www.eso.org/~eltmgr/CII/latest/manuals/html/docs/oldb.html
2. https://www.eso.org/~eltmgr/CII/latest/manuals/html/docs/error_api.html
3. JsonPath, https://github.com/json-path/JsonPath

13.1 Overview

This document is a manual for the CII-internal configuration system. It is used by CII to manage the metadata of the Online database, the capture configurations of the Telemetry service, and others.

13.2 Introduction

The Core Integration Infrastructure (CII) Internal Configuration system provides the mechanisms for distributing configuration data to the CII services software and to allow the collection of configuration data from these systems.

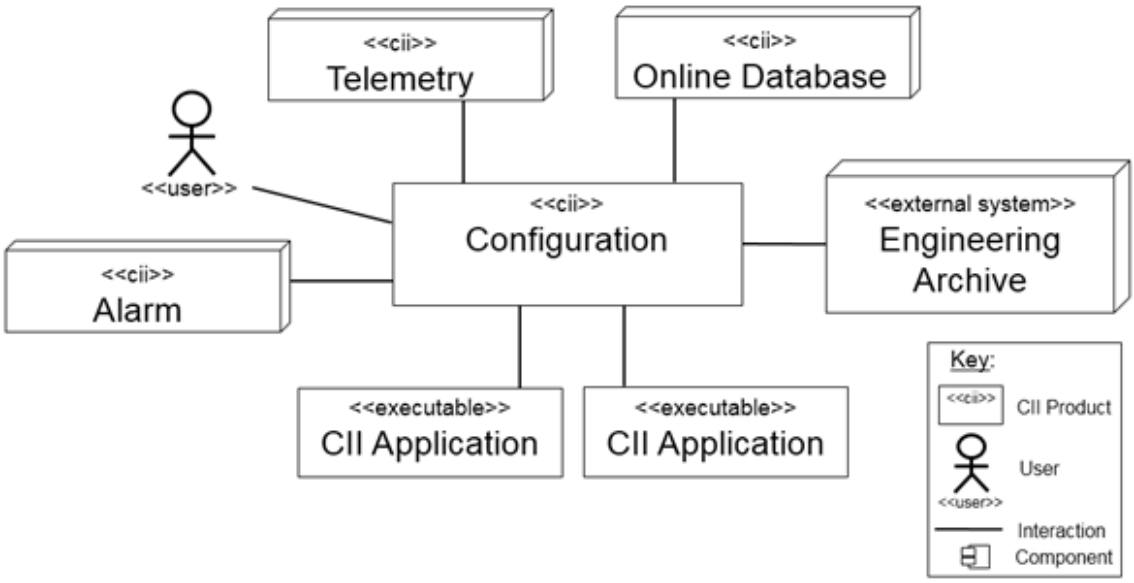


Figure 2-1 Configuration interactions

Configuration data is stored as JSON documents, and supports three different storage modes:

- **Cache**: Data is stored into client API local memory. This mode is not persistent.
- **Local DB**: Data is stored into local data storage on client computers. This mode is persistent.
- **Remote DB**: Data is stored in the Engineering Archive using the config service. As config service together with Engineering Archive storage provides a complete solution to access and store data to remote storage, it is also referred to as “remote DB”. This mode is persistent.

The configuration is a set of **data** and **metadata.** The types and structure of configuration is defined in classes. Metadata classes define the structure of config points and metadata, while configuration classes define the structure of a target configuration (configuration instance). Both configuration and metadata classes support inheritance.

The configuration and metadata classes are based on YAML structures used to generate language-specific classes. The structure of the YAML defines the class names, class member fields, and member types.

Configuration data is composed of metadata instances containing the metadata values and default config values, and target configuration containing config values of the target config points. The configuration instances and metadata instances are JSON data structures. Instances define how values of the actual data-bind to corresponding config and metadata classes.

Configuration instances are instances of the configuration classes which present the actual target

configurations. Therefor “target configuration” and “configuration instance” have the same meaning.

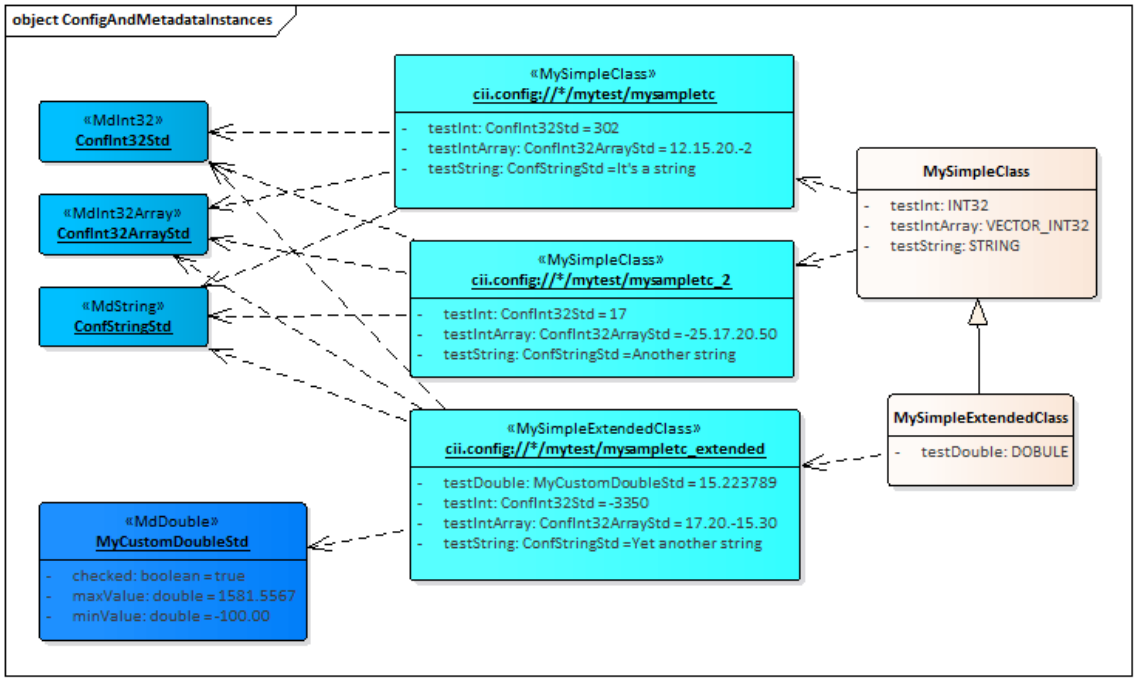


Figure 2-2 Configuration/metadata classes and instances

Figure 2-2 shows the example of relationship between configuration system (configuration and metadata) classes and instances. The figure shows configuration classes in orange, target configurations in light blue and metadata instances in dark blue colour. The configuration classes MySimpleClass and MySimpleExtendedClass define the member fields (testInt, testIntArray, testString, testDouble) of chosen type.

Based on these classes, target configurations (configuration instances) at specified endpoints are created. The following target configurations are created: *cii.config:///*mytest/mysampletc*, *cii.config:///*mytest/mysampletc_2*, *cii.config:///*mytest/mysampletc_extended*. Target configurations hold the configuration values for the fields defined in the configuration classes.

In this example, every target configuration except MyCustomDoubleStd uses default metadata instances to set the metadata of the specific field. The default metadata instances don’t hold any particular metadata information and exist for the ease of usage. The MyCustomDoubleStd is a custom metadata instance that defines the minimum and maximum value limits. The instance also sets the checked flag to true. This means that the set limits will be used when the value is set to testDouble member field. In case that the value is of the limits, CiiConfigLimitOutOfBoundsException will be thrown.

More detailed description of the system can be found in the sections below.

13.2.1 Classes

Classes only present the structure of the configuration and thus cannot contain any values. Classes, in general, have the following properties:

- Classes present the structure of the configuration.
- Classes don't contain any values.
- Classes definitions are generated into selected programming language class bindings.
- Changing the class structure forces recompilation of the client code.
- Classes cannot be versioned, but they can be updated and deleted.
- Classes are generated from remote storage or a corresponding local YAML file.

Classes cannot be versioned, but they support inheritance. In the standard workflow, classes cannot be deleted or updated. For defining a new class structure, a new derived class must be created. The new derived class must inherit from the root (configuration or metadata) class which is already a part of CII configuration. This root class has no members, so it doesn't define any class structure. Members must be added to the derived class.

Note on class versioning: As classes present the structure and are a base for the class bindings, the versioning of the classes would present additional complexity to the configuration. Every change of the class version would mean the recompilation of the code. A part of a good design principle is to have fixed structures defined. Versioning of classes would break this concept, as every version could completely change the structure. As the instances are bound to the structure, this would also mean completely changing the instances under the existing TC URI. As the versioning of the classes is not supported, the structure can be extended by using the inheritance. Inheritance maintains the base structure, but adds the ability to add additional attributes to the configuration.

Deleting and updating the classes can only be used in the process of development and testing.

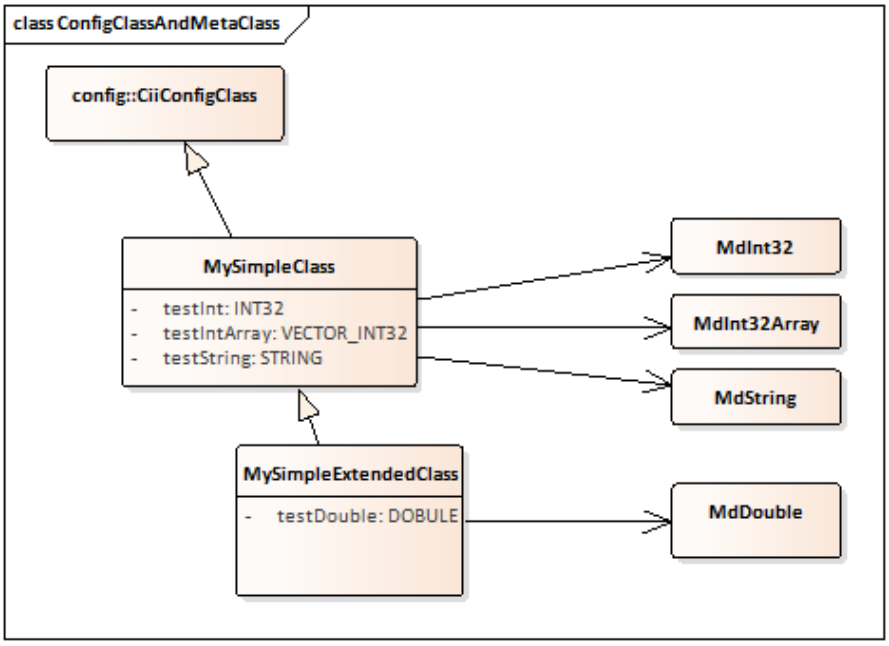


Figure 2-3 Configuration and metadata classes relationship

Figure 2-3 shows relationship diagram of config and metadata classes on basic example. The class MySimpleClass derives from the config base class (CiiConfigClass), which is the root class for all the Cii configuration classes. The MySimpleClass contains 3 member fields, testInt, testIntArray and testString. Field types are defined by metadata classes (MdInt32, MdInt32Array, MdString). For the list of the supported types refer to 2.1.2 and Appendix C.

Additionally, MySimpleExtendedClass derives from the MySimpleClass and adds additional field testDouble which is of type MdDouble. All other fields are inherited from the base class (MySimpleClass).

Configuration classes

Configuration classes are structures which define the type and member fields for generated specific language classes. They have the following properties:

- Configuration classes present nodes in the config structure.
- Configuration class member fields present the configuration data points – config points.
- Config points can be metadata classes or references to another configuration class.
- Configuration classes cannot contain complex types or list of complex types, the only exception is a reference to another class.
- All configuration classes derive from basic class (CiiConfigClass).
- All configuration classes must be stored in the elt.config.classes package.

Metadata classes

Metadata classes are structures which are used to generate language specific classes that contain metadata values. Metadata classes contain custom metadata fields and a special value field, which holds the actual value of the config point. The value field is added to the metadata classes for simplification and more straightforward usage of Java API. The actual value field is not serialized in metadata instances, but in configuration instances, as the value is not directly part of metadata. Usage of the value field in the metadata classes simplifies typing of metadata and config points, as the types for value, default value and metadata are all contained in one class.

Config points (configuration class member fields) are in most cases metadata classes. The naming convention for this classes follows the rule: Md + chosen name (e.g.: MdNumber). Metadata classes have the following properties:

- Metadata classes can only contain a limited set of CiiBaseTypes (CII Basic Data Type Set), which are synced with corresponding language types.
- Metadata classes cannot contain complex types.
- Metadata member fields cannot contain big arrays. These are limited to the special file field, which only exist in MdArray and all its derivates.
- Metadata classes support generic member types, so the type of desired members can be set according to the defined generic type of the class (as Java generics).
- All metadata classes derive from basic class (MdBase).
- All metadata classes must be stored in the elt.config.classes.meta package.

Set of basic metadata classes is pre-defined. These classes derive directly from MdBase and cannot be deleted or changed as they are part of the configuration system. Basic metadata classes also contain check functions. The check functions are executed when calling the setValue method. The exception is thrown when the value is not properly evaluated by the check function.

MdBase is a root metadata class for all metadata. All other metadata classes must derive from this class. Basic types are defined by metadata classes. The metadata types are shown in Figure 2-4 and in the table in Appendix C.

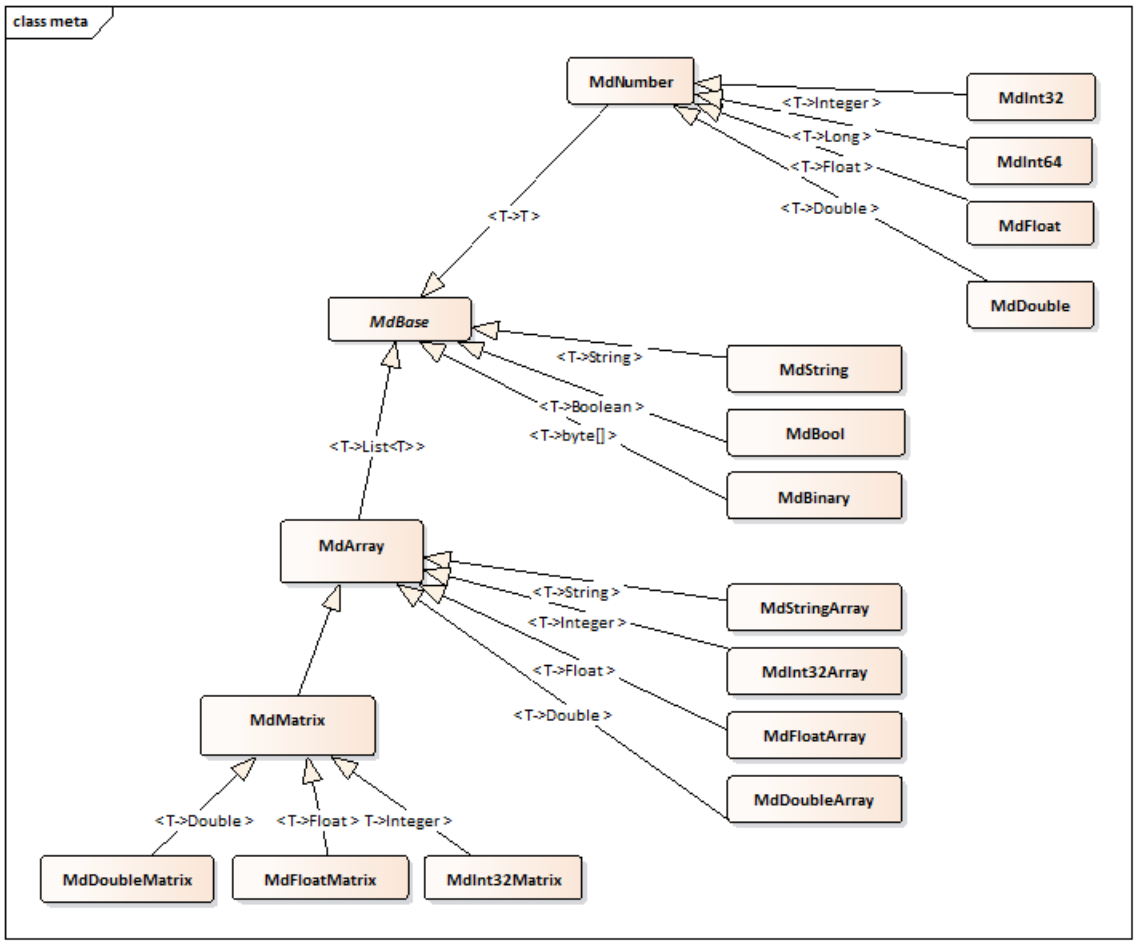


Figure 2-4 Metadata type system

To store array and matrix values, MdArray and MdMatrix derivates must be used.

The data for matrix is represented as a single array, with specified width (member field of the MdMatrix class), so matrix is represented as a continuous array of elements. The width attribute determines at which array element number the array should be cut into row to form a matrix. All matrices are row based; this means that the data is split into rows. When using default values, the width is set to 50.

Binary values are always stored as file references. Like for all the other types of data, the value for binary is set by using the setValue and retrieved by using the getValue method. On setValue method call, the data is written to a file and a reference is stored in the target configuration JSON file. On getValue method call, the file reference is resolved and data from the reference is retrieved. There is no limit specified for the binary data values.

Vector (array) and matrix values are stored as a string up to the limit of 100KB. All values exceeding this limit are stored as file references.

For more complex configurations, custom metadata classes can be defined. The classes can extend existing metadata classes.

The data of the config and metadata classes cannot be manipulated using the config client API.

Classes and metadata classes data shall be centrally stored in the remote DB. The config GUI is used to the manipulate the classes on the remote DB. For the development and testing purposes the classes can also be stored on local developer or test machines as YAML definitions.

13.2.2 Instances

Instances are actual value holders for the corresponding configuration and metadata classes. Instances are entities of JSON structure stored in local or remote DB and bind data to corresponding class defined fields. Class and instances member fields must be linked by name, otherwise exception CiiConfigTcBindException is thrown.

Metadata instances

Metadata instances define metadata for the config point. Metadata instances have the following properties:

- Metadata instances are identified by name. The names must be unique.
- Metadata instances can be created, updated and deleted.
- They can be stored to cache, local DB or remote DB. The asterisk "*" authority in metadata instances is not supported. The proper storage location for reading the reference directly form the TC is determined by the TC URI authority.
- Config point metadata is configured by metadata instance. Configuration instances only hold references to metadata instances. As metadata instances are not directly part of configuration instances, they are stored as separate entities.
- Metadata instances can be versioned.
- Set of default (predefined) metadata instances is provided (Appendix C). These default metadata instances cannot be saved or updated.

When saveTargetConfigWithMetadata method is used, metadata instances are automatically saved with target configuration. For other types of metadata operations, API contains methods that support operations just with metadata instances. These methods enable saving, updating and deleting metadata instances.

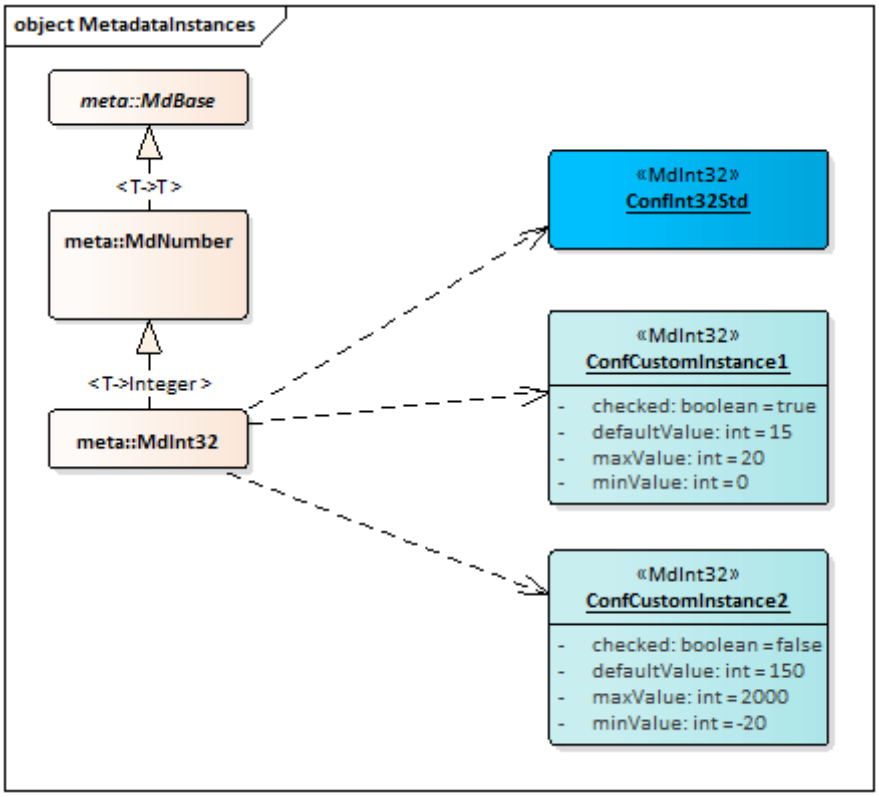


Figure 2-5 Metadata instances

Figure 2-5 shows the relationship between the metadata classes and metadata instances. The metadata class MdInt32 which derives from the MdNumber and root metadata class MdBase defines the structure for the metadata instances. Three metadata instances are created based on the MdInt32 class. The ConfInt32Std is a basic predefined metadata instance and has no special values defined. The custom metadata instances ConfCustomInstance1 and ConfCustomInstance2 have custom values set. The values of the class member fields (defaultValue, minValue, maxValue, checked, etc.) can be set for these instances.

Configuration instance

Target configuration (configuration instance) contains actual configuration data for the specific configuration class. The configuration instance is identified with URI location. Each URI defines the location from which the configuration instance is obtained. The authority part of URI defines the storage location. The following authorities are supported:

- Cache: ‘cii.config://cache/exampleconfig/c1’
- Local DB: ‘cii.config://local/exampleconfig/c1’
- Remote DB: ‘cii.config://remote/exampleconfig/c1’
- All locations: “cii.config://*/exampleconfig/c1”

Note: URI locations that start with number are not allowed. Examples of not valid URIs: cii.config:///*exampleconfig/1abc or cii.config:///*1exampleconfig/abc

The asterisk "*" authority can be used for retrieving configuration instance. When saving the instances, it is better to provide a specific location. When "*" authority is used, target configuration is obtained in the following order: cache, local DB, remote DB. In case there is no data stored under specified URI in the specific storage location, the location is skipped.

If there is no explicit version specified for retrieving the target configuration, the last existing version stored in the specific storage location will be retrieved. These versions can differ between the storage locations. As the config service can be used also without the presence of the remote DB, an additional setting was added to the client, to control the exception case.

The setting exceptionWhenRemoteNotAccessible can be set. Appendix G contains details on how to apply this setting to the client.

The following rules apply when retrieving the Target configuration with the "*" authority is used:

- exceptionWhenRemoteNotAccessible=true: If no remote DB is present – throw an exception.
- exceptionWhenRemoteNotAccessible=false: If no remote is present, check local and cache.

When saving or updating the Target configuration using the "*" authority, the versions between the different storage locations can become out of sync. In general, the following rules apply:

- saveTC("*"): save always creates a new version.
- updateTC("*"): always update the last version.
- updateTC(specificVersion) – updates specific version. It is always expected for a specific version on location (cache, local, remote) to exist. If it doesn't exist an exception is thrown.

As the configuration can be used without the remote DB, the following specifics apply:

- exceptionWhenRemoteNotAccessible=true:
  - saveTC("*"): remote DB is version maintainer. Max version off all the locations is retrieved, using the rule (cache|local|remote) + 1. This version is used to make new cache|remote|local Target configuration. This way versions between the locations are always in sync, but version holes can exist.
  - updateTC("*"): remote DB is version maintainer (the last version is retrieved from the remote DB). It is expected that the same version exists in local and cache and that this is the last version (important). If this is not the case, an exception is thrown.
  - updateTC(specificVersion): All versions are expected to exist. Updates for the specific version to all locations are made (cache|local|remote). If there is no specific version present in any of the DB, an exception is thrown.
- exceptionWhenRemoteNotAccessible=false:
  If remote DB is present, the behavior is the same as above. If not, the following actions are applied after the timeout:
- saveTC("*"): local DB is version maintainer. We get max version from (cache|local) + 1. We use

this version to make a new cache|local TC.

- updateTC("*"): local DB is version maintainer (the last version is retrieved from the local DB). It is expected that the same version exists in cache and local and that this is the last version. If this is not the case, an exception is thrown.
  - updateTC(specificVersion): All versions are expected to exist. Updates for the specific version to these locations are made (cahce|local). If there is no specific version present in any of the DB, an exception is thrown.

To use the configuration in Local DB mode, YAML definition files for classes and values must be created. The *config-tool* shall be used to generate language-specific classes and to deploy configuration data into the local DB.

When working with Remote DB, *config-gui* shall be used to manipulate the data. The remote DB also contains the config and metadata class definitions.

A cache can be used to store the data into memory. From the operations point of view, the cache acts the same as the local DB or remote DB and it is transparent to the client. The major difference to the other storage location is that cache is isolated per process and will not provide any permanent storage. The major benefit of using the cache is the fast storage and retrieval of configuration data. The cache is cleared when the process terminates.

For the configuration instances that use the config point fields of MdArray and MdBinary class, the data is not stored as a part of the JSON, but it is stored to a large storage location. The data to the large storage location is written as raw data in the big-endian order.

13.3 Prerequisites

In order to use the configuration system, the following packages must be built and installed:

- MAL, MAL ZPB (elt-mal)
- Client APIs (client-APIs)
- Config service (srv-config)
- Config tools (config-tools)
- Config GUI (config-gui)

Config service depends on the ElasticSearch. ElasticSearch server must be accessible to the config service. The default configuration assumes that ElasticSearch is available on http://ciielastichost:9200. Appendix H contains details on how to configure the config service.

For remote large data storage, the Hadoop HDFS system is used. HDFS system must be accessible in order to store large data files. The default configuration assumes that HDFS is accessible on http://ciihdfshost:9870/.

To use the remote database, the remote config service has to be installed and started.

See Error Handling and Configuration Transfer Document [2] for the instructions on how to install and configure the configuration system and its components. The document also contains a description of how to set the ElasticSearch and Hadoop hosts.

13.3.1 Local DB initialization

Local DB stores it's data into the **$INTROOT/localdb** folder. To use the default metadata instances used in the examples, the initial predefined metadata instances must be deployed:

```
config-tool init
```

13.3.2 Includes/Imports

For basic usage of the Config API the user needs the CiiConfigClient class and all generated binding configuration classes that will be used. If any custom metadata classes are used, the imports of these classes are also needed. Next, the language specific URI class is needed. Java also needs import of all the checked exceptions used. Following sections present the code imports and includes:

Java imports

```
import elt.config.client.CiiConfigClient;
import elt.config.exceptions.CiiConfigInitializationError;
import elt.config.exceptions.CiiConfigNoTcException;
import elt.config.exceptions.CiiConfigWriteDisabledException;
import elt.config.exceptions.CiiConfigSaveException;
import elt.config.exceptions.CiiConfigWriteDisabledException;
import elt.error.CiiInvalidTypeException;
import elt.error.CiiInvalidURIException;
import java.net.URI;
import java.net.URISyntaxException;
```

CPP Includes

```
#include <ciiDataProviderFactory.hpp>
#include <ciiLocalDataProvider.hpp>
#include <ciiCacheDataProvider.hpp>
#include <ciiRemoteDataProvider.hpp>
#include <ciiConfigApi.hpp>
```

Python imports

```
import os
import elt.config
```

13.4 Basic usage

This section presents basic usage examples. The examples in this section can be found in the *config-examples* project (M*ySimpleClient.java, MySimpleClient.py, mySimpleClient.cpp*). Refer to the *config-examples* project for details about files, folder structure and the WAF wscript files configuration. The examples project is build using the standard WAF build procedure:

```
waf distclean configure build install
```

To run the examples, use the provided executables (*run-java, cpp-config-sample-app, python MySimpleClient.py).*

The examples show the usage of Config tool (6.1) and Config GUI (7). The Config tool only deploys/undeploys data from YAML files to local DB. Config tool is intended to be used in the development process, where developer can quickly change definitions in YAML files on local machine, while Config GUI will work with remote with all the defined TCs. Config tool cannot be used to deploy data to remote DB. This can however be done with usage of Client API (the data can be read from local DB and stored to remote DB). For this reason, the examples show usage of both Config tools and Config GUI in for the same example.

Note: In case examples are run directly from *config-examples* project, make sure that all the needed local DB configuration instance files are deployed. The deploy commands are listed in *README.md* file of the project.

13.4.1 Example description

This section presents simple CII configuration example. All the YAML definitions in this example are also stored in the *yaml* folder of the *config-examples* project.
In the example we create a simple configuration with one string, one double and one integer type field. The simple configuration uses default metadata instances. The following YAML class definition is used (mySimpleClass.cdt.yaml):

```
---
configuration:
  configClassesConfiguration:
   classes:
    - name: MySimpleClass
      parent: CiiConfigClass
      __comment__: Personal config test class
      members:
      - name: testIntValue
        type: MdInt32
      - name: testDoubleValue
        type: MdDouble
```

(continues on next page)

(continued from previous page)

```
      - name: testStringValue
        type: MdString
```

The types names and language mappings for all types can be found in Appendix A. The syntax of YAML schema is explained in Appendix E. Not all words are allowed for the field names of config or metadata classes. The table in Appendix D contains all the reserved words.

In the example we create a sample application. The application is placed in the root of config-examples project. To prepare the directory structure we create the following subdirectories: java, cpp, cpp-app, cpp-pybindings, python and YAML:

```
mkdir java
mkdir cpp
mkdir cpp-app
mkdir cpp-pybindings
mkdir python
mkdir yaml
```

13.4.2 Creating a simple configuration using YAML definitions

In the simple class definition, we defined the structure of the configuration class. We store this definition in YAML file named *mySimpleClass.cdt.yaml* in the *yaml* directory. Based on this YAML definition we must first generate the classes used by the application:

```
config-tool generate -i yaml/mySimpleClass.cdt.yaml -o java/
config-tool generate -i yaml/mySimpleClass.cdt.yaml -o cpp/ -l cpp
config-tool generate -i yaml/mySimpleClass.cdt.yaml -o cpp-pybindings/ -l python␣
↪-ws
```

The sample code generated by the config-tool can be found in Appendix I.

Once the class is prepared, we must also define target configuration YAML. The target configuration will use the generated class MySimpleClass and default metadata instances. List of default metadata instances with internal types is listed in Appendix C. The following TC YAML is prepared:

```
config:
  instance:
    __comment__: Configuration for cryo cooler
    data:
      "@type": MySimpleClass
      fields:
        testStringValue:
           "@type": MdString
           metadataInstance: ConfStringStd
           metadataInstanceVersion: 1
           value: 'My string defined value'
        testIntValue:
```

(continues on next page)

(continued from previous page)

```
            "@type": MdInt32
            metadataInstance: ConfInt32Std
            metadataInstanceVersion: 1
            value: 155
         testDoubleValue:
            "@type": MdDouble
            metadataInstance: ConfDoubleStd
            metadataInstanceVersion: 1
            value: 155.33
```

We store the target configuration under the *testTC.cdi.yaml* file in *yaml* directory.

Note: Types defined in target configuration must be the same as types defined in the classes in order for the configuration to work properly. The depending on the language the JSON (de)serializers will throw exception with indication that (de)serialization could not be executed. For the types with embedded arrays, the types used in target configuration must have the vector types defined. Please refer to Appendix C for details.

To have the default metadata instances ready, we must first initialize the local DB:

```
config-tool init
```

Then we deploy the target configuration to local DB under **cii.config://mytest/mysampletc:**

```
config-tool deploy -i yaml/testTC.cdi.yaml -a cii.config://mytest/mysampletc
```

After the target config is deployed to local DB, we can start using the clients. For successful code builds, proper build dependencies must be set for each of the clients. Please check the *config-examples* project for details about the WAF dependencies.

Results of deploying *testTC.cdi.yaml* to local DB (JSON file) can be found in Listing 7-5.

Java client

For the Java sample, we will put our code in the java/src directory. We name our client sample code *mySimpleClient.java.* We use "*" authority for retrieving the stored target config. Data is stored under **cii.config://local/mytest/mysampletc** location.
Listing 4-1 contains the sample code. In the example we first read the data that was previously deployed in local DB. The client API will not allow save, update and delete operations. To enable these operations, we set the attribute setWriteEnabled to true. The data is then changed and stored under a different location. At the end the data is read in again (Listing 4-1).

Listing 4-1 Simple client Java sample code

```
public static void main(String[] args) throws Exception {

    URI myUri = new URI("cii.config://*/mytest/mysampletc");

    //Retrieve config from local or remote DB
    try(CiiConfigClient client=CiiConfigClient.getInstance()){
    MySimpleClass data = client.retrieveConfig(myUri).getData(MySimpleClass.
↪class);
    System.out.println("My double value:" + data.getTestDoubleValue().
↪getValue());
    System.out.println("My int value:" + data.getTestIntValue().getValue());
    System.out.println("My string value:" + data.getTestStringValue().
↪getValue());


    //Store changed data into new TC location -- local DB
    client.setWriteEnabled(true);
    URI localUri = new URI("cii.config://local/mytest/mysampletc2");
    data.getTestDoubleValue().setValue(121442242.224242424);
    CiiConfigClient.getInstance().saveTargetConfig(localUri, data);

    //Read changed data
    MySimpleClass changedData = client. retrieveConfig(localUri).
↪getData(MySimpleClass.class);
    System.out.println("My double value:" + changedData.getTestDoubleValue().
↪getValue());
    System.out.println("My int value:" + changedData.getTestIntValue().
↪getValue());
    System.out.println("My string value:" + changedData.getTestStringValue().
↪getValue());
  }
 }
}
```

CPP client

Listing 4-2 contains a sample CPP client code. The code file is named *mySimpleClient*.cpp and stored in the *cpp-app* folder. The code follows the same flow as the Java code.

Listing 4-2 Simple client CPP sample code

```
int main(int ac, char* av[]) {

::elt::config::classes::MySimpleClass mySimpleClassDataCh;

std::shared_ptr<::elt::config::classes::MySimpleClass> retrievedData =
      elt::config::CiiConfigClient::getConfigData<
          ::elt::config::classes::MySimpleClass>(
```

(continues on next page)

(continued from previous page)

```
          elt::mal::Uri("cii.config://*/mytest/mysampletc"));

std::cout << "My int value:" << retrievedData->get_testIntValue() << std::endl;
std::cout << "My double value:" << retrievedData->get_testDoubleValue() <<␣
↪std::endl;
std::cout << "My string value:" << retrievedData->get_testStringValue() <<␣
↪std::endl;

//Store changed data into new TC location -- local DB
::elt::config::CiiConfigClient::setWriteEnabled(true);
mySimpleClassDataCh.set_testStringValue("Changed string value");
mySimpleClassDataCh.set_testIntValue(retrievedData->get_testIntValue());
mySimpleClassDataCh.set_testDoubleValue(retrievedData->get_testDoubleValue());

int version = elt::config::CiiConfigClient::saveTargetConfig(
      elt::mal::Uri("cii.config://local/mytest/mysampletc2"),␣
↪mySimpleClassDataCh);

//Read changed data
std::shared_ptr<::elt::config::classes::MySimpleClass> retrievedData2 =
      elt::config::CiiConfigClient::getConfigData<
          ::elt::config::classes::MySimpleClass>(
          elt::mal::Uri("cii.config://local/mytest/mysampletc2"));


std::cout << "My int value:" << retrievedData2->get_testIntValue() << std::endl;
std::cout << "My double value:" << retrievedData2->get_testDoubleValue() <<␣
↪std::endl;
std::cout << "My string value:" << retrievedData2->get_testStringValue() <<␣
↪std::endl;

}
```

Python client

Listing 4-3 contains a sample Python code. The code file is named *mySimpleClient.py* and stored in the *python* folder. The code follows the same flow as the CPP code.

Listing 4-3 Simple client Python sample code

```
import os
import elt.config
from MySimpleClassPyB import MySimpleClass


uri = elt.config.Uri("cii.config://*/mytest/mysampletc")
uri2 = elt.config.Uri("cii.config://local/mytest/mysampletc2")
```

(continues on next page)

(continued from previous page)

```
client = elt.config.CiiConfigClient

#Retrieve config configuration
retrieved = client.retrieve_config(uri, -1)

print("My string value:" + retrieved.get_testStringValue())
print("My integer value: {}".format(retrieved.get_testIntValue()))
print("My double value: {}".format(retrieved.get_testDoubleValue()))

#Store changed data into new TC location -- local DB
client.set_write_enabled(True)

retrievedNew = MySimpleClass.get_new_node_instance()
retrievedNew.set_testStringValue("New test value")
retrievedNew.set_testIntValue(retrieved.get_testIntValue())
retrievedNew.set_testDoubleValue(retrieved.get_testDoubleValue())

version = client.save_target_config(uri2, retrievedNew)
retrieved2 = client.retrieve_config(uri2, -1)

print("My string value:" + retrieved2.get_testStringValue())
print("My integer value: {}".format(retrieved2.get_testIntValue()))
print("My double value: {}".format(retrieved2.get_testDoubleValue()))
```

13.4.3 Creating a simple configuration using GUI

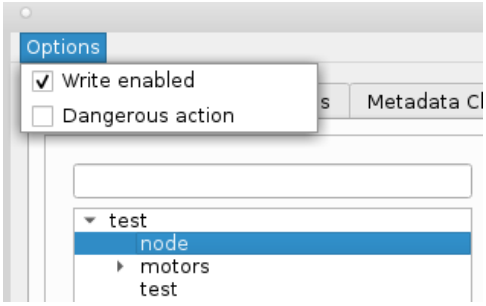
In this section GUI is used to define classes and target configuration on remote DB.

We first start the configuration GUI:

```
config-gui
```

Make sure that GUI has write tick-box enabled (Figure 4-1):

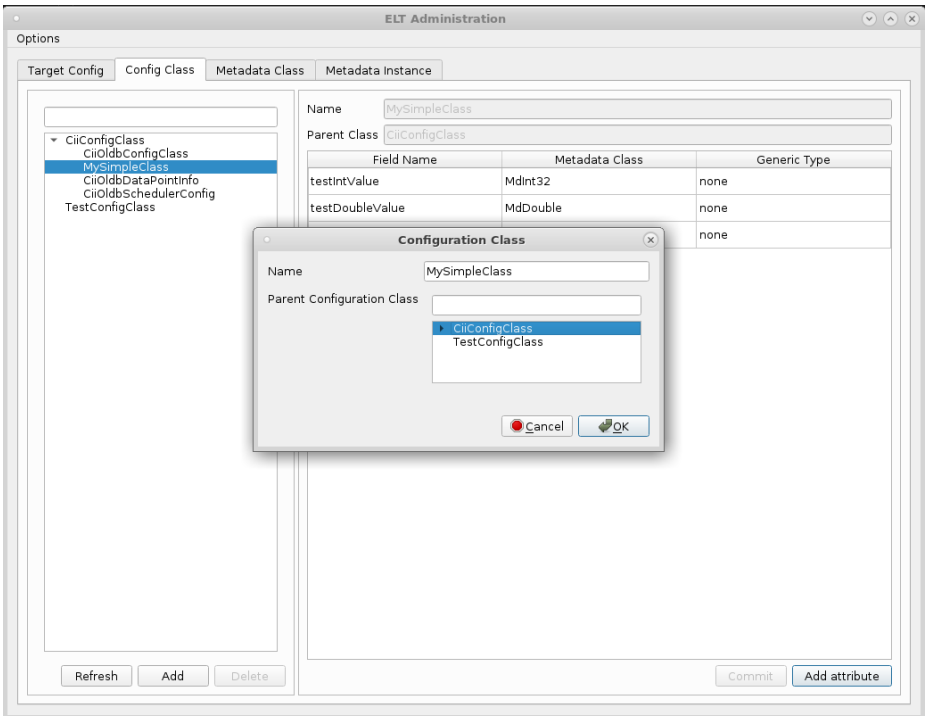
Figure 4-1 Write enabled tick box



Creating a simple class in GUI

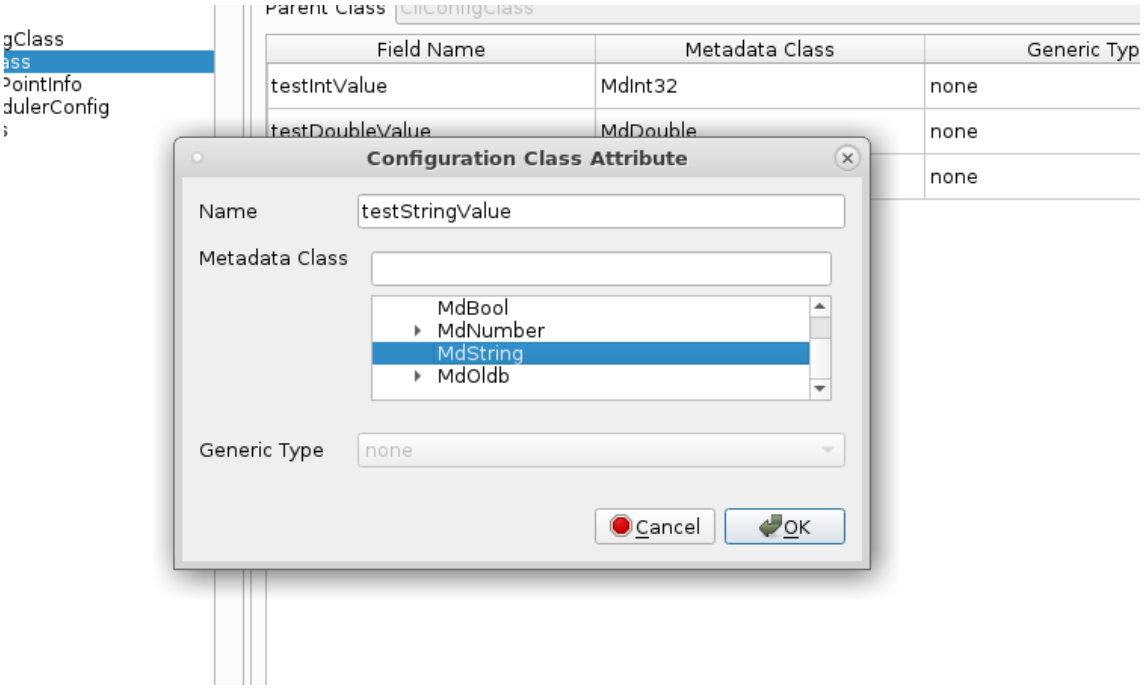
Click on the Configuration class tab, and click *Add* button to add a configuration class (Figure 4-2). Select the *CiiConfigClass* to be the parent class, and name the class MySimpleClass.

Figure 4-2 Adding configuration class in GUI



After the class is created, we add attributes to the class. To add attributes (members), we click on the *Add attribute* button and fill in the details of every class attribute (Figure 4-3). We use the same names and types as in the YAML class definition in section 4.1.

Figure 4-3 Adding class members in GUI

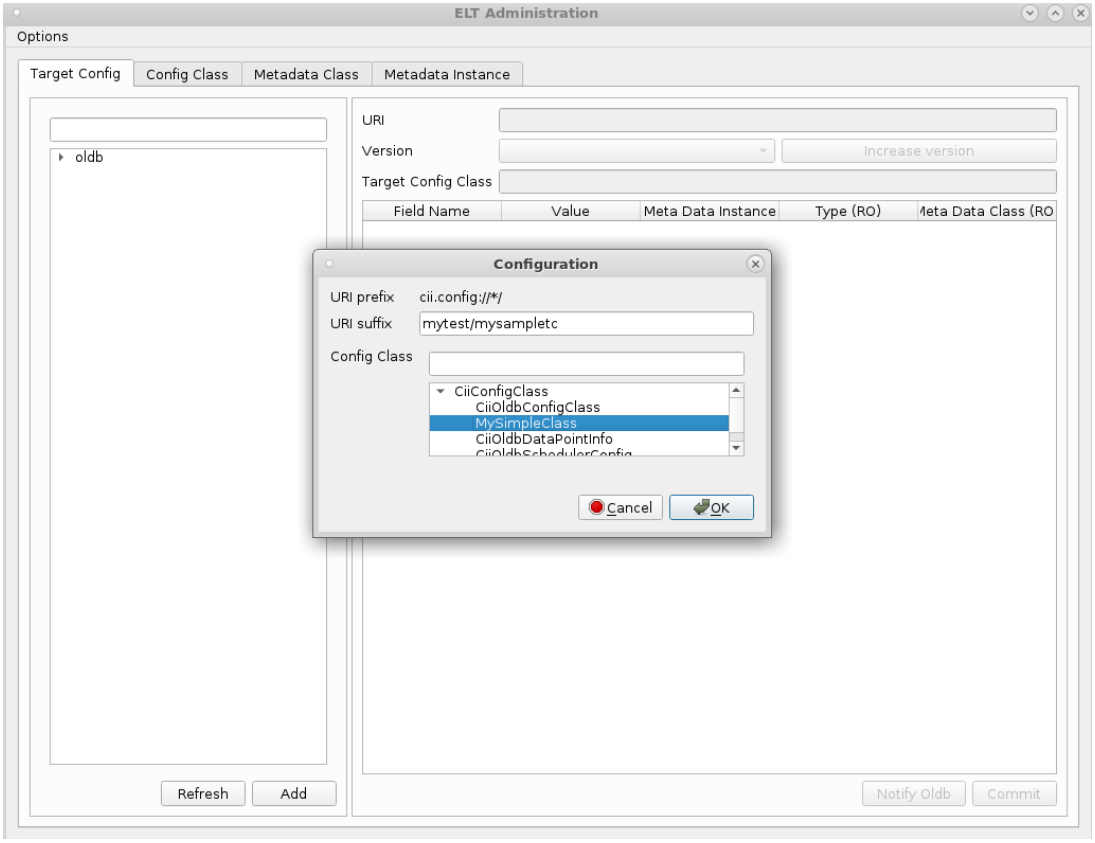


After we added all the attributes (*testIntValue*, *testDoubleValue, testStringValue),* definition changes must be confirmed and propagated to remote DB by clicking on the *Commit* button.

Creating a target configuration for simple class in GUI

To create a TC we click on the Target Config tab and then the Add button. We choose the proper Configuration class and define a URI suffix that corresponds to the URI used when deploying to the local DB, for example **mytest/mysampletc**. (Figure 4-4)

Figure 4-4 Creating target configuration



After the target configuration is created, we must refresh the tree by clicking the “Refresh button”. Next, we bind the metadata instances used by the TC to the member fields. To select the metadata instance, we click on the cells (Figure 4-6) under Meta Data Instance column. For every class member field name, we choose metadata instance (Figure 4-5). Predefined default metadata instances are used (ConfInt32Std, ConfDoubleStd, ConfStringStd).

Figure 4-5 Target configuration definition in GUI

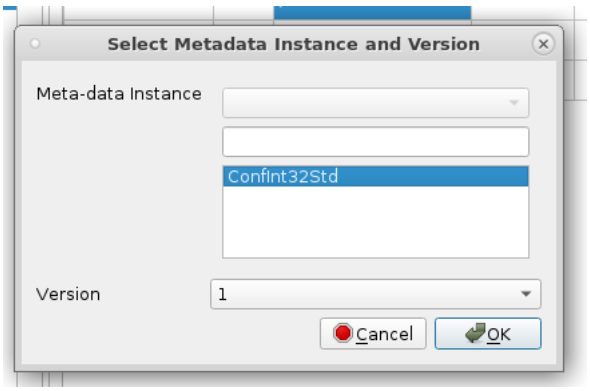
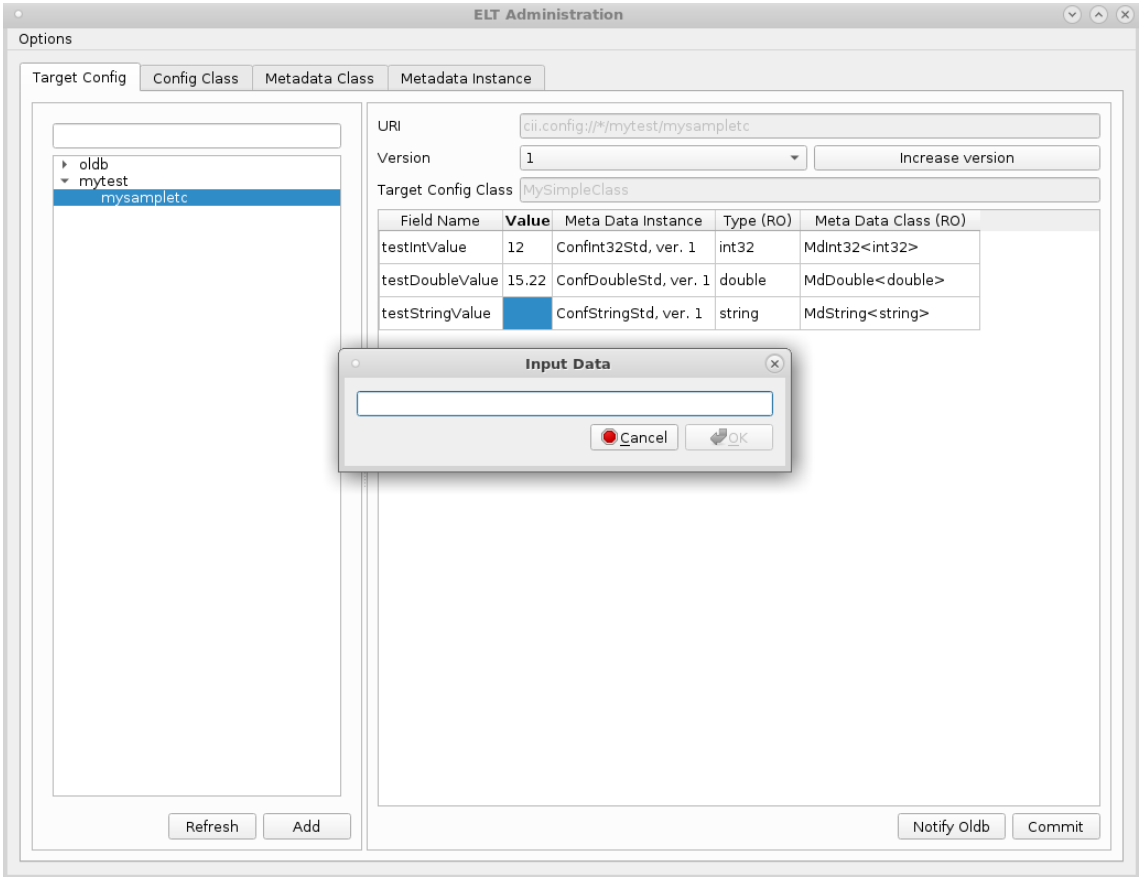


Figure 4-6 Target configuration definition in GUI



After metadata instances are set, we set the values for every member (cells under Value column), and confirm the changes by clicking on the *Commit* button (Figure 4-6).

Generating classes from the classes defined in GUI

To generate classes defined in GUI (stored in remote DB), the config tool is called without an input argument:

```
config-tool generate -o java/
config-tool generate -o cpp/ -l cpp
config-tool generate -o cpp-pybindings/ -l python -ws
```

To read the data with the client, we use the same code as shown in the previous example (Listing 4-1, Listing 4-2, Listing 4-3). To read the data from the remote DB, the “remote” authority must be used. In case “*” authority is used, all data will first be read from the local DB. Only if the data is not present in local DB, the remote DB will be used.

13.4.4 CRUD operations

The following section presents the code for creating, reading, updating and deleting metadata and configuration instances. The examples for all languages follow the same flow, and can be easily understood from the code samples. The examples first show the operations on the metadata instance, followed by operation on target configuration. The TC is saved under the cii.config://remote/mytest/my_test_instance URI. Config API also supports the listing of config points stored under certain path. To get all the children, the getChildren method can be used. Refer to Appendix A for details about the API.

Note: When setting only one attribute of a configuration programmatically, the metadata for the other attributes are not set automatically. It is expected that all the fields of the configuration class are set by user.

The same MySimpleClass as it was used in section 4.2 is also used in the following examples.

Java CRUD

Listing 4-4 Java metadata CRUD example

```
//Metadata CRUD example
final String mdiName = "MyCustomMetaInstanceName";
MyCustomMetaClass<Integer> customMetaClass = new MyCustomMetaClass<>(mdiName,
 ↪"This is my custom metadata instance name",
                                  123, false, "m/s", 9);
URI metadataLocal = new URI("cii.config://local");
//Save metadata under local DB with name MyCustomMetaInstanceName
int version2 = client.saveMetadata(metadataLocal, customMetaClass);

//Retrieve and update the same version
MyCustomMetaClass<Integer> customMetaClassChange = client.
 ↪retrieveMetadata(metadataLocal, mdiName, -1,  MyCustomMetaClass.class);
System.out.println("Metadata int:" + customMetaClassChange.getMetaOfInt());

customMetaClassChange.setMetaOfInt(15);
client.updateMetadata(metadataLocal, mdiName, version2, customMetaClassChange);

//delete the metadata
System.out.println("Deleting metadata: " + mdiName + ", version: " + version2);
client.deleteMetadata(metadataLocal, mdiName, version2);
```

Listing 4-5 Java target config CRUD example

```
//TC CRUD example
//Create another instance of MySimple class and store it to remote
MySimpleClass newSimpleInstance = new MySimpleClass();
newSimpleInstance.setTestDoubleValueValue(12.0d);
newSimpleInstance.setTestIntValueValue(100);
```

(continues on next page)

(continued from previous page)

```
newSimpleInstance.setTestStringValueValue("My test string");

URI newInstanceUri = new URI("cii.config://remote/mytest/my_test_instance");
int version1 = client.saveTargetConfig(newInstanceUri, newSimpleInstance);

//Retrieve and update the same version
MySimpleClass newSimpleInstanceChange = client.retrieveConfig(newInstanceUri).
↪getData(MySimpleClass.class);
newSimpleInstanceChange.setTestStringValueValue("My test string changed");
client.updateConfig(newInstanceUri, version1, newSimpleInstanceChange);

System.out.println("My string value:" + newSimpleInstanceChange.
↪getTestStringValue().getValue());

//Delete TC
client.deleteConfig(newInstanceUri, -1);
```

CPP CRUD

Listing 4-6 CPP metadata CRUD example

```
//Metadata CRUD example
std::string mdiName("MyCustomMetaInstanceName");
std::shared_ptr<elt::config::classes::meta::MyCustomMetaClass<std::int32_t>>␣
↪customMetaClass(
      elt::config::CiiDataPointMetadataFactory::getNewMetadataInstance
↪<elt::config::classes::meta::MyCustomMetaClass<std::int32_t>>(mdiName));
customMetaClass->set_metaOfBool(false);
customMetaClass->set_metaOfString("m/s");
customMetaClass->set_metaOfInt(9);

//Save metadata under local DB with name MyCustomMetaInstanceName
int version2 = elt::config::CiiConfigClient::saveMetadata("local",␣
↪*customMetaClass);

//Retrieve and update the same version
std::shared_ptr<elt::config::classes::meta::MyCustomMetaClass<std::int32_t>>␣
↪customMetaClassChange =
      std::dynamic_pointer_cast<elt::config::classes::meta::MyCustomMetaClass
↪<std::int32_t>>(elt::config::CiiConfigClient::retrieveMetadata("local",␣
↪mdiName));

std::cout << "Metadata int:" << customMetaClassChange->get_metaOfInt() <<␣
↪std::endl;
customMetaClassChange->set_metaOfInt(15);

elt::config::CiiConfigClient::updateMetadata("local", version2,␣
↪*customMetaClassChange);
```

(continues on next page)

(continued from previous page)

```
//delete the metadata
std::cout << "Deleting metadata: " <<  mdiName <<  ", version: " << version2 <<␣
↪std::endl;
elt::config::CiiConfigClient::deleteMetadata("local", mdiName, version2);
```

Listing 4-7 CPP target config CRUD example

```
//TC CRUD example
//Create another instance of MySimple class and store it to remote
std::shared_ptr<::elt::config::classes::MySimpleClass> newSimpleInstance =
      CiiNodeFactory::getNewNodeInstance<::elt::config::classes::MySimpleClass>
↪();
newSimpleInstance->set_testStringValue("My test string");
newSimpleInstance->set_testIntValue(100);
newSimpleInstance->set_testDoubleValue(12.0);

elt::mal::Uri newInstanceUri("cii.config://remote/mytest/my_test_instance");
int version1 = elt::config::CiiConfigClient::saveTargetConfig(newInstanceUri,␣
↪*newSimpleInstance);

//Retrieve and update the same version
std::shared_ptr<::elt::config::classes::MySimpleClass> newSimpleInstanceChange =
elt::config::CiiConfigClient::getConfigData
↪<::elt::config::classes::MySimpleClass>(newInstanceUri);

newSimpleInstanceChange->set_testStringValue("My test string changed");
elt::config::CiiConfigClient::updateConfig(newInstanceUri, version1,␣
↪*newSimpleInstanceChange);

std::cout << "My string value:" << newSimpleInstanceChange->get_
↪testStringValue() << std::endl;

//Delete TC
elt::config::CiiConfigClient::deleteConfig(newInstanceUri, -1);
```

Python CRUD

Listing 4-8 Python metadata CRUD example

```
#Metadata CRUD example
mdiName = "MyCustomMetaInstanceName"
customMetaClass = MyCustomMetaClassINT32.get_new_metadata_instance(mdiName)
customMetaClass.set_metaOfBool(False);
customMetaClass.set_metaOfString("m/s");
customMetaClass.set_metaOfInt(9);
```

(continues on next page)

(continued from previous page)

```
#Save metadata under local DB with name MyCustomMetaInstanceName
version2 = client.save_metadata("local", customMetaClass);

#Retrieve and update the same version
customMetaClassChange = client.retrieve_metadata("local", mdiName)
customMetaClassChange = MyCustomMetaClassINT32.cast(customMetaClassChange)

#Cannot print as wrong type is returned.
print("Metadata int: %i" % customMetaClassChange.get_metaOfInt())
customMetaClassChange.set_metaOfInt(15)
print("Metadata int changed: %i" % customMetaClassChange.get_metaOfInt())
client.update_metadata("local", version2, customMetaClassChange);

#delete the metadata
print("Deleting metadata: %s, version: %i" % (mdiName, version2))
client.delete_metadata("local", mdiName, version2);
```

Listing 4-9 Python target config CRUD example

```
#TC CRUD example
#Create another instance of MySimple class and store it to remote
newSimpleInstance = MySimpleClass.get_new_node_instance()
newSimpleInstance.set_testStringValue("My test string")
newSimpleInstance.set_testIntValue(100)
newSimpleInstance.set_testDoubleValue(12.0)

newInstanceUri = elt.config.Uri("cii.config://remote/mytest/my_test_instance")
version1 = client.save_target_config(newInstanceUri, newSimpleInstance)

#Retrieve and update the same version
newSimpleInstanceChange = client.retrieve_config(newInstanceUri)

newSimpleInstanceChange.set_testStringValue("My test string changed");
client.update_config(newInstanceUri, version1, newSimpleInstanceChange);

print("My integer value: {}".format(newSimpleInstanceChange.get_
↪testStringValue()))

#Delete TC
client.delete_config(newInstanceUri, -1);
```

13.4.5 Using the cache operations

The following examples present the usage of cache operations. From the operations point of view, the cache acts the same as the local DB or remote DB and it is transparent to the client. The cache is not permanent storage, as it lives with the process. The cache can be used for inter-process fast storing/retrieval of Target Configurations.

The example first copies Target Configuration from the remote DB (cii.config://remote/mytest/my_test_instance), then it stores it to the cache. Then the values are changed and stored back to same location in cache (cii.config://cache/mytest/my_test_instance). Finally, these values are stored back to the original remote DB location.

In the example, the same path is used for remote DB and cache. If the same path is used, the Target Configuration can be retrieved using the "*" authority. It will first check the cache, then the local and remote DB. The same Configuration class data can be stored to any URI, as there is no limitation for cache to be on the same path as remote and local DB.

Java cache example

Listing 4-10 Java cache example

```
//The cache workflow -- data in cache only lives as long as the process is alive
//Retrieve the TC from the remote, and store it to cache.

URI newInstanceUri = new
                       URI("cii.config://remote/mytest/my_test_instance");
URI cacheInstance = new URI("cii.config://cache/mytest/my_test_instance");
  client.saveTargetConfig(cacheInstance,
      client.retrieveConfig(newInstanceUri).getData(MySimpleClass.class));

MySimpleClass dataFromCache =
  client.retrieveConfig(cacheInstance).getData(MySimpleClass.class);

System.out.println("Original cache data: " +
  dataFromCache.getTestIntValueValue() + ", " +
  dataFromCache.getTestStringValueValue());
//manipulate cache data
dataFromCache.setTestIntValueValue(15);
dataFromCache.setTestStringValueValue("My string value");
System.out.println("Changed cache data: " +
  dataFromCache.getTestIntValueValue() + ", " +
  dataFromCache.getTestStringValueValue());

//store data into cache
  client.saveTargetConfig(cacheInstance, dataFromCache);

//retrieve data form cache and store it to remote.
MySimpleClass dataFromCacheChanged =
```

(continues on next page)

(continued from previous page)

```
  client.retrieveConfig(cacheInstance).getData(MySimpleClass.class);
System.out.println("Retrieved changed cache data: " +
  dataFromCacheChanged.getTestIntValueValue() + ", " +
  dataFromCacheChanged.getTestStringValueValue());

//save the changed data to remote
client.saveTargetConfig(newInstanceUri, dataFromCacheChanged);


//Delete TC
client.deleteConfig(newInstanceUri, -1);
```

CPP Cache

Listing 4-11 CPP cache example

```
//The cache workflow -- data in cache only lives as long as the process is alive
//Retrieve the TC from the remote, and store it to cache.
elt::mal::Uri newInstanceUri
                          ("cii.config://remote/mytest/my_test_instance");
elt::mal::Uri cacheInstance("cii.config://cache/mytest/my_test_instance");

//as there are no metadata instances in cache, they must be saved
elt::config::CiiConfigClient::SaveTargetConfig(cacheInstance,
  *elt::config::CiiConfigClient::GetConfigData<::elt::config::classes::
  MySimpleClass>(newInstanceUri));

std::shared_ptr<::elt::config::classes::MySimpleClass> dataFromCache =
  elt::config::CiiConfigClient::GetConfigData<::elt::config::classes::
  MySimpleClass>(cacheInstance);

std::cout << "Original cache data: " << dataFromCache->get_testIntValue()
   << ", " << dataFromCache->get_testStringValue()  << std::endl;
//manipulate cache data
dataFromCache->set_testIntValue(15);
dataFromCache->set_testStringValue("My string value");

//store data into cache
elt::config::CiiConfigClient::SaveTargetConfig(cacheInstance, *dataFromCache);

//retrieve data form cache and store it to remote.
std::shared_ptr<::elt::config::classes::MySimpleClass> dataFromCacheChanged =
  elt::config::CiiConfigClient::GetConfigData<::elt::config::classes::
  MySimpleClass>(cacheInstance);
std::cout << "Changed cache data: " << dataFromCacheChanged->get_testIntValue() <
↪< ", " << dataFromCacheChanged->get_testStringValue()  << std::endl;
```

(continues on next page)

(continued from previous page)

```
//save the changed data to remote
elt::config::CiiConfigClient::SaveTargetConfig(newInstanceUri,␣
↪*dataFromCacheChanged);

//Delete TC
elt::config::CiiConfigClient::DeleteConfig(newInstanceUri, -1);
```

Python cache example

Listing 4-12 Python cache example

```
#The cache workflow -- data in cache only lives as long as the process is alive
#Retrieve the TC from the remote, and store it to cache.
newInstanceUri = elt.config.Uri("cii.config://remote/mytest/my_test_instance")
cacheInstance = elt.config.Uri("cii.config://cache/mytest/my_test_instance")
client.save_target_config(cacheInstance, client.retrieve_config(newInstanceUri))

dataFromCache = client.retrieve_config(cacheInstance)

print("Original cache data: %i, %s" % (dataFromCache.get_testIntValue(),␣
↪dataFromCache.get_testStringValue()))

#manipulate cache data
dataFromCache.set_testStringValue("My string value")
dataFromCache.set_testIntValue(15)

#store data into cache
client.save_target_config(cacheInstance, dataFromCache)

#retrieve data form cache and store it to remote.
dataFromCacheChanged = client.retrieve_config(cacheInstance)
print("Changed cache data: %i, %s" % (dataFromCacheChanged.get_testIntValue(),␣
↪dataFromCacheChanged.get_testStringValue()))

#save the changed data to remote
client.save_target_config(newInstanceUri, dataFromCacheChanged)

#Delete TC
client.delete_config(newInstanceUri, version1)
```

13.5 Advanced usage

This section presents advanced usage examples. The examples in this section are suffixed with advanced in the config-examples project (M*yAdvancedClient.java, MyAdvancedClient.py, myAdvancedClient.cpp*). The examples project is build using the standard WAF build procedure:

```
waf distclean configure build install
```

To run the examples, use the provided executables (*run-java-advanced, cpp-config-sample-app-advanced, python MyAdvancedClient.py).*

Note: In a case when examples are run directly from *config-examples* project, make sure that all the needed local DB configuration instance files are deployed. The deploy commands are listed in *Readme.md* file of the project.

13.5.1 Custom metadata class

In this example we create a custom metadata class, which will be used as a member field in the configuration class. Listing 5-1 shows the YAML code of the metadata class definition. We name the custom metadata class MyCustomMetaClass and save the metadata class definition as *myCustomMetadata.cdt.yaml* in the folder named *yaml*. The class derives from the MdNumber. This class has generic type set to *T*. This means that the actual type of the configuration class member field can be set at the class configuration class definition.

Listing 5-1 MyCustomMetaClass definition

```
---
configuration:
  metadataClassesConfiguration:
   classes:
    - name: MyCustomMetaClass
      parent: MdNumber
      __comment__: Custom metadata class
      genericType: T
      members:
      - name: metaOfBool
        type: BOOLEAN
      - name: metaOfString
        type: STRING
      - name: metaOfInt
        type: INT32
```

The class has 3 metadata fields defined: metaOfBool (type Boolean), metaOfString (type string), metaOfInt (type int32). All other metadata fields that can be used by the metadata instances (minValue, maxValue, checked, defaultValue, ...) are already defined by the MdNumber and its parent MdBase.

Next, we create a configuration class, which will use this custom metadata class. We name the

class MyClassWithCustomMeta and save the configuration class definition as *myClassWithCustomMeta.cdt.yaml* in the folder named *yaml*. The class derives from the parent configuration class CiiConfigClass and has only one Config point defined: intPointWithCustomMeta. As the MyCustomMetaClass definition has the type defined as T, we need to set the generic type of the member field to proper type. We set the intPointWithCustomMeta to be INT32. Listing 5-2 shows the YAML definition of the class.

Listing 5-2 MyClassWithCustomMeta definition

```
---
configuration:
  configClassesConfiguration:
   classes:
    - name: MyClassWithCustomMeta
      parent: CiiConfigClass
      __comment__: Class with custom metadata
      members:
      - name: intPointWithCustomMeta
        type: MyCustomMetaClass
        genericType: INT32
```

After the configuration classes are defined, we must create YAML files which will define the metadata instance and TC values. Listing 5-3 shows the YAML definition which defines the values of the custom metadata instance member fields for the MyCustomMetaClass. We name the definition *myCustomMDI.mdi.yaml* and save it in the *yaml* folder. The definition sets the values and sets the type. As metadata default value can be of any number type, we must also define the type with @genType directive in the metadata instances definition file.

Listing 5-3 Definition of custom metadata instance (myCustomMDI)

```
metadata:
  instances:
  - "@type": MyCustomMetaClass
    "@name": myCustomMDI
    "@genType": INT32
    comment: "Integer MDI with values"
    metaOfBool: "true"
    metaOfString: "The string metadata example"
    metaOfInt: "12"
```

Finally, we must define the TC values file. We name this file *simpleCustomMetaTC.cdi.yaml* and save it in the *yaml* folder. The file contains the values for linking the metadata instance *myCustomMDI* to the config point *intPointWithCustomMeta*. The file also sets the value of the point to 13. Listing 5-4 shows the file definition.

Listing 5-4 Definition of target config with custom meta data (simpleCustomMetaTC)

```
config:
  instance:
    __comment__: Configuration for cryo cooler
```

(continues on next page)

(continued from previous page)

```
    data:
      "@type": MyClassWithCustomMeta
      fields:
        intPointWithCustomMeta:
           "@type": MyCustomMetaClass
           "@genType": INT32
           metadataInstance: myCustomMDI
           metadataInstanceVersion: 1
           value: 13
```

Once all the classes are defined, we must generate the metadata binding classes. We generate the classes for all three programming languages with the config tool:

```
config-tool generate -i yaml/myCustomMetadata.mdt.yaml -t metadata -o java/
config-tool generate -i yaml/myCustomMetadata.mdt.yaml -t metadata -o cpp/ -l cpp
config-tool generate -i yaml/myCustomMetadata.mdt.yaml -t metadata -o cpp-
↪pybindings/ -l python
```

Next, we generate the configuration classes with our custom metadata:

```
config-tool generate -i yaml/myClassWithCustomMeta.cdt.yaml -o java/
config-tool generate -i yaml/myClassWithCustomMeta.cdt.yaml -o cpp/ -l cpp
config-tool generate -i yaml/myClassWithCustomMeta.cdt.yaml -o cpp-pybindings/ -
↪l python -ws
```

After the language bindings for the configuration and metadata classes are generated, the produced code files can be built. Please note that *config-tool* generator will not generate all the WAF wscript files. For the proper build, the wscript files with dependencies have to be added to the *java*, *cpp*, and *cpp-pybindings* directory. The python bindings will assume, that CPP code and python bindings are located in *cpp* and *cpp-pybindings* directory respectively. The config tool provides option "-ws" to generate wscript file for generated python binding modules.

To retrieve the data from the local DB, the YAML value files have to be deployed. The following example shows the CLI commands to deploy the metadata instance values and the configuration instance values. The target configuration is deployed under the **cii.config://mytest/myCustomMetaTC**:

```
config-tool deploy -i yaml/myCustomMDI.mdi.yaml -t metadata
config-tool deploy -i yaml/simpleCustomMetaTC.cdi.yaml -a cii.config://mytest/my_
↪custom_meta_tc
```

Note: Config-tool can be used without the authority (*cii.config://mytest/)* or with the "*" authority (*cii.config:///*mytest/).* In case of "*" is used, the tool will discard authority and in all cases deploy data to local DB.

Finally, we create the Java (Listing 5-5), CPP (Listing 5-6), and Python (Listing 5-7) sample applications to read the metadata values.

Listing 5-5 Java custom metadata class code example

```
public class mySimpleClient{

public static void main(String[] args) throws Exception {


    URI myUri = new URI("cii.config://local/mytest/my_custom_meta_tc");

    try(CiiConfigClient client=CiiConfigClient.getInstance()){

    MyClassWithCustomMeta data = client.retrieveConfig(myUri).
↪getData(MyClassWithCustomMeta.class);
    System.out.println("The value:" + data.getIntPointWithCustomMeta().
↪getValue());
    System.out.println("Metadata one:" + data.getIntPointWithCustomMeta().
↪getMetaOfBool());
    System.out.println("Metadata two:" + data.getIntPointWithCustomMeta().
↪getMetaOfString());
    System.out.println("Metadata three:" + data.getIntPointWithCustomMeta().
↪getMetaOfInt());

  }
}

}
```

Listing 5-6 CPP custom metadata class code example

```
int main(int ac, char* av[]) {

::elt::config::classes::MyClassWithCustomMeta MyClassWithCustomMetaCh;

std::shared_ptr<::elt::config::classes::MyClassWithCustomMeta> retrievedData =
      elt::config::CiiConfigClient::getConfigData<
          ::elt::config::classes::MyClassWithCustomMeta>(
          elt::mal::Uri("cii.config://local/mytest/my_custom_meta_tc"));

std::cout << "The value:" << retrievedData->get_intPointWithCustomMeta() <<␣
↪std::endl;
std::cout << "Metadata one:" << retrievedData->get_metadata_
↪intPointWithCustomMeta()->get_metaOfBool() << std::endl;
std::cout << "Metadata two:" << retrievedData->get_metadata_
↪intPointWithCustomMeta()->get_metaOfString() << std::endl;
std::cout << "Metadata three:" << retrievedData->get_metadata_
↪intPointWithCustomMeta()->get_metaOfInt() << std::endl;

}
```

Listing 5-7 Python custom metadata class code example

```
uriCm = elt.config.Uri("cii.config://*/mytest/my_custom_meta_tc")

#Retrieve config configuration
retrievedCm = client.retrieve_config(uriCm, -1)

print("The value: %i" % retrievedCm.get_stringPointWithCustomMeta())
print("The first metadata value : %s" % retrievedCm.get_metadata_
↪stringPointWithCustomMeta().get_metaOfBool())
print("The second metadata value: %s" % retrievedCm.get_metadata_
↪stringPointWithCustomMeta().get_metaOfString())
print("The third metadata value: %s" % retrievedCm.get_metadata_
↪stringPointWithCustomMeta().get_metaOfInt())
```

13.5.2 Using referenced classes

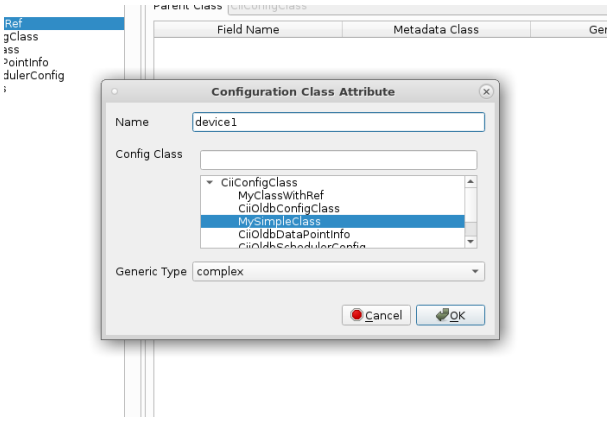
In this example we create a class that has a field which links a referenced class. This means that the filed will act as a placeholder for all class members of the class that is defined as referenced. The members can be accessed using the "." dot keyword in the instance YAML definition. The class which we are refereeing to must already exist or be defined in the same YAML definition. The class uses **isRef** directive to identify the reference type in the configuration class YAML definition.

In the example we name the class MyClassWithRef. The following class definition YAML is prepared (*configClassWithRef.cdt.yaml*):

```
configuration:
  configClassesConfiguration:
   classes:
    - name: MyClassWithRef
      parent: CiiConfigClass
      __comment__: Configuration class with referenced class
      members:
      - name: device1
        type: MySimpleClass
        isRef: true
```

The configuration expects that referenced class, MySimpleClass is already present in the source code. To create the same reference field in GUI, the Generic Type "complex" must be used (Figure 5-1).

Figure 5-1 Adding referenced class in GUI



Once the class is prepared, we must define target configuration YAML. The target configuration will use the generated class MyClassWithRef. The following TC YAML is prepared (testRefTc.cdi.yaml):

```
config:
  instance:
    __comment__: Configuration for cryo cooler
    data:
      "@type": MyClassWithRef
      fields:
        device1.testStringValue:
           "@type": MySimpleClass.MdString
           metadataInstance: ConfStringStd
           metadataInstanceVersion: 1
           value: 'Ref string defined value'
        device1.testIntValue:
           "@type": MySimpleClass.MdInt32
           metadataInstance: ConfInt32Std
           metadataInstanceVersion: 1
           value: 100
        device1.testDoubleValue:
           "@type": MySimpleClass.MdDouble
           metadataInstance: ConfDoubleStd
           metadataInstanceVersion: 1
           value: 200.33
```

When using the referenced classes, target configuration YAML fields and class types must be separated with "." dot. For example: To set the value of testDobuleValue from the referenced field device1, the device1.testDoubleValue is used for the field value definition. Also, both types must be set, the class type and the metadata type (MySimpleClass.MdDouble).

After the class is generated (chapter 4.2 shows the example), we deploy the target configuration under the **cii.config://mytest/myreftc**:

```
config-tool deploy -i yaml/testRefTc.cdi.yaml -a cii.config://mytest/myreftc
```

The following sections show usage examples of referenced class in all languages.

Java referenced type

Listing 5-8 Java referenced type example

```
/*** Usage of the referenced class */
    CiiConfigClient client=CiiConfigClient.getInstance();

    URI myUri = new URI("cii.config://*/mytest/myreftc");

    //Retrieve config from local or remote DB
    MyClassWithRef data=
      client.retrieveConfig(myUri).getData(MyClassWithRef.class);

    //Retrieve referenced class
    MySimpleClass dataFromRef = data.getDevice1();

    System.out.println("My double value:" +
      dataFromRef.getTestDoubleValueValue());
    System.out.println("My int value:" +
      dataFromRef.getTestIntValue().getValue());
    System.out.println("My string value:" +
      dataFromRef.getTestStringValue().getValue());
```

CPP referenced type

Listing 5-9 CPP referenced type example

```
/*** Usage of the referenced class */
std::shared_ptr<::elt::config::classes::MyClassWithRef> retrievedData =
  elt::config::CiiConfigClient::getConfigData<
  ::elt::config::classes::MyClassWithRef>(elt::mal::Uri("cii.config://*/mytest/
↪myreftc"));

std::shared_ptr<::elt::config::classes::MySimpleClass> retrievedDataRef =␣
↪retrievedData->get_device1();

std::cout << "My int value:" << retrievedDataRef->get_testIntValue() <<␣
↪std::endl;
std::cout << "My double value:" << retrievedDataRef->get_testDoubleValue() <<␣
↪std::endl;
std::cout << "My string value:" << retrievedDataRef->get_testStringValue() <<␣
↪std::endl;
```

Python referenced type

Listing 5-10 Python referenced type example

```
# Usage of the referenced class
uri = elt.config.Uri("cii.config://*/mytest/myreftc")

client = elt.config.CiiConfigClient

#Retrieve config configuration
retrieved = client.retrieve_config(uri, -1)
retFromRef = retrieved.get_device1()

print("My string value:" + retFromRef.get_testStringValue())
print("My integer value: {}".format(retFromRef.get_testIntValue()))
print("My double value: {}".format(retFromRef.get_testDoubleValue()))
```

13.5.3 Using check values

The metadata classes support a check for setting minimum and maximum values on *number* types. The limits can be set by setting the values of the minLimit (setMinLimit), maxLimit (setMaxLimit) metadata instance member fields. The check is disabled by default. To enable the checks, the flag *checked* (setChecked) has to be set to true.

The minLimit uses “value >= minLimit” check logic, while maxLimit uses “value < maxLimit” check logic. In case the value exceeds the limit, the CiiConfigLimitOutOfBoundsException exception is thrown. The check is executed when setting the value using the API.

The MdString class also supports allowed values check. This check verifies that the values for config points are part of a set of allowed values. The array of allowed values has to be set. In case the value is not part of the allowed values, the CiiConfigAllowedValuesException is thrown.

The following examples show the usage of min and max limits. The usage of allowed values follows the same principle.

Java checked functions

Listing 5-11 Java checked functions example

```
/**
 * Using the check functions
 */
try {
MdInt32 checkedIntClass = MdInt32.getDefaultMDI();
checkedIntClass.setMinLimit(1);
checkedIntClass.setMaxLimit(200);
checkedIntClass.setChecked(true);
```

(continues on next page)

(continued from previous page)

```
//value in limits
checkedIntClass.setValue(50);

//value out of limits
checkedIntClass.setValue(200);

MySimpleClass testClass = new MySimpleClass();
testClass.setTestIntValue(checkedIntClass);
testClass.setTestDoubleValueValue(12.0d);
testClass.setTestStringValueValue("My test string");

} catch (CiiConfigLimitOutOfBoundsException e) {
    System.out.println(e.getMessage());
}
```

CPP checked functions

Listing 5-12 CPP checked functions example

```
/**
 * Using the check functions
 */

std::shared_ptr<MdNumber<std::int32_t>> checkedIntClass =
     std::dynamic_pointer_cast<elt::config::MdNumber<std::int32_t>>(
      ␣
↪CiiDataPointMetadataFactory::getMetadataInstance(elt::common::CiiBasicDataType::INT32))
↪

checkedIntClass->set_checked(true);
checkedIntClass->set_min_limit(1);
checkedIntClass->set_max_limit(200);

try{
  std::shared_ptr<::elt::config::classes::MySimpleClass> newSimpleInstance =
        CiiNodeFactory::getNewNodeInstance<::elt::config::classes::MySimpleClass>
↪();

  //value in limits
  newSimpleInstance->set_testIntValue(110, checkedIntClass);

  //value out of limits
  newSimpleInstance->set_testIntValue(201, checkedIntClass);

  newSimpleInstance->set_testStringValue("My test string");
  newSimpleInstance->set_testDoubleValue(12.0);
```

(continues on next page)

(continued from previous page)

```
  }
catch(const elt::config::CiiConfigLimitOutOfBoundsException& e) {
  std::cout << e.what() <<  std::endl;
}
```

Python checked functions

Listing 5-13 Python checked functions example

```
#Using the check functions

# this has to be called
checkedIntClass = elt.config.MdInt32.get_new_metadata_instance('')
checkedIntClass.set_checked(True)
checkedIntClass.set_min_limit(1)
checkedIntClass.set_max_limit(200)

try:
    newSimpleInstance = MySimpleClass.get_new_node_instance()
    newSimpleInstance.set_testStringValue("My test string")
    #value in limits
    newSimpleInstance.set_testIntValue(10, checkedIntClass)
    #value out of limits
    newSimpleInstance.set_testIntValue(202, checkedIntClass)
    newSimpleInstance.set_testDoubleValue(12.0)
except elt.config.CiiConfigLimitOutOfBoundsException as cix:
    print(cix)
```

13.6 Additional information

13.6.1 Config tool

The configuration tool (config-tool) is used to generate classes and to deploy and undeploy the YAML defined configurations to/from the local DB JSON structure. The tool also supports class generation from remote DB.

The **config-tool** has the following usage pattern: **config-tool <operation> <options>.**

To use the predefined simple metadata classes (ConfInt32Std, ConfStringStd, etc.), the classes must be deployed in the local DB. The **config-tool init** operation supports this. To initialize the local DB use:

```
config-tool init
```

For remote DB the config-initES.sh must be run. Please refer to [2] for instructions.

Class generation

The tool supports programming language configuration and metadata class files generation. The files are generated from YAML files (optionally JSON). To generate classes the **generate** option must be used. The following options are supported:

- lang: cpp, java, python. Usage: -l <arg>. Default: java
- type: config, metadata. Usage: -t <arg>. Default: config
- input file. Usage: -i <arg>
- output directory. Usage: -o <arg>
- input file is in JSON. Usage: -j
- generate wscript files for python bindings. Usage: -ws

The argument “o” is mandatory, other arguments are optional. When generating metadata classes, the type “t” must be properly set. To generate the classes from the remote DB, run the tool without “i” argument.

It is not allowed to mix the metadata and configuration classes in one YAML file. As configuration classes change frequently and metadata classes will not change much, the metadata and configuration class generation are separated. The “t” argument forces the user to explicitly know that one is actually generating metadata classes.

The class generation tool has no support for checking the names of existing classes and field names of parent classes. It is up to the user, to properly check the existing fields of the parent classes. However, the tool will check for the reserved words for field names which also contain the field names of the predefined classes. Appendix D contains the list of reserved words.

Deployment and undeployment

The tool supports deployment and undeployment of instances files to/from local DB. To deploy the instances the **deploy** option must be used. The following options are supported:

- address: target configuration URI. Option: -a <arg>
- input file. Option: -i <arg>
- type: config, metadata. Option: -t <arg> Default: config
- input file is in JSON. Option: -j
- version. Option: -v <arg>

The arguments “a” and “i” are mandatory. When deploying metadata instances the type “t” must be properly set. If no version “-v” is specified the instances are always deployed under version 1.

To undeploy instances, the **undeploy** option must be used. The following options are supported:

- address: target configuration URI. Option: -a <arg>
- instancenames: Metadata instance names to be deleted,

separated with “,”. Option: -n <args>

- type: config, metadata. Option: -t <arg> Default: config
- input file is in JSON. Option: -j
- version to be deleted. Option: -v <arg>

The version “v” must be explicitly specified (for safety reasons). To delete all versions, use -1.

The argument “n” must be used with type *metadata*. In this case the argument “a” is ignored.

To get a list of all available operations, run the:

```
config-tool -h
```

To get help on the chosen operation, run **config-tool <operation> -h**. Example:

```
config-tool undeploy -h
```

13.6.2 Generic types

Metadata classes partly support generic type inference. The generic type can be used in all the metadata classes, but the actual type used must be defined in the configuration class definition. There is no limit for the actual type to be defined before in the hierarchy of metadata. When the generic type is fixed, there is no need to define the type in the configuration class definition. Example of these classes are all classes with defined type (MdString, MdInt32). These classes have the actual type defined in the metadata class definition, so the type cannot be defined in the configuration class definition. To set the type of the metadata class to be generic, the **T** keyword must be used (no other

keywords are allowed). The keyword tells the metadata class definition that the actual type will be set by the configuration class. Using the defined generic types in the metadata class definition (e.g.: INT32) will determine the generic type of the class. If the type is fixed in the metadata class hierarchy, the *genericType* attribute of configuration class must be left out or set to *genericType*: NONE.

Listing 6-1 shows example of custom generic metadata class (MyCustomMetaClass) definition in YAML. Listing 6-2 shows use of the custom generic metadata class in configuration class YAML definition (MyClassWithCustomMeta). The type INT32 is set in the configuration class definition.

Listing 6-1 Generic type metadata class

```
configuration:
  metadataClassesConfiguration:
   classes:
    - name: MyCustomMetaClass
      parent: MdNumber
      __comment__: Custom metadata class
      genericType: T
      members:
      - name: metaOfBool
        type: BOOLEAN
      - name: metaOfString
        type: STRING
      - name: metaOfInt
        type: INT32
```

Listing 6-2 Usage of generic type metadata class

```
configuration:
  configClassesConfiguration:
   classes:
    - name: MyClassWithCustomMeta
      parent: CiiConfigClass
      __comment__: Class with custom metadata
      members:
      - name: stringPointWithCustomMeta
        type: MyCustomMetaClass
        genericType: INT32
```

CII Configuration provides an option to have generic type metadata classes and to set the type of the config point metadata class in the configuration class definition. This option can be used for classes, however the instances definition YAML files must have generic type specified for serializers to properly determine the type of data provided.

Types defined in instances must be the same as types defined in the classes in order for configuration to work properly. Listing 6-3 and Listing 6-4 show the examples of instances YAML definition files.

Listing 6-3 Config instance when generic type is used

```
metadata:
```

(continues on next page)

(continued from previous page)

```
  instances:
  - "@type": MyCustomMetaClass
    "@name": myCustomMDI
    "@genType": INT32
    comment: "String MDI with values"
    metaOfBool: "true"
    metaOfString: "The string metadata example"
    metaOfInt: "12"
```

Listing 6-4 Configuration instances with generic type YAML

```
config:
  instance:
    __comment__: Configuration for cryo cooler
    data:
      "@type": MyClassWithCustomMeta
      fields:
        stringPointWithCustomMeta:
           "@type": MyCustomMetaClass
           "@genType": INT32
           metadataInstance: myCustomMDI
           metadataInstanceVersion: 1
           value: 13
```

13.6.3 Java search example

Listing 6-5 shows the Java example for configuration search. The same API can also be used for CPP and Python. A different search language is used for local and remote DB:

- When searching in local database, a json path expression should be used: https://github.com/json-path/JsonPath
- When searching in remote database, an Elasticsearch query DSL should be used: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html#query-string-syntax

Searching with both languages is shown in the example.

Listing 6-5 Java search example

```
// remote search example
    CiiConfigClient client = CiiConfigClient.getInstance();

    // search or configuration with name 'temperatureSensor'
    String configurationName = "temperatureSensor";
    final String searchExpression = String.format("data.name:%s",␣
↪configurationName);
    try {
```

(continues on next page)

(continued from previous page)

```
      List<CiiTargetConfig> foundConfigurations = client
          .searchConfigRepo(ApiSearch.REMOTE, searchExpression);
    } catch (CiiConfigNoTcException e) {
      // handle exception in case there was no such configuration
    } catch (CiiConfigSearchException e) {
      // handle exception in case an error occurred while searching for␣
↪configuration
    }


    // local search example
    CiiConfigClient client = CiiConfigClient.getInstance();

    // search locally for configuration with name 'temperatureSensor'
    String configurationName = "temperatureSensor";
    final String search = String.format("$..[?(@.name == '%s')]",␣
↪configurationName);
    try {
      List<CiiTargetConfig> foundConfigurations = client.
↪searchConfigRepo(ApiSearch.LOCAL, search);
    } catch (CiiConfigNoTcException e) {
      // handle exception in case there was no such configuration
    } catch (CiiConfigSearchException e) {
      // handle exception in case an error occurred while searching for␣
↪configuration
    }
```

13.6.4 Large data – binary files

The MdArray and MdBinary types support storing of large data. The CII Configuration will store all binary data (MdBinary) as file references. All the arrays (MdArray) exceeding 100kb (this limit is hard coded) will be stored as binary files and only references to files will be passed to predefined metadata field “file”. Data is stored as a raw data in big-endian order.

Configuration instance YAML definition supports deployment of the raw data files. To properly deploy the file, it must be placed in the subfolder named “files”. The name of the file must be set in the file field of the config point. In Listing 6-6 the *picture1* config point will use the file *MyBigData.raw* as file reference. When the instance is deployed the file will be copied to local DB or remote large data storage.

Listing 6-6 Configuration instance field with file

```
picture1:
   "@type": MdBinary
   metadataInstance: ConfBinaryStd
   metadataInstanceVersion: 1
   value:
```

(continues on next page)

(continued from previous page)

```
file: MyBigData.raw
```

13.6.5 Extending metadata classes with additional logic

Cii Configuration provides built-in check functions (5.3), but has no direct support for extending the metadata classes with additional logic. The only way to add additional check functions is to modify the source code of the CII Configuration. For such an action, deep knowledge of internals is needed. As it is not intended for the end user to modify the source code, here only some short guidelines are provided.

To add additional logic in Java, the logic has to be put into one of the existing metadata classes. The logic has to be programmed in a separate private method, and then added to existing setValue method. The recommended classes for additional logic are MdNumber, MdArray, MdString. The logic method can also be put in MdBase (as protected), but it has to be called in setValue method of the MdNumber, MdArray or MdString classes.

For CPP additional logic has to be added as a private method in the MdBase.hpp file. The custom logic private method then has to be added to the existing validate method. As the method must support different types of an input value, it has to be a templated method.

As the Python bindings to CPP are used, no additional work is needed for the Python.

13.7 Config client API listing

Config client API provides methods for the CRUD operations of configuration and metadata instances. After the listing with methods description, the source code of specific programming language API is provided (A.1,A.2,A.3). Additional methods for searching the configurations are also present. The following methods are provided:

getInstance()
This is Java only method that returns instance of Config client API

saveTargetConfig(uri, rootNode)
Saves the configuration class to the specified URI location (cache, local, remote) and returns a version that was assigned to the saved configuration. Used programming language instance of configuration class must be set to rootNode parameter.

saveTargetConfigWithMetadata(uri, rootNode)
Saves the configuration class to the specified URI location (cache, local, remote) and returns a version that was assigned to the saved configuration. The method will also save all the metadata instances that are programmatically created and used in the programming language configuration class instance. Used programming language instance of configuration class must be set to rootNode parameter.

retrieveConfig(uri) | retrieveConfig(uri, version)
Retrieves the chosen version of target configuration from the specified location (determined by the uri). If the version is set to -1, the last target configuration will be retrieved (configuration with the highest version that is stored at the specified location. The overload with no version will return the latest version.

updateConfig(uri, rootNode) | updateConfig(uri, version, rootNode)
Updates target configuration at the specified URI with the data from the Configuration class set to the rootNode parameter. Overload with no version updates the last known configuration version.

deleteConfig(uri, version)
Deletes exact version of target config at defined URI location. If the version is set to -1, all configurations of the given URI will be deleted.

saveMetadata(host, metadataInstance)
Saves new metadata instance into the specified location. Parameter host determines the location under which metadata will be saved to. In Java the first part of the path that determines the location must be provided (cii.config://cache/, cii.config://local/ or cii.config://remote/). The CPP and Python API uses string representation of location (“cache”, “local”, “remote”).

retrieveMetadata(host, instanceName) | retrieveMetadata(host, instanceName, version)
Retrieves specific version of the metadata from the chosen location saved under the chosen instanceName. Parameter host determines the location where metadata will be retrieved from. The overload without version will return the last saved version. In Java the first part of the path that determines the location must be provided (cii.config://cache/, cii.config://local/ or cii.config://remote/). The CPP and Python API uses string representation of location (“cache”, “local”, “remote”).

updateMetadata(host, instanceName, version)
Updates specific version of the metadata with instanceName, under the chosen host location. Parameter host determines the location in which metadata will be updated. In Java the first part of the path that determines the location must be provided (cii.config://cache/, cii.config://local/ or cii.config://remote/). The CPP and Python API uses string representation of location (“cache”, “local”, “remote”).

deleteMetadata (host, instanceName, version)
Deletes specific version of the metadata with instanceName, under the chosen host location. Parameter host determines the location where metadata will be deleted from In Java the first part of the path that determines the location must be provided (cii.config://cache/, cii.config://local/ or cii.config://remote/). The CPP and Python API uses string representation of location (“cache”, “local”, “remote”).

searchConfigRepo (host, query)
Searches for target configurations that can be found using the chosen search expression. Depending on the location of the searches different search expressions should be used. For local searches a search expression should be written in JSON path format [3]. When searching in remote database a search expression should be written in elastic search query DSL format [4]. Host: In Java the first part of the path that determines the location must be provided (cii.config://local/ or cii.config://remote/). The CPP and Python API uses string representation of location (“local”, “remote”).

getChildren (uri)
The method returns all the children config point URIs under the certain path. The method will return list of URIs with the config points.

setWriteEnabled (enabled) | isWriteEnabled()
Sets or checks the write enabled state flag. If write is enabled we can save, update and delete target configurations. If write is not enabled we can only retrieve the data.

13.7.1 Java API code

```
public static synchronized CiiConfigClient getInstance() throws␣
↪CiiConfigInitializationError { ... }


public List<CiiConfigStatus> getConfigStatus(URI uri) throws␣
↪CiiInvalidURIException { ... }

public int saveTargetConfig(URI uri, CiiConfigClass rootNode)
      throws CiiConfigSaveException, CiiInvalidURIException,␣
↪CiiConfigWriteDisabledException { ... }

public int saveTargetConfigWithMetadata(URI uri, CiiConfigClass rootNode)
      throws CiiConfigWriteDisabledException, CiiInvalidURIException,␣
↪CiiConfigSaveException { ... }

public void updateConfig(URI uri, CiiConfigClass rootNode)
      throws CiiConfigNoTcException, CiiConfigUpdateException,␣
↪CiiInvalidURIException, CiiConfigWriteDisabledException { ... }

public void updateConfig(URI uri, int version, CiiConfigClass rootNode)
      throws CiiConfigNoTcException, CiiConfigUpdateException,␣
↪CiiInvalidURIException, CiiConfigWriteDisabledException { ... }

public CiiTargetConfig retrieveConfig(URI uri)
      throws CiiConfigNoTcException, CiiInvalidURIException { ... }

public CiiTargetConfig retrieveConfig(URI uri, int version)
      throws CiiConfigNoTcException, CiiInvalidURIException { ... }

public void deleteConfig(URI uri, int version)
      throws CiiConfigDeleteException, CiiInvalidURIException,␣
↪CiiConfigWriteDisabledException { ... }

public List<CiiTargetConfig> searchConfigRepo(ApiSearch searchPath, String␣
↪searchExp)
      throws CiiConfigNoTcException, CiiConfigSearchException { ... }

public List<String> remoteIndexSearch(String index, String query)
      throws CiiConfigSearchException { ... }

public <T extends MdBase> T retrieveMetadata(URI searchLocation,
```

(continues on next page)

(continued from previous page)

```
      String metadataInstanceName, int version, Class<T> clazz)
      throws IOException, CiiInvalidURIException { ... }
public <T extends MdBase> int saveMetadata(URI saveLocation, T metadataInstance)
      throws CiiConfigSaveException, CiiInvalidURIException,␣
↪CiiConfigWriteDisabledException { ... }

public void deleteMetadata(URI deleteLocation, String metadataInstanceName, int␣
↪version)
      throws CiiConfigWriteDisabledException, CiiInvalidURIException,␣
↪CiiConfigDeleteException { ... }

public List<URI> getChildren(URI uri) throws CiiInvalidURIException,␣
↪CiiConfigSearchException { ... }

public void setWriteEnabled(boolean writeEnabled) { ... }

public boolean isWriteEnabled() { ... }
```

13.7.2 CPP API code

```
static int saveTargetConfig(const elt::mal::Uri& uri, const CiiConfigClass& root_
↪node);

static int saveTargetConfigWithMetadata(const elt::mal::Uri& uri, CiiConfigClass&
↪ root_node);

static std::list<std::shared_ptr<CiiDataPointMetadataBase>>␣
↪getTargetConfigMetadata(CiiConfigurationBase& root_node);

static int saveMetadata(const std::string& host, CiiDataPointMetadataBase&␣
↪metadata);

static std::shared_ptr<CiiConfigClass> retrieveConfig(
              const elt::mal::Uri& uri, int version = -1);
static std::shared_ptr<CiiDataPointMetadataBase> retrieveMetadata(
              const std::string& host, const std::string& instance_name, int␣
↪version = -1);

static void updateConfig (const elt::mal::Uri& uri, int version,
          const CiiConfigClass& targetConfig);

static void updateMetadata(const std::string& host, const␣
↪CiiDataPointMetadataBase& metadata);

static void updateMetadata(const std::string& host, int version, const␣
↪CiiDataPointMetadataBase& metadata);
```

(continues on next page)

(continued from previous page)

```
static void deleteConfig (const elt::mal::Uri& uri, int version = -1);

static void deleteMetadata (const std::string& host, const std::string& instance_
↪name, int version = -1);

template <typename T>
static std::shared_ptr<T> getConfigData(const elt::mal::Uri& uri, int version = -
↪1) {
      return std::dynamic_pointer_cast<T>(retrieveConfig(uri, version));
  }

template <typename T>
static std::shared_ptr<T> getMetadata(const std::string& host, const std::string&
↪ instance_name, int version = -1) {
      return std::dynamic_pointer_cast<T>(retrieveMetadata(host, instance_name,␣
↪version));
  }

static const std::list<elt::mal::Uri> getChildren(const elt::mal::Uri& parent_
↪uri);

static const std::list<std::shared_ptr<CiiConfigClass>> searchConfigRepo(const␣
↪std::string& host, const std::string& query);

static void setWriteEnabled(bool enabled);

static bool isWriteEnabled();
```

13.7.3 Python bindings code

```
py::class_<elt::config::CiiConfigClient, std::shared_ptr
↪<elt::config::CiiConfigClient>>(m, "CiiConfigClient",
            "Configration Client API")
  .def_static("save_target_config", &
↪elt::config::CiiConfigClient::saveTargetConfig,
              "Save a new version of configuration class")
  .def_static("save_target_config_with_metadata", &
↪elt::config::CiiConfigClient::saveTargetConfigWithMetadata,
               "Save configuration node and the metadata object it references")
  .def_static("retrieve_config", &elt::config::CiiConfigClient::retrieveConfig,
              "Retrieve a configuration object previously saved via␣
↪saveTargetConfig")
  .def_static("retrieve_config",
              [](const ::elt::mal::Uri &uri) { return␣
↪elt::config::CiiConfigClient::retrieveConfig(uri, -1); },
              "Retrieve a configuration object previously saved via␣
↪saveTargetConfig")
```

(continues on next page)

(continued from previous page)

```
  .def_static("update_config", &elt::config::CiiConfigClient::updateConfig,
              "Overwrite previously saved configuration or save it using a␣
↪specific version")
  .def_static("delete_config", &elt::config::CiiConfigClient::deleteConfig,
              "Delete a configuration object")
  .def_static("delete_config", [](const elt::mal::Uri &uri) {
      elt::config::CiiConfigClient::deleteConfig(uri);
    }, "Delete a configuration object")
  .def_static("get_config_data", &elt::config::CiiConfigClient::getConfigData
↪<elt::config::CiiConfigClass>,
              "Retrieve a configuration object")
  .def_static("save_metadata", &elt::config::CiiConfigClient::saveMetadata,
              "Save a new version of a metadata instance")
  .def_static("update_metadata", &elt::config::CiiConfigClient::updateMetadata,
              "Retrieve a metadata object previously saved via saveMetadata()")
  .def_static("delete_metadata", &elt::config::CiiConfigClient::deleteMetadata,
              "Delete a metadata object")
  .def_static("delete_metadata", [](const std::string &host, const std::string&␣
↪instance_name) {
      ::elt::config::CiiConfigClient::deleteMetadata(host, instance_name);
    }, "Delete a metadata object")
  .def_static("retrieve_metadata", &
↪elt::config::CiiConfigClient::retrieveMetadata,
              "Retrieve a metadata object previously saved via saveMetadata")
  .def_static("retrieve_metadata", [](const std::string &host, const std::string␣
↪&instance_name) {
      return elt::config::CiiConfigClient::retrieveMetadata(host, instance_name);
    }, "Retrieve a metadata object previously saved via saveMetaData")
  .def_static("search_config_repo", &
↪elt::config::CiiConfigClient::searchConfigRepo,
              "Search a data provider for specific configurations")
  .def_static("set_write_enabled", &
↪elt::config::CiiConfigClient::setWriteEnabled,
              "Enables or disables the possibility of writing into the␣
↪configuration databases")
  .def_static("is_write_enabled", &elt::config::CiiConfigClient::isWriteEnabled,
              "Returns True if the API allows writing into configuration␣
↪databases, false otherwise")
  .def_static("get_children", &elt::config::CiiConfigClient::getChildren,
              "Query children nodes")
```

13.8 Config point value type to language types mapping

Below is the mapping table between config point types and data types of all programming languages that the configuration can use.

Table 7-1 Config point types mapping table

ELT CII BASIC TYPE	Java	CPP		Python
INT32	Integer	std::int32_t		int
INT64	Long	std::int64_t		int
UINT8	Integer	std::uint8_t		int
UINT16	Long	std::uint16_t		int
UINT32	Long	std::uint32_t		int
SINGLE	Float	float		float
DOUBLE	Double	double		float
BOOLEAN	Boolean	bool		bool
STRING	String	std::string		str
BINARY	byte[]	std::vector<std::uint8_t>		bytearray
YAML		elt::config::datatypes:: YamlNode		elt. config. datatypes. YamlNode

13.9 Default metadata instance mapping

Table 7-2 shows the default metadata instance and class type mappings. The columns “metadata instance @genType” and “class genericType” are CII Basic types. These columns are used in YAML class definition for metadata classes and metadata instances. The fields are described in Appendix E.

Table 7-2 Default metadata mappings

Metadata class name	Metadata instance @genType	Class genericType	Is Array	Default metadata instance
MdInt32	INT32	INT32	F	ConfInt32St d
MdInt64	INT64	INT64	F	ConfInt64St d
MdFloat	SINGLE	SINGLE	F	ConfFloatSt d
MdDouble	DOUBLE	DOUBLE	F	ConfDoubleS td
MdBool	BOOLEAN	BOOLEAN	F	ConfBoolStd
MdString	STRING	STRING	F	ConfStringS td
MdStringArr ay	VECTOR_STRI NG	STRING	T	ConfStringA rrayStd
MdInt32Arra y	VECTOR_INT3 2	INT32	T	ConfInt32Ar rayStd
MdFloatArra y	VECTOR_SING LE	SINGLE	T	ConfFloatAr rayStd
MdDoubleArr ay	VECTOR_DOUB LE	DOUBLE	T	ConfDoubleA rrayStd
MdInt32Matr ix	MATRIX2D_IN T32	INT32	T	ConfInt32Ma trixStd
MdFloatMatr ix	MATRIX2D_SI NGLE	SINGLE	T	ConfFloatMa trixStd
MdDoubleMat rix	MATRIX2D_DO UBLE	DOUBLE	T	ConfDoubleM atrixStd
MdBinary	BINARY	BINARY	F	ConfBinaryS td
MdYaml	YAML	YAML	F	ConfYamlStd

13.10 Class definition reserved words

Table 7-3 shows list of words that cannot be used as field names in config or metadata class definitions.

Table 7-3 List of words not allowed for config and metadata class names

Field name
name
uri
defaultValue
value
comment
ciiType
genType
metadataChanged
minLimit
maxLimit
allowedValues
size
field
file
genericType
parent
__comment__
isRef

13.11 YAML Data point type

Inclusion of the support for YAML data point type was requested with ECII-702 which requested such support in OLDB. As OLDB relies on config, YAML data point type support had to be also added to config.

It was requested that OLDB should support data points that should accept Yaml::Node data type defined by yaml-cpp.

Directly mapping Yaml::Node datatype was not feasible as it is unknown Python, therefore intermediate data type elt::config:datatypes::YamlNode was defined in C++.

Datatype internally serialises to yaml document (string). Python binding for elt::config::datatypes::YamlNode is available as elt.config.datatypes.YamlNode.

YamlNode can be initialized in one of the following ways in C++:

```
#include <datatypes/yamlNode.hpp>
...
// Empty yaml document
auto node = elt::config::datatypes::YamlNode();
// Initialize with valid yaml document source. When argument cannot be
// parsed as valid yaml, elt::config::CiiValue is thrown
node = elt::config::datatypes::YamlNode("[this, is, yaml, list]");
// Initialize with YAML::Node argument
YAML::Node source = ...
node = elt::config::datatypes::YamlNode(source);
// Extract yaml document as string
std::string doc = node.AsString();
// Extract yaml document as YAML::Node
YAML::Node ydoc = node.AsYamlObject();
// Assignment of new value from string
node = "[this, is, another, yaml, list]";
// Assignment of new value from YAML::Node
node = ydoc;
```

Python bindings replicate and extend C++ functionality:

```
from elt.config.datatypes import YamlNode
# Empty yaml document
node = YamlNode()
# Initialize yaml document from string containing valid yaml. Again␣
↪CiiValueError is raised in # case argument does not contain valid yaml.
node = YamlNode('[10, 20, 30, 40]')
# Initialize yaml document from python object
node = YamlNode(['this', 'is', 'python', 'list'])
# Update value with new yaml document source
node.set_source('[90, 20, 30, 40]')
# Update yaml node with new python object
node.set_value([90, 20, 30, 40])
```

(continues on next page)

(continued from previous page)

```
# Extract yaml document source as string
s = node.as_string()
# Extract yaml document as python object
assert(node.as_value(), [90, 20, 30, 40])
```

Apart from the fact that YamlNode data type is not native and must be initialized in the one of the ways listed above, Config API supports data points that contain this type.

Vectors of YamlNode elements are not supported by Config API.

13.12 JSON/YAML Schema

Schemas are defined for classes and instances. All local and remote DB JSON files must be in line with the defined schemas. Target configuration and instances schemas are used for value bindings. Class schemas are used for class generation. For local and remote DB, YAML structures are translated into JSON structures with nested elements.

The following schema descriptions are indented in the same manner as the actual YAML files. The indentation follows the YAML files indentation structure.

All class schemas start with the starting structure, which is presented only here and not listed in the subchapters. This structure contains the following elements: “field: type, required/optional; description”. The definition is as follows:

- **configuration**: string, required; defines the root for all configuration schemas
  - **configClassesConfiguration** or **metadataClassesConfiguration** string, required; defines the type of configuration schema.

13.12.1 Configuration class schema

Configuration class schema (field: type, required/optional; description):

- **classes**: array, required; List of classes to be generated.
  - **name:** string, required; Name of the configuration class.
  - **parent**: string, required; Parent configuration class name.
  - **__comment__**: string, optional; Class comment. This comment is changed to programming language specific comment on bindings generation.
  - **members**: array, required; member fields of classes (config points)
    - **name**: string, required; config point name.
    - **type**: string, required; config point metadata class type.
    - **genericType**: string, optional; generic type of config point (CII data type)
    - **isRef**: boolean, optional; True if this member is a reference to another class.

13.12.2 Metadata class schema

Metadata class schema (field: type, required; description):

- **classes**: array, required; List of metadata classes to be generated.
  - **name:** string, required; Name of the metadata class.
  - **parent**: string, required; Parent metadata class name.
  - **__comment__**: string, optional; Metadata class comment. This comment is changed to programming language specific comment on bindings generation.
  - **members**: array, required; member fields of metadata classes
    - **name**: string, required; metadata member field name.
    - **type**: string, required; CII data type of metadata class.

13.12.3 Configuration instance schema

Target configuration (configuration instance) schema has the following starting structure:

- **config:** string, required; Root node.
  - **instance:** string, required; Definition of config instance.

The starting structure is followed by the schema structure for the details. The details schema is child node of **instance** (field: type, required/optional; description):

- **__comment__**: string, optional; YAML/JSON comment for the target configuration.
- **data**: Object, required; Data object.
  - **@type**: string, required; Configuration class type.
  - **fields**: string, required; List of all members that have the same member name and type as defined in the corresponding class definition. Member name and value pairs are added to YAML (“field”: “value”):
    - **@type:** string, required; Metadata class type.
    - **metadataInstance**: string, required; Metadata instance name.
    - **metadataInstanceVersion**: string, optional; Version of metadata instance. Last existing version is used if the version is not specified.
    - **value**: string | number | boolean | array, required; the actual value of config point.
    - **@genType:** string, optional; generic type used in the instance.

To address the fields of referenced classes, special name rule is used. The name is a concatenation of reference instance name followed by the reference field name, separated by a dot “.” character is used. The same logic is used when one reference class references another class.

Examples:

1. Configuration class contains referenced adFlowClass. The “adFlow1.sensorsSize” will address the sensorsSize field in the adFlow1 referenced class instance.
2. Configuration class contains referenced adFlowClass, which references currentFlowClass. The “adFlow1.currFlow1.running” will address the running field in the currFlow1 referenced class instance, which is referenced from adFlow1.

13.12.4 Metadata instances schema

Metadata instances schema has the following starting structure:

- **metadata:** string, required; Root node.
  - **instances:** string, required; Definition of metadata instance.

The starting structure is followed by the schema structure for the details. The details are child nodes of **instances** node: (field: type, required; description):

- **@name:** string, required; Name of metadata instance.
- **@type**: string, required; Metadata class of metadata instance.
- **@genType**: string, required; Generic type used by the instance. This type must be the same as the defined metadata class type.
- List of all member names that are in line with metadata class. Member name and value pair are added to YAML (“field”: “value”):
  - **field**: string, required; metadata class field name.
  - **value**: string | number | boolean | array, required; the actual value of metadata field.

13.13 Configuration client settings

Configuration client API needs some internal setting to work properly.

They can be provided through a config file in ini format, which is pointed to by the CONFIG_CLIENT_INI environment variable.

```
# Example of config client ini file

# Uri to the redis cluster member
redisAddress localhost:7000

# Timeout in ms for socket operations
# connectionTimeout: 15000

# Number of connections maintained in the connection pool
# redisConnectionPoolSize: 15

# Timeout in ms waiting to receive connection from connection pool
# redisConnectionPoolWaitTimeoutMs: 10000

# Lifetime of connection within redis connection pool
# redisConnectionPoolConnectionLifetimeMs: 30000
```

For values that you don't define, default values will be used.

13.14 Code produced by the generators

Listing 7-1 Java generated code by the config tool

```
/*************************************************************
 * @copyright (c) Copyright ESO 2019 All Rights Reserved ESO (eso.org) is an
 *  Intergovernmental
 *            Organisation, and therefore special legal conditions apply.
 *************************************************************/
package elt.config.classes;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.ObjectWriter;
import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.core.JsonProcessingException;
import java.util.List;
import elt.error.CiiInvalidTypeException;
import elt.config.exceptions.CiiConfigException;
import elt.config.CiiConfigClass;

import elt.config.classes.meta.MdInt32;
import elt.config.classes.meta.MdDouble;
import elt.config.classes.meta.MdString;

 //Personal config test class
public class MySimpleClass extends CiiConfigClass {

  private MdInt32 testIntValue;
  private MdDouble testDoubleValue;
  private MdString testStringValue;



  //default constructor is needed by the ser/deser engine.
  public MySimpleClass(){
    super();
    this.testIntValue = MdInt32.getDefaultMDI();
    this.testDoubleValue = MdDouble.getDefaultMDI();
    this.testStringValue = MdString.getDefaultMDI();
  }



  public MySimpleClass(String name,
      MdInt32 testIntValue,
      MdDouble testDoubleValue,
      MdString testStringValue) {
    super(name);
    this.testIntValue = testIntValue;
    this.testDoubleValue = testDoubleValue;
```

(continues on next page)

(continued from previous page)

```
    this.testStringValue = testStringValue;
  }


  public MySimpleClass(String name,
      int testIntValue,
      double testDoubleValue,
      String testStringValue) throws CiiInvalidTypeException {

    super(name);
    this.testIntValue = new MdInt32(testIntValue);
    this.testDoubleValue = new MdDouble(testDoubleValue);
    this.testStringValue = new MdString(testStringValue);
  }


  public MdInt32 getTestIntValue() {
    return testIntValue;
  }

  public void setTestIntValue(MdInt32 testIntValue) {
    this.testIntValue = testIntValue;
  }

  @JsonIgnore
  public int getTestIntValueValue() {
    return testIntValue.getValue();
  }

  @JsonIgnore
  public void setTestIntValueValue(int testIntValue) throws CiiConfigException  {
    this.testIntValue.setValue(testIntValue);
  }
  public MdDouble getTestDoubleValue() {
    return testDoubleValue;
  }

  public void setTestDoubleValue(MdDouble testDoubleValue) {
    this.testDoubleValue = testDoubleValue;
  }

  @JsonIgnore
  public double getTestDoubleValueValue() {
    return testDoubleValue.getValue();
  }

  @JsonIgnore
  public void setTestDoubleValueValue(double testDoubleValue) throws␣
↪CiiConfigException  {
```

(continues on next page)

(continued from previous page)

```
    this.testDoubleValue.setValue(testDoubleValue);
  }
  public MdString getTestStringValue() {
    return testStringValue;
  }

  public void setTestStringValue(MdString testStringValue) {
    this.testStringValue = testStringValue;
  }

  @JsonIgnore
  public String getTestStringValueValue() {
    return testStringValue.getValue();
  }

  @JsonIgnore
  public void setTestStringValueValue(String testStringValue) throws␣
↪CiiConfigException  {
    this.testStringValue.setValue(testStringValue);
  }

  public String toString() {
    try {
      ObjectWriter writer = new ObjectMapper().writer().
↪withDefaultPrettyPrinter();
      return writer.writeValueAsString(this);
    } catch (JsonProcessingException e) {
      e.printStackTrace();
      return "";
    }

  }
}
```

Listing 7-2 HPP generated code from by config tool

```
#ifndef CII_SVCS_CONFIG_TEST_MYSIMPLECLASS_HPP_
#define CII_SVCS_CONFIG_TEST_MYSIMPLECLASS_HPP_
/*************************************************************
 * @copyright (c) Copyright ESO 2019 All Rights Reserved ESO (eso.org) is an␣
↪Intergovernmental
 *            Organisation, and therefore special legal conditions apply.
 *************************************************************/

#include <ciiNodeContainer.hpp>
#include <ciiNodeFactory.hpp>

#include <string>
#include <vector>
```

(continues on next page)

(continued from previous page)

```
using namespace elt::config;


namespace elt{
namespace config{
namespace classes{
class MySimpleClass : public ::elt::config::CiiConfigClass {

public:
  DECLARE_NODE_CLASS();

  DECLARE_NODE_FIELD_METADATA(MdInt32, testIntValue);
  DECLARE_NODE_FIELD_METADATA(MdDouble, testDoubleValue);
  DECLARE_NODE_FIELD_METADATA(MdString, testStringValue);
};
// class MySimpleClass
} //namespace elt
} //namespace config
} //namespace classes

#endif // CII_SVCS_CONFIG_TEST_MYSIMPLECLASS_HPP_
```

Listing 7-3 CPP generated code from the config tool

```
/*******************************************************************
 * @copyright (c) Copyright ESO 2019 All Rights Reserved ESO (eso.org) is an␣
↪Intergovernmental
 *            Organisation, and therefore special legal conditions apply.
 *******************************************************************/
#include <ciiNodeFactory.hpp>
#include "mySimpleClass.hpp"

namespace elt{
namespace config{
namespace classes{

IMPLEMENT_NODE_CLASS(MySimpleClass, MySimpleClass)

} //namespace elt
} //namespace config
} //namespace classes
```

Listing 7-4 Python generated code from the config tool

```
py::class_<MySimpleClass, std::shared_ptr<MySimpleClass>,
   ::elt::config::CiiConfigClass>(m,"MySimpleClass")
  .def_static("getNewNodeInstance", []() {
     return ::elt::config::CiiNodeFactory::getNewNodeInstance<MySimpleClass>();
```

(continues on next page)

(continued from previous page)

```
    })
  .def("get_testIntValue", &MySimpleClass::get_testIntValue)
  .def("set_testIntValue", (void (MySimpleClass::*)(const␣
↪CiiCPIntegerDataClass::data_type_t&))
                                                          &MySimpleClass::set_
↪testIntValue)
  .def("set_testIntValue", (void (MySimpleClass::*)(const␣
↪CiiCPIntegerDataClass::data_type_t&,
                                                          const std::shared_ptr
↪<CiiCPIntegerDataClass>&))
                                                          &MySimpleClass::set_
↪testIntValue)

  .def("get_testDoubleValue", &MySimpleClass::get_testDoubleValue)
  .def("set_testDoubleValue", (void (MySimpleClass::*)(const␣
↪CiiCPDoubleDataClass::data_type_t&))
                                                          &MySimpleClass::set_
↪testDoubleValue)
  .def("set_testDoubleValue", (void (MySimpleClass::*)(const␣
↪CiiCPDoubleDataClass::data_type_t&,
                                                          const std::shared_ptr
↪<CiiCPDoubleDataClass>&))
                                                          &MySimpleClass::set_
↪testDoubleValue)

  .def("get_testStringValue", &MySimpleClass::get_testStringValue)
  .def("set_testStringValue", (void (MySimpleClass::*)(const MdString::data_type_
↪t&))
                                                          &MySimpleClass::set_
↪testStringValue)
  .def("set_testStringValue", (void (MySimpleClass::*)(const MdString::data_type_
↪t&,
                                                          const std::shared_ptr
↪<MdString>&))
                                                          &MySimpleClass::set_
↪testStringValue)

;
}
```

Listing 7-5 Deployed JSON file from the config-tool

```
{
  "__comment__" : "Configuration for cryo cooler",
  "data" : {
    "@type" : "MySimpleClass",
    "testStringValue" : {
      "@type" : "MdString",
      "metadataInstance" : "ConfStringStd",
```

(continues on next page)

(continued from previous page)

```
      "metadataInstanceVersion" : 1,
      "value" : "My string defined value",
      "@genType" : "STRING"
    },
    "testIntValue" : {
      "@type" : "MdNumber",
      "metadataInstance" : "ConfInt32Std",
      "metadataInstanceVersion" : 1,
      "@genType" : "INT32",
      "value" : 155
    },
    "testDoubleValue" : {
      "@type" : "MdNumber",
      "metadataInstance" : "ConfFloatStd",
      "metadataInstanceVersion" : 1,
      "@genType" : "SINGLE",
      "value" : 155.33
    }
  }
}
```