# CONTENTS

1 Preface 1  
1.1 About this guide ................................................. 1  
1.2 Intended audience ................................................. 1  

2 Introduction 2  
2.1 Key Concepts ...................................................... 2  
2.2 Data Model ...................................................... 3  
2.3 ThingClass and TagGroup Definitions ......................... 3  
2.4 Thing Context .................................................... 4  
2.5 Thing ID ......................................................... 4  
2.6 QoS Profiles ..................................................... 5  

3 Implement a Thing 7  
3.1 The Basics ....................................................... 7  
3.1.1 Data River Configuration ...................................... 8  
3.1.2 Data River Instantiation ...................................... 8  
3.1.3 Register TagGroup ............................................ 9  
3.1.3.1 TagGroup JSON Definition .................................. 9  
3.1.4 Register ThingClass .......................................... 9  
3.1.5 Create Reader Thing .......................................... 10  
3.1.6 Create Writer Thing .......................................... 11  
3.1.7 Create Thing with Auto Generated Unique ID ............... 11  
3.1.8 Write Data .................................................... 12  
3.1.9 Read Data ..................................................... 12  
3.2 Real-world scenario 1: Connecting a sensor .................. 13  
3.2.1 TagGroup and Thing Class Definition ....................... 13  
3.2.2 Thing Properties ............................................ 14  
3.2.3 Application Code ........................................... 14  

4 Discovering Things and Meta-Data 15  
4.1 Discovery ........................................................ 15  
4.2 Real-world scenario 2: Connect a dashboard ................ 16  
4.2.1 Using a Read Selector ........................................ 16  
4.2.1.1 Select versus Filter ....................................... 17  
4.2.1.2 Selector API ............................................... 18  
4.2.2 Thing Discovery ............................................. 18  

5 Flow ID 20  
5.1 Introduction on flowId ........................................... 20  
5.1.1 Use-case for a custom flowId ................................. 20
5.2 Real-world scenario 3: A derived value processing service ........................................ 21
   5.2.1 Derived Value Service Thing Implementation ................................................ 22
   5.2.2 Reader implementation ................................................................................. 23
   5.2.3 Listener Pattern ............................................................................................ 24
   5.2.4 Custom Dispatcher ....................................................................................... 24
   5.2.5 Use of TypeDefinitions .................................................................................. 25
5.3 Real-world scenario 2A: Connect a dashboard using FlowId filter .................................. 27
   5.3.1 FlowId Filter ................................................................................................. 27
   5.3.2 flowIdFilters expressions ............................................................................ 28

6 More Real-World Scenarios ............................................................................................................. 29
   6.1 Real-world scenario 4: A Gateway Service ......................................................... 29
      6.1.1 Dynamic Inputs ............................................................................................. 30
      6.1.2 Processing data using read_next .................................................................. 31
      6.1.3 Meta-data for received data ......................................................................... 32
      6.1.3.1 Flow State ................................................................................................. 32
   6.2 Real-world scenario 5: Dynamic Browsing .............................................................. 33
      6.2.1 Registries ..................................................................................................... 34

7 QoS Profile Override ....................................................................................................................... 36
   7.1 Create Custom QoS Profile ................................................................................... 36
   7.2 Update Reader’s Thing Configuration Properties ..................................................... 38

8 Google Protocol Buffer (GPB) TagGroup Definitions ................................................................. 40
   8.1 Overview ............................................................................................................. 40
      8.1.1 What are Google Protocol Buffers (GPB)? ..................................................... 40
      8.1.2 GPB Limitations ............................................................................................ 41
      8.1.3 Examples ....................................................................................................... 41
   8.2 GPB TagGroup .proto Files ................................................................................ 41
      8.2.1 Convert existing TagGroup JSON files to .proto files ....................................... 42
      8.2.2 Define new .proto files ................................................................................. 42
   8.3 Generate Data Access Classes ............................................................................. 43
      8.3.1 Invoking protoc for C++ ............................................................................... 43
      8.3.2 Generated output C++ ............................................................................... 45
      8.3.3 Generating C++ with CMake ........................................................................ 45
   8.4 Application Code API ........................................................................................... 47
      8.4.1 Accessing Generated C++ classes ................................................................. 47
      8.4.2 Registering Protobuf TagGroups .................................................................... 47
      8.4.3 Use ThingEx instead of Thing ....................................................................... 48
      8.4.4 Handling variant data: VLoanedDataSamples and VDataSample ................. 48
      8.4.5 ThingEx read(), read_next() and DataAvailableListenerEx ....................... 50
      8.4.6 Writing Protobuf data .................................................................................. 50
      8.4.7 Advanced: Dealing with unknown protobuf types ........................................ 51
   8.5 Type Evolution ....................................................................................................... 52
      8.5.1 How a generation of applications evolve from JSON/NVP_SEQ to Protobuf ....... 52
      8.5.2 How a generation of applications evolve a Protobuf TagGroup ....................... 53
   8.6 Real-world scenario 2: Connect a dashboard (GPB) ..................................................... 53
      8.6.1 Temperature TagGroup .proto ....................................................................... 54
      8.6.2 ThingClass Definitions ............................................................................... 54
      8.6.3 Include .......................................................................................................... 55
      8.6.4 Register TagGroup ....................................................................................... 55
      8.6.5 ThingEx ........................................................................................................ 55
      8.6.6 Write ............................................................................................................. 55
      8.6.7 Read .............................................................................................................. 56
1.1 About this guide

The Edge SDK User guide is a starting point for anyone developing Internet of Things (IoT) applications using the ADLINK Edge SDK. The purpose of this guide is to provide an introduction to the key concepts of the Thing model and enable C++ programmers using the Edge SDK to build IoT functionality. This guide contains an explanation of the design and implementation of a number of real-world use-cases for the Edge SDK. The complete source code for these examples can be found in the ‘examples’ directory of the SDK.

1.2 Intended audience

The Edge SDK User Started Guide is intended to be used by C++ programmers who are using the ADLINK Edge SDK 1.x to develop IoT applications.
INTRODUCTION

The **ADLINK Edge SDK** provides a simple API for creating and maintaining digital twins of typically physical assets (*Things*) on a data sharing framework, the **ADLINK Data River**. A (connected) *Thing* projects a digital twin of itself on the DataRiver. Things typically represent a physical device, but can also be a software-only asset. A *Thing* can represent e.g. status, inputs and outputs of that asset.

![Functional view of a Thing](image)

**Fig. 1: Functional view of a Thing**

### 2.1 Key Concepts

This section explains some of the the key concepts used in solutions built with the Edge SDK:

**Thing** In IoT a *Thing* is typically something physical and somehow connected to become part of a network of Things. From a logical view, a Thing is the digital projection (also called digital twin) of a physical Thing or a software-only asset. Applications use the Edge SDK to create *Things* that connect to a *Data River*. This guide contains real-world examples of *Things* representing physical assets and *Things* representing software-only assets.

**Data River** The Data River is a data sharing framework with data-centric capabilities. The Data River makes data available to Things where and whenever it is needed. The Edge SDK helps interfacing with the Data River and hides the details of the underlying middleware platform.

**ThingClass** A *ThingClass* specifies a ‘kind of thing’ (e.g. sensors, actuators, subsystems, database-integrations, analytics-solutions). The *ThingClass* acts as a blueprint for a *Thing* and has a name, context, version, and a list of named inputs and outputs.

**TagGroup** The inputs and outputs in a *ThingClass* are defined by means of *TagGroups*. A *TagGroup* has a name, context, version, a structure of Tags and a related QoS profile. Each tag has a name, kind (data type), description, unit and validation criteria. The data type of a Tag can be either a built-in type (e.g. INT32, FLOAT, STRING).
or refer to a *TypeDefin**ition*. A *TypeDefin**ition* is defined as a named list of Tags that is intended to be reusable over multiple TagGroups. Usage of a *TypeDefin**ition* shown in example scenario 3.

Fig. 2: Key entities used in the Edge SDK model

### 2.2 Data Model

The Edge SDK model provides an API for representing Things on the Data River. Using the Edge SDK API a *Thing* can be created as an instantiation of a *ThingClass*. Things are provided with a context that identifies its logical role within the system. For example, a system that has temperature sensors for inside and outside of a building, the *contextId* of these sensors can be set to ‘mySystem.temperatureSensorInside’ and ‘mySystem.temperatureSensorOutside’.

Things can share data with the Data River via their output(s) and Things receive data from other Things on the river by their input(s). The data structure for inputs and outputs of a *Thing* is defined by means of *Tags* that are grouped together in a *TagGroup*. *Things* exchange data in the form of samples of *TagGroups*. The *ThingClass* defines the *TagGroups* that the Thing uses for input and output.

In the temperature sensor example above, the output for the sensor could be defined as *TagGroup* ‘Temperature’ that has a single Tag ‘temperature’. A Thing in this system that reads this data would use the ‘Temperature’ TagGroup for its input.

### 2.3 ThingClass and TagGroup Definitions

Edge SDK includes a JSON schema for defining *ThingClass* and *TagGroups*.

JSON is used as the specification format for *ThingClass* definitions in the Edge SDK.

*TagGroup* definitions can be specified using JSON or Google protocol buffers (in Edge SDK C++).

Fig. 2 shows a *Thing* refers to a *ThingClass* and a *ThingClass* can refer to one or more *TagGroups*. Both references are in the format ‘name:context:version’, to uniquely refer to a specific definition version in the correct context. More details on using context can be found in the section *Context* below. The example scenarios in this document contain several examples of references using this format.

**Note:** GPB TagGroup support is currently provided for Edge SDK C++.
2.4 Thing Context

During instantiation a Thing is provisioned with a contextId to identify the context of the Thing in the system. This context is a logical identification that also captures what the Thing is 'part of’, e.g. ‘generatorA.fuelTemp’. Discoverable information about created Things also include this contextId. The context of a Thing can be exploited to identify dataflows published by a Thing on the Data River. For more details on dataflows, see Flow ID.

Note: TagGroups and ThingClasses also have a context. The context value for a TagGroup and ThingClass is used to make the definition (that may have a generic name, e.g. ‘Temperature’ or ‘FuelTempSensor’) unique, e.g. when placed in a central repository. The suggested naming strategy for the context of a TagGroup or a ThingClass is the same as the Java package convention of the reversed URL (adlinktech.com becomes com.adlinktech). However, it is completely valid to use any other naming convention.

2.5 Thing ID

In addition to the logical context, a Thing must have a thingId that uniquely identifies the Thing within its ThingClass. For example, for a Thing that represents a physical device, the thingId can be set to the serial number of the physical device it represents.

For Things that do not have a connected physical device, the ThingID can be any unique string (within the ThingClass). For example, a derived value service that is only a software process. A randomly generated UUID could be used here.

![Fig. 3: Thing Context and Thing ID](image-url)
2.6 QoS Profiles

To control the Quality-of-Service (QoS), a TagGroup has an associated QoSProfile.

The QoSProfile:

- Indicates the kind of data that the TagGroup represents.
- Describes how the Data River handles network connectivity issues.
- Controls how much (historical) data needs to be available.

The Edge SDK has a set of pre-defined QoSProfiles, which can be extended with custom profiles. The pre-defined profiles support the following kinds of data:

- **telemetry** – for digital snapshots of the analogue world, where the most recent data is more relevant than old data.
- **state** – data that describes the current (last-known) or desired (future) state of a Thing. The data must be retained and made available to existing and emerging Things on the Data River.
- **event** – this kind of data typically represents periodic (noteworthy) occurrences such as:
  - alarms
  - warnings
  - errors/exceptions
  - data streams

Where, correct interpretation requires all events to be delivered to interested parties (unlike ‘telemetry’ where the (most) recent data is sufficient).

- **video** - data is characterized by large samples with relatively low rates. This profile exploits best effort delivery (unlike the telemetry, state and event profiles that have reliable delivery).

  The cost of re-transmission of old frames is too high when compared to the low-impact of loosing a frame every now-and then.

  The Video profile has a keep-last history at both the writer-side and reader-side, which allows downsampling towards the network (while retaining high-throughput) and downsampling towards a (temporary) slow reader of the data.

- **vibration** - for high-frequency data flows with small samples. For example, data from a vibration sensor. This profile uses a reasonable history-depth at both the writer-side (to facilitate packing that reduces wirefrequency and overhead), and on the reader-side (to allow batch-reads that reduce per-read overhead and increases cache-efficiency).

- **bestEffortTelemetry** - this is a variant to Telemetry, and applies to use cases where all the samples do not need to be absolutely reliably delivered, and which allows for a lighter, more efficient protocol for distribution over the network.

It is important to properly characterize the various kinds of data (telemetry, state, event, video, vibration) as they imply not only how the Data River treats each kind of data, but also how it impacts the application business logic that produces and/or consumes that (kind of) data.

For example, although event data guarantees ordered and reliable delivery of all events, this kind of data implies flow-control between a writer and all readers (a writer can be blocked due to a slow reader, even when only one of many readers is slow). This impacts both the autonomy of the writing-application (dealing with the behavior of others), and poses requirements of swift consumption on each reader-application.
In addition, telemetry as well as state data does not imply flow-control, and therefore preserves the autonomy of producing/consuming applications. This is because a (slow) reader is always provided with the freshest data, and does not necessarily receive all samples.

The ‘state’ kind of data is special in the sense that this data will be automatically provided to new/emerging consuming applications. This both reduces the application complexity of dealing with (desired and/or actual-)state in a system as well as improves that system’s robustness as preserving the availability of that state data is not tight to the (correct behavior of) application’s business-logic.

The QoS profile for video data uses a dedicated communication channel in the middleware that has a highly reduced network queue size. This avoids memory exhaustion in overload cases, when the writer of the data produces data at a higher rate than the middleware’s network service can send over the wire. The vibration profile on the other hand, exploits a large network queue to reduce the overhead of sending a large number of small samples (by facilitating packing at the writer and batch reads in the reader).
This section describes how to use the Edge SDK to:

- create a Thing
- read data from the Data River
- write data to the Data River

The basic steps are explained, and a simple real-world scenario is described.

### 3.1 The Basics

The basic steps to read and write to the Data River:

- **Step 1:** Data River Configuration
- **Step 2:** Data River Instantiation
- **Step 3:** Register TagGroup
- **Step 4:** Register ThingClass
- **Step 5:** Create Reader Thing
- **Step 6:** Create Writer Thing
- **Step 7:** Create Thing with Auto Generated Unique ID
- **Step 8:** Write Data
- **Step 9:** Read Data

**Note:** The example code snippets provided are coded for TagGroups defined using JSON. Slight modifications to the application code is required for applications using Google protocol buffers (GPB) to define TagGroups. For more details, see Google Protocol Buffer (GPB) TagGroup Definitions.
3.1.1 Data River Configuration

By default, a Data River configuration file is provided with the Edge SDK installation. The default configuration is provided with working default values.

In order to facilitate the setup of an Edge SDK environment, the Edge SDK default configuration file provides an override mechanism, that allows users to easily modify commonly used settings. A brief explanation of each setting is included in the file.

For each setting, the `override` attribute indicates whether the setting overrides an existing environment variable.

- **IF `override="false"`**
  - If an environment variable with the same name exists, the environment variable value will be used, and the Data River configuration file value will be ignored.
  - If an environment variable with the same name does not exist, the Data River configuration file value will be used.

- **IF `override="true"`**
  - The value specified in the Data River configuration file will be used.

Example: Override DDS_DOMAINID

```xml
<!-- Value to select the proper DDS domainID, default is 0,
no communication will take place between (different) DDS-domains
so domains can be seen as both a communication- as well as a
semantics-boundary -->
<setting name="DDS_DOMAINID" override="true" value="16" />
```

In some cases, it may be necessary to use a custom Data River configuration file. For example, the case where a Data River application requires data security. For more details on security, see *Securing the Data River*.

3.1.2 Data River Instantiation

Edge SDK applications require a `DataRiver` instance, that is instantiated using a factory method.

`DataRiver` instantiation using default configuration file:

```cpp
DataRiver river = DataRiver::getInstance();
```

`DataRiver` instantiation using custom configuration file URI:

```cpp
DataRiver river = DataRiver::getInstance(
    "file:///home/usr/ADLINK/EdgeSDK/1.2.x/etc/config/datariver_ospl_config_secure_v1.2.xml");
```
3.1.3 Register TagGroup

A Thing refers to a ThingClass and a ThingClass can refer to one or more TagGroups. The TagGroups for a Thing must be registered with the appropriate DataRiver registry.

3.1.3.1 TagGroup JSON Definition

- A Thing’s TagGroup definition is defined in JSON format and can be supplied as a string or file URI.

  Example: TagGroup definition

```json
{
    "name": "MyTagGroup",
    "context": "com.company.example",
    "version": "v1.0",
    "qosProfile": "telemetry",
    "description": "My TagGroup",
    "tags": [{
        "name": "message",
        "description": "Sample telemetry message",
        "kind": "STRING",
        "unit": "Text"
    }]
}
```

- Register the TagGroup(s) with a TagGroupRegistry. Add the TagGroupRegistry to the DataRiver instance.

```cpp
JSonTagGroupRegistry tgr;
tgr.registerTagGroupsFromURI("file://./definitions/TagGroup/com.company.example/MyTagGroup.json");
river.addTagGroupRegistry(tgr);
```

3.1.4 Register ThingClass

The ThingClass for a Thing must be registered with the appropriate DataRiver registry.

- A ThingClass definition is in JSON format. A ThingClass references a TagGroup by using the name, context and version. An array of TagGroups can be specified to receive data from multiple sources.

  Example: Reader ThingClass definition

```json
{
    "name": "MyReader",
    "context": "com.company.example",
    "version": "v1.0",
    "description": "My kind of Thing",
    "inputs": [
        {
            "name": "MyInput",
            "tagGroupId": "MyTagGroup:com.company.example:v1.0"
        },
        {
            "name": "MyInput2",
            "tagGroupId": ["MyTagGroup:com.company.example:v1.0", "MyTagGroup2:com.company.example:v1.0"]
        }
    ]
}
```

(continues on next page)
Example: Writer \texttt{ThingClass} definition

\begin{verbatim}
{
  "name": "MyWriter",
  "context": "com.company.example",
  "version": "v1.0",
  "description": "Kind of Thing used to write a message",
  "outputs": [
    {
      "name": "MyOutput",
      "tagGroupId": "MyTagGroup:com.company.example:v1.0"
    }
  ]
}
\end{verbatim}

• Register \texttt{ThingClass} with a \texttt{ThingClassRegistry}. Add the \texttt{ThingClassRegistry} to the \texttt{DataRiver} instance.

\begin{verbatim}
JSonThingClassRegistry tcr;
tcr.registerThingClassesFromURI("file://./definitions/ThingClass/com.company.example/MyWriterThingClass.json");
tcr.registerThingClassesFromURI("file://./definitions/ThingClass/com.company.example/MyReaderThingClass.json");
river.addThingClassRegistry(tcr);
\end{verbatim}

\subsection{3.1.5 Create Reader Thing}

A \texttt{Thing} is created on the \texttt{DataRiver} using a JSON object that represents its configuration properties.

• A \texttt{Thing} configuration properties file is in JSON format. It includes the \texttt{ThingId [id]}, \texttt{ContextId} and \texttt{ThingClass [classId]}.

Example: Reader \texttt{Thing} configuration properties

\begin{verbatim}
{
  "id": "my-reader-thing-id",
  "classId": "MyReader:com.company.example:v1.0",
  "contextId": "MyReader",
  "description": "The actual Thing that read messages"
}
\end{verbatim}

• Create a \texttt{ThingProperties} object and create the \texttt{DataRiver Thing}.

\begin{verbatim}
string thingPropertiesUri = argv[0]; // e.g. "file:///./config/MyReaderProperties.json"
JSonThingProperties tp;
tp.readPropertiesFromURI(thingPropertiesUri);
Thing readerThing = river.createThing(tp);
\end{verbatim}

\textbf{Note:} In all examples, the Thing properties JSON file is in a directory named ‘config’, and the JSON files for
TagGroup and ThingClass in a directory named ‘definitions’. This is to distinguish the application-specific Thing properties from the reusable TagGroup and ThingClass definitions. The URI for the Thing properties JSON file is therefore typically not hard-coded in a Thing’s implementation, but provided as a configuration parameter so that multiple instances of a Thing can be created with their own properties (id, contextId and description).

Note: In the current release of the Edge SDK, only the file scheme is supported in an URI for registering a TagGroup, ThingClass or Thing. Support for other schemes will be added in future releases.

3.1.6 Create Writer Thing

Creating a Thing that writes data to the Data River is very similar to creating the reader.

- A Thing configuration properties file is in JSON format. It includes the ThingId [id], ContextId and ThingClass [classId].

  Example: Writer Thing configuration properties

```json
{
   "id": "my-writer-thing-id",
   "classId": "MyWriter:com.company.example:v1.0",
   "contextId": "MyWriter",
   "description": "The actual Thing that writes messages"
}
```

- Create a ThingProperties object and create the DataRiver Thing.

```cpp
string thingPropertiesUri = argv[0]; // e.g. "file:///./config/
MyWriterProperties.json"
JSONThingProperties tp;
   tp.readPropertiesFromURI(thingPropertiesUri);
   Thing writerThing = river.createThing(tp);
```

3.1.7 Create Thing with Auto Generated Unique ID

A Universally Unique Identifier (UUID) can be automatically generated for a Thing.

The configuration for creating the Thing is passed as a JSON object. To generate the UUID automatically, the [id] property string should include: _AUTO_. The _AUTO_ string/substring will be replaced with a generated UUID.

Generating UUID for Thing example 1:

```json
{
   "id" : "_AUTO_",
   "classId": "MyReader:com.company.example:v1.0",
   "contextId": "MyReader",
   "description": "The actual Thing that read messages"
}
```

Generating UUID for Thing example 2:

```json
{
   "id" : "abc_AUTO_",
   "classId": "MyReader:com.company.example:v1.0",
}
```
(continues on next page)
3.1.8 Write Data

The writer Thing writes data to the DataRiver. The data object is of type IOT_NVP_SEQ (IoT name-value pair sequence). In this code fragment a string value ‘My data!’ of type string with (Tag) name ‘message’ is sent, as defined in the TagGroup definition for ‘MyTagGroup’.

```cpp
IOT_VALUE message_v;
message_v.iotv_string("My data!");
IOT_NVP_SEQ nvp_seq = { IOT_NVP(string("message"), message_v) };
writerThing.write("MyOutput", nvp_seq);
```

3.1.9 Read Data

The reader Things can get the data from the DataRiver. The following code snippet shows how to read data using the read function on the Thing. This function gets the name of the input (as used in the ThingClass ‘inputs’ specification) as a required parameter. The optional second parameter can be used to set the maximum time (in milliseconds) the read call should block, and defaults to infinite. In this case we use infinite blocking:

```cpp
vector< DataSample<IOT_NVP_SEQ> > msgs = readerThing.read<IOT_NVP_SEQ>("MyInput");
```

The reader Thing receives the data: the read call returns a vector with data-samples. Each sample is an object of type DataSample<IOT_NVP_SEQ>. DataSamples are parametrized and contain the data, the ThingClass and Thing Id of the sender, the flowId and a timestamp. With a call to the getData member function the actual IOT_NVP_SEQ is returned.

```cpp
const IOT_NVP_SEQs data = msg.getData();
for (const IOT_NVPs nvp : data) {
    if (nvp.name() == "message") {
        cout << nvp.value().iotv_string();
    }
}
```

In this example the read call will return all data that is available on this input. It is possible to set specific selection criteria when reading data from a particular input, by using the concept of a read selector. This will be explained in example scenario 2 in the chapter Discovering Things and Meta-Data.
3.2 Real-world scenario 1: Connecting a sensor

Building on the basics for creating a Thing and read or write data covered in the previous section, this section describes a real-world scenario of implementing a Thing (digital-twin) that represents a sensor in an IoT system. Let’s assume we want to connect a sensor for measuring temperature and a display that show the actual temperature measured by the sensor.

![Diagram](image.png)

Fig. 1: Scenario 1: Connecting a sensor

### 3.2.1 TagGroup and Thing Class Definition

The ThingClass definition is based on what kind of Thing the sensor is.

A new ThingClass can be defined, or an existing ThingClass can be re-used. A ThingClass for a new ‘kind of thing’ should be designed in such a way that it can be re-used in future solutions. The portfolio of existing ThingClasses also contains the TagGroups that are exploited by these ‘kind of things’.

**Example:** ThingClass temperature sensor

```json
{
    "name": "TemperatureSensor",
    "context": "com.adlinktech.example",
    "version": "v1.0",
    "description": "ADLINK Edge SDK Example Temperature sensor",
    "outputs": [{
        "name": "temperature",
        "tagGroupId": "Temperature:com.adlinktech.example:v1.0"
    }]
}
```

A TagGroup for temperature data, is given as an example. The TagGroup named ‘Temperature’ defines the data the sensor is producing and has the QosProfile ‘telemetry’.

The optional range tag allows for the specification of valid range(s) for the number and sequence of number types. An error, E_VALUE_OUT_OF_RANGE gets thrown if an attempt is made to write invalid data.

**Example:** TagGroup temperature

```json
{
    "name": "Temperature",
    "context": "com.adlinktech.example",
    "qosProfile": "telemetry",
    // Optional range tag
    "range": {
        "min": -100,
        "max": 100
    }
}
```

(continues on next page)
"version": "v1.0",
"tags": [{
  "name": "temperature",
  "description": "Temperature",
  "kind": "FLOAT32",
  "unit": "°C",
  "range": [0.0, 45.5]
}]
}

3.2.2 Thing Properties

A Thing’s properties are unique for the specific Thing instance. This is in contrast with the TagGroups and ThingClass definitions, which are re-usable.

The id in the Thing-properties property (the Thing instance Id) is typically set to the serial number of the physical device this Thing represents. The contextId defines the position of the device (instance) in the system (in this case the temperature sensor, part of the system ‘example1’).

Example: Thing properties

```
{
  "id": "79EC6787DA88",
  "classId": "TemperatureSensor:com.adlinktech.example:v1.0",
  "contextId": "example1.sensor",
  "description": "Edge SDK example 1 temperature sensor"
}
```

3.2.3 Application Code

The application code to connect a sensor can be implemented using the following steps:

- Obtain an instance of the DataRiver.
- Register the ThingClass and TagGroups for the sensor(s).
- Instantiate the Thing that writes sensor data. After instantiation, the application code needs to keep the ‘digital-twin’ in sync with the physical sensor by forwarding data from the physical device towards the DataRiver. This is done by mapping the data to an IOT_NVP_SEQ object and using the write method of the Thing object to publish the data.
- Create the Thing instance for the temperature display that reads data from the river.
  - Register the ThingClass ‘TemperatureDisplay’. The ‘TemperatureDisplay’ ThingClass definition uses the TagGroup ‘Temperature’ as tagGroupId for its input, which is the same TagGroup definition the sensor uses as its output.
  - Create an instance of the display Thing, that references the ‘TemperatureDisplay’ ThingClass.
- Now the application code of the temperature display can read temperature data from the DataRiver.

3.2. Real-world scenario 1: Connecting a sensor
4.1 Discovery

As soon as an actual Thing is created, information about this Thing and its ThingClass becomes discoverable for other Things in the system (the Data River takes care of that automatically). Things can respond to specific data or other Things as they become available.

The Edge SDK dynamically discovers other Things (and their data) emerging on the Data River. It allows applications to be notified of discovered ‘entities’ by means of the following types of listeners:

- TagGroupDiscoveredListener - triggered when a new TagGroup is discovered on the Data River
- ThingClassDiscoveredListener - triggered when a new ThingClass is discovered on the Data River
- ThingDiscoveredListener - triggered when a new Thing is discovered on the Data River
- ThingLostListener - triggered when a Thing is disconnected from the Data River

Listeners for notification of new data flows and for notification of new data-samples can be set on a Thing:

- NewDataFlowListener - triggered when a data-flow is detected with a flowId that is not seen before
- PurgedDataFlowListener - triggered when a data-flow is purged; whereas a newdataFlowListener offers awareness of the ‘start of the lifecycle’ of a dataFlow within the system, a PurgedDataFlowListener offers awareness of the ‘end of the lifecycle’ of a dataFlow within the system (i.e. when the Thing that writes the data calls the purge method for this flow)
- DataAvailableListener - triggered when new data samples are available for any input of the Thing

Note: A listener function that uses the default dispatcher should not perform any long-running tasks, because this will block the dispatcher thread. During execution of a listener function, the dispatcher will not handle events until the executing listener function returns and other listeners (including built-in listeners) will not be triggered.

Example scenario 2 shows the code that registers listeners on the Data River. Using a listener on a Thing for notification on new data-samples (as an alternative pattern for the read call) is described in scenario 3. (See Flow ID.)
4.2 Real-world scenario 2: Connect a dashboard

This scenario describes the implementation of a dashboard that displays (device) data that is available on the Data River. Two new concepts will be introduced in this example:

- **Using a Read Selector**
- **Thing Discovery**

For this scenario we assume there is a Data River with a number of temperature sensors (Things of the ThingClass ‘TemperatureSensor’) that publish temperature data (TagGroup ‘Temperature’) for different floors and rooms in a building. Each floor has a dashboard that shows the actual temperatures for all rooms on that floor. When a new temperature sensor is connected, a message will be shown in a notification list on the dashboard showing the description of the new sensor.

![Diagram of a building with temperature sensors on different floors and a Data River connecting them](image)

**Fig. 1: Scenario 2: Connect a dashboard**

### 4.2.1 Using a Read Selector

A read selector allows for the selection of a subset of data.

The **ThingClass** for the temperature sensors (not shown here) defines an output of TagGroup ‘Temperature’. The **contextId** of a sensor (that is typically set when provisioning the system) contains information about the role of the Thing. In this case, the position of the sensor in the building.

The configuration for creating a sensor **Thing** instance:

```json
{
    "id": "734EB7D7ACC0",
    "classId": "TemperatureSensor:com.adlinktech.example:v1.0",
    "contextId": "floor1.room1",
    "description": "Floor 1 - room 1 temperature sensor"
}
```
The `ThingClass` for the dashboard defines the ‘Temperature’ `TagGroup` as input:

```json
{
    "name": "TemperatureDashboard",
    "context": "com.adlinktech.example",
    "version": "v1.0",
    "description": "Example dashboard that shows temperature sensor data",
    "inputs": [
        {
            "name": "temperature",
            "tagGroupId": "Temperature:com.adlinktech.example:v1.0"
        }
    ]
}
```

As the dashboard in this scenario is only interested in data for the current floor, the application code needs to select the right subset of the data. This is done by creating a read selector for this input that selects specific values of the `flowId` of a data-sample, which is the logical context of the data (the `flowId` in this example is provisioned with the default value, the `contextId` of the sensor `Thing`). Whereas in this code fragment a selection on `flowId` is used, the section `Selector API` gives a short overview of other functionality of the `selector` (e.g. to select on `ThingClass`, `ThingId`). A full description of the members of the `Selector` class can be found in the API documentation.

The code in the temperature dashboard creates a selector on input ‘temperature’ for samples with a `flowId` starting with a specific value:

```cpp
string floorId = "floor1"; // e.g. get value from command line parameter
string selection = floorId + ".*";
Thing::Selector selector = thing.select("temperature").flow(selection);
```

The dashboard performs the `read` call on the selector and not on the `Thing` (as was done in previous examples). Because the selector will be re-used for every read call in a loop, the selector is created outside this loop. The `read` call on a selector has a single parameter that is optional, and can be used to specify the blocking time (similar to the second parameter of the `read` call on a `Thing` described in the previous chapter). This example uses a read call with infinite blocking:

```cpp
do {
    // Read data using selector
    vector<DataSample<IOT_NVP_SEQ>> msgs = selector.read<IOT_NVP_SEQ>();

    // Process samples
    for (const DataSample<IOT_NVP_SEQ>& msg : msgs) {
        cout << "Received data from " << floorId << " temperature sensor" << endl;
        // process the data
    }
} while ( ... );
```

4.2.1.1 Select versus Filter

The input of a `Thing`, as defined by its `ThingClass`, describes what data a `Thing` is interested in. The `Data River` will only deliver data-samples matching these input criteria to the `Thing`. In this example the temperature dashboard receives all data of the `Temperature TagGroup`. By using a `selector`, the `Thing` can select (to process) a subset of the data available at its input, i.e. read only Temperature data that matches the configured floor. However, temperature data for other floors will also be delivered and available on this `Thing`’s input, just not processed.

The filter feature makes it possible to use an expression to specify a filter condition related to the input definition, as part of the `Thing` instance properties. When this feature is used, the application only receives data that matches the filter condition. This ‘provisioning of filters’ as part of a `Thing`’s properties allows sets of related yet generic services (Things) to receive only relevant (flows of) data.

4.2. Real-world scenario 2: Connect a dashboard
4.2.1.2 Selector API

*Using a Read Selector* showed how to use a selector to read data with a specific `flowId`. A Selector object has the following member functions for selecting a subset of the data:

- **thingClass** - Select data written by *Things* with the specified *ThingClass*
- **thingId** - Select data from a *Thing* with a specific Id (this function also requires the *ThingClass* as parameter because the Thing Id is only unique within it’s *ThingClass*)
- **flow** - Select data with a specific `flowId`. Wildcard characters (*,?) can be used in the `flowId`: * matches 0..n occurrences of any character; ? matches 1 occurrence of any character (the example above uses the * to select all sensors that have a `flowId` that starts with 'floor1.')
- **maxSamples** - [not supported in this release of the Edge SDK] Can be used to limit the number of data samples that is returned by a read call
- **content** - [not supported in this release of the Edge SDK]

The functions on a selector can be chained, so that in a single line of code multiple selection criteria can be added and data can be read, e.g.:

```cpp
vector<DataSample<IOT_NVP_SEQ> > data =
    thing.select("temperature")
    .thingId(ThingClassId::NIL_ID, "E5784CB5302A")
    .flow(flowId-selection)
    .maxSamples(10)
    .read<IOT_NVP_SEQ>(0);
```

**Note:** The selector class has member functions to add or remove a listener, but in the current release of the Edge SDK this functionality is not available. Support will be added in a future release of the SDK.

4.2.2 Thing Discovery

To show a notification message when new sensors are connected, the temperature dashboard sets up a listener of type `ThingDiscoveredListener`:

```cpp
class SensorDiscoveredListener : public ThingDiscoveredListener {
    void notifyThingDiscovered(const DiscoveredThing& thing) {
        if (thing.getClassId().getName() == "TemperatureSensor") {
            cout << "Temperature sensor: " << thing.getDescription() << endl;
        } else {
            cout << "Incompatible Thing: " << thing.getClassId().getName() << endl;
        }
    }
};
```

**Note:** The check for a compatible sensor in this example is only based on the *ThingClass* name. In a real-world scenario the *ThingClass* context and optionally the version should be included in this check too.

Using this code the listener is added to the Data River instance:

```cpp
auto sensorDiscoveredListener = SensorDiscoveredListener();
dataRiver.addListener(sensorDiscoveredListener);
```
When setting your handler with above API, an Edge SDK provided background thread (the default Dispatcher) will invoke your handler when the corresponding event occurs. For a complete list of event listeners, see *Discovering Things and Meta-Data*.
CHAPTER FIVE

FLOW ID

5.1 Introduction on flowId

Samples of data on the Data River are identified by the thingClass, thingId and the flowId. This allows other Things to select the proper data by filtering or querying on these identifiers.

Unlike thingClass and thingId, which are related to the source of the data (the Thing that writes the data), the flowId is related to the logical meaning of the data flow. The flowId gives context (meaning) to the data that is in a sample. By default the flowId of data is set to the Thing’s contextId, but can be explicitly set by the processing logic of the Thing.

By default, a Thing writes samples for an output as a single data flow, with a flowId that is equal to the contextId of the Thing that writes the data. In many use-cases this will be sufficient, for example in a system where a temperature sensor outputs the temperature of a single physical object. Setting a specific flowId (other than the auto-populated value) is useful in cases where a single Thing generates data flows on multiple objects. By using a specific flow the Thing can write samples for a single output in multiple logical flows.

5.1.1 Use-case for a custom flowId

A use-case for using a custom flowId is a smart-vision system that reads barcodes. The barcode can be used as part of the flowId when writing data (e.g. set flowId to ‘station1.camera1:barcode1’). By creating a data-flow per barcode, the QoSProfile for this data can be set to Telemetry, as only the most recent update is relevant for each barcode. Without using explicit flows (all samples having the same flowId), the data would require the profile ‘event’, because the receiver needs to process all samples to be sure not to miss updates for any barcode. Using a specific flowId enables more efficient data distribution in this use-case: data can be down-sampled without jeopardizing application logic and at the same time this hugely improves both the system efficiency (telemetry is less heavyweight than events) as well as complexity (‘event’ data implies frequency-coupling between the source and destination of that flow). For more details on the characteristics of the ‘telemetry’ and ‘event’ kinds-of-data, see Introduction.
5.2 Real-world scenario 3: A derived value processing service

This examples shows a use-case for a custom flowId.

A Derived Value Service augments existing data by publishing values derived from original ones. There are two fundamental methods:

1. Publishing a derived value alongside the original value. This section describes the implementation of a Thing using this method, and a reader that uses both the original and derived values.

2. Deriving a new (typically higher-level) data-model.

This example also describes using TypeDefinitions to define a re-usable set of Tags that can be shared by multiple TagGroups, see Use of TypeDefinitions.
5.2.1 Derived Value Service Thing Implementation

We assume there is a Data River with connected Things that publish location data (e.g. the GPS location of a delivery truck). The derived value service that will be implemented in this scenario is a Thing that calculates the distance of the truck from the warehouse and the (simulated) estimated time of arrival (ETA) for the truck given its location and current traffic situation. For all truck-location data samples that are published to the Data River, the derived value service Thing publishes the distance and ETA information to the Data River as a derived value from the location data.

The ThingClass for our derived value service could look like this (the definitions for the TagGroups ‘Location’ and ‘Distance’ are omitted here):

```
{
    "name": "DistanceService",
    "context": "com.adlinktech.example",
    "version": "v1.0",
    "description": "ADLINK Edge SDK Example Derived Value Service that calculates distance from a specific point for incoming GPS location samples",
    "inputs": [{
        "name": "location",
        "tagGroupId": "Location:com.adlinktech.example:v1.0"
    }],
    "outputs": [{
        "name": "distance",
        "tagGroupId": "Distance:com.adlinktech.example:v1.0"
    }]
}
```

The Thing that is instantiated from the ‘TruckDistanceService’ ThingClass registers a DataAvailableListener listener to receive incoming samples, that will be of the ‘Location’ TagGroup. After calculating the distance and the simulated ETA, the Thing writes the data as new sample of the ‘Distance’ TagGroup, using with the same flowId as the original data (so that it can be correlated to the location by the reader):

```c++
for (auto& locationMessage : locationMessages) {
    string myLocationFlowId = locationMessage.getFlowId();
    IOT_VALUE dist_v;
    dist_v.iotv_float64(distance);
}
```

(continues on next page)
IOT_VALUE eta_v;
eta_v.iotv_float32(eta);
IOT_VALUE timestamp_v;
timestamp_v.iotv_uint64(timestamp);

IOT_NVP_SEQ distanceData = {
    IOT_NVP(string("distance"), dist_v),
    IOT_NVP(string("eta"), eta_v),
    IOT_NVP(string("timestampUtc"), timestamp_v)
};

myThing.write("distance", myLocationFlowId, distanceData);

Note: The ‘Distance’ data sample includes a timestamp. The timestamp value is copied from the incoming ‘Location’ sample and is the timestamp of the GPS location for the truck. In the dashboard, described in the next section, this timestamp is displayed with the data to show the actuality of the data.

5.2.2 Reader implementation

The Thing that subscribes to the location and distance data on the Data River (e.g. a dashboard in the warehouse that shows the delivery trucks on a map) is now able get distance and ETA data that is related to a received location sample. The ThingClass definition for the location dashboard used in the example code:

```
{
    "name": "LocationDashboard",
    "context": "com.adlinktech.example",
    "version": "v1.0",
    "description": "Example Dashboard that displays location and distance data",
    "inputs": [{
        "name": "location",
        "tagGroupId": "Location:com.adlinktech.example:v1.0"
    }, {
        "name": "distance",
        "tagGroupId": "Distance:com.adlinktech.example:v1.0"
    }]
}
```

Once the reader Thing is instantiated, the program waits for any location and position data. In this example a loop with non-blocking read calls (parameter blocking time set to 0) is used to get the data. The read calls return all available samples for the specified input (first parameter in the call). Because the derived value service Thing in this example uses the flowId from the incoming Location samples for writing the Distance samples, the dashboard can easily correlate the incoming location and distance samples based on the flowId. The dashboard displays the last known distance (a derived value) and related timestamp of each truck and ‘annotates’ that with the last-known location (an ‘original’ value) of that truck. By also showing the timestamp of the latest received location you can observe if there are (significant) delays in either of the ‘original’ and ‘derived’ flows about a truck.

Both location and distance data is of kind ‘telemetry’, so only the last sample is relevant to the dashboard (because this reflects the most recent situation). Therefore the reader is optimized to only read data when an update of the (console) output to the user is required, and the thread is suspended between updates.

5.2. Real-world scenario 3: A derived value processing service
do {
    // Retrieve and process location samples
    vector<DataSample<IOT_NVP_SEQ> > locationSamples =
        m_thing.read<IOT_NVP_SEQ>("location", 0);
    for (const DataSample<IOT_NVP_SEQ>& sample : locationSamples) {
        processLocationSample(sample);
    }

    // Retrieve and process distance samples
    vector<DataSample<IOT_NVP_SEQ> > distanceSamples =
        m_thing.read<IOT_NVP_SEQ>("distance", 0);
    for (const DataSample<IOT_NVP_SEQ>& sample : distanceSamples) {
        processDistanceSample(sample);
    }

    // Update output
    displayStatus();

    // Sleep before next update
    this_thread::sleep_for(chrono::milliseconds(READ_DELAY));
} while ( ... );

5.2.3 Listener Pattern

Whereas the previous example scenario used the read method on the reader Thing instance to get data from the Data River, the derived value service in this example uses the listener pattern to get data. Registering a listener that is triggered when new data is available can be done by:

```cpp
class GpsSensorDataListener : public DataAvailableListener<IOT_NVP_SEQ> {
    void notifyDataAvailable(const std::vector<DataSample<IOT_NVP_SEQ>>& data) {
        // process data
    }
}

auto gpsDataReceivedListener = GpsSensorDataListener();
m_thing.addListener(gpsDataReceivedListener);
```

Because in this code fragment no dispatcher is provided in the addListener call, the Edge SDK’s default dispatcher will be used. The section Custom Dispatcher below describes how to override the default dispatcher and use your own (which is actually used in the Distance Service implementation in this scenario).

5.2.4 Custom Dispatcher

When registering a listener on the Data River (for discovery) or on a Thing instance (for receiving data samples or notification on new or purged flows) either a default or custom dispatcher can be used. The default dispatcher will be started in a new thread by the Edge SDK and will call the listeners when corresponding events occur. Alternatively, a custom dispatcher can be provided, which makes it possible to handle events in the main thread of the application:

```cpp
// Create a custom dispatcher object
Dispatcher dispatcher = Dispatcher();

// Add listener for new data-samples (using custom dispatcher)
auto gpsDataReceivedListener = GpsSensorDataReceivedListener();
m_thing.addListener(gpsDataReceivedListener, dispatcher);
```

(continues on next page)
// Process events with our dispatcher
do {
  // block call for 1000ms
  dispatcher.processEvents(1000);
} while (/* condition */);

// Remove listener
m_thing.removeListener(gpsDataReceivedListener, dispatcher);

5.2.5 Use of TypeDefinitions

A TypeDefinition is a definition of a list of Tags that can be re-used over multiple TagGroups or multiple Tags within a single TagGroup. It is also possible to use TypeDefinitions recursively, having Tags defined in a TypeDefinition refer to another TypeDefinition.

In this example scenario, a TypeDefinition for GPS coordinates is created with the name ‘Coordinates’ (see the definition in the next json fragment below). The TagGroup ‘Location’ uses this TypeDefinition for its Tag ‘location’.

Note: The ‘kind’ of a Tag that references a TypeDefinition is ‘NVP_SEQ’, as the Tag holds a list of Tags of type IOT_NVP.

```json
{
  "name": "Location",
  "context": "com.adlinktech.example",
  "qosProfile": "telemetry",
  "version": "v1.0",
  "description": "ADLINK Edge SDK Example Location TagGroup",
  "tags": [
    {
      "name": "location",
      "description": "GPS Location",
      "kind": "NVP_SEQ",
      "unit": "n/a",
      "typedefinition": "Coordinates"
    }
  ]
}
```

A TypeDefinition needs to be defined in the same json file where the TagGroups it is defined in. Multiple TagGroups and TypeDefinitions can be included in a single json file by using an array at the top-level. A TypeDefinition may be defined after the TagGroup (or other TypeDefinition) that refers to it.

```json
[ {
  "name": "Location",
  "context": "com.adlinktech.example",
  "qosProfile": "telemetry",
  "version": "v1.0",
  "description": "ADLINK Edge SDK Example Location TagGroup",
  "tags": [
    {
      "name": "latitude",
      "description": "GPS Latitude",
      "kind": "FLOAT32",
      "unit": "°"
    }
  ]
}, {
  "typedefinition": "Coordinates",
  "tags": [{
    "name": "latitude",
    "description": "GPS Latitude",
    "kind": "FLOAT32",
    "unit": "°"
  }, {
    "name": "longitude",
    "description": "GPS Longitude",
    "kind": "FLOAT32",
    "unit": "°"
  }]
}
```

(continues on next page)
To construct a data sample of the ‘Location’ TagGroup, a structure of nested IOT_NVP values needs to be created. The example implementation uses this code (in the GPS sensor class) to create the data that will be written to the Thing’s output:

```c++
IOT_VALUE lat_v;
lat_v.iotv_float32(locationLat);
IOT_VALUE lng_v;
lng_v.iotv_float32(locationLng);

IOT_VALUE location_v;
location_v.iotv_nvp_seq({
    IOT_NVP(string("latitude"), lat_v),
    IOT_NVP(string("longitude"), lng_v)
});

IOT_VALUE timestamp_v;
timestamp_v.iotv_uint64(timestamp);

IOT_NVP_SEQ sensorData = {
    IOT_NVP(string("location"), location_v),
    IOT_NVP(string("timestampUtc"), timestamp_v)
};
```

Reading data from a Tag that refers to a TypeDefinition can be done by iterating over the value of that Tag, which is of type IOT_NVP_SEQ:

```c++
// Loop over values in top-level type for Location TagGroup
for (const IOT_NVPs nvp : locationData) {
    if (nvp.name() == "location") {
        // Loop over values in location Tag that is of
        // type Coordinates (the TypeDefinition)
        for (const IOT_NVPs locationNvp : nvp.value().iotv_nvp_seq()) {
            if (locationNvp.name() == "latitude") {
                truckLocationLat = locationNvp.value().iotv_float32();
            } else if (locationNvp.name() == "longitude") {
                truckLocationLng = locationNvp.value().iotv_float32();
            }
        }
    } else if (nvp.name() == "timestampUtc") {
        timestamp = nvp.value().iotv_uint64();
    }
}
```

In case the top-level type of a TagGroup is a TypeDefinition (i.e. the TagGroup only consists of Tags defined in this TypeDefinition), the list of Tags can be omitted in the TagGroup definition and replaced by a reference to the TypeDefinition. The following json fragment shows the syntax.

```json
[ {
    "name": "ExampleTagGroup",
    "description": "Example TagGroup",
    "kind": "EXAMPLE",
    "unit": "Example Unit"
} ]
```

5.2. Real-world scenario 3: A derived value processing service
5.3 Real-world scenario 2A: Connect a dashboard using FlowId filter

This scenario is the same as the Real-world scenario 2, except that the flowId feature is used to filter the incoming data.

In the Real-world scenario 2 example, the application code uses a read selector to select the desired subset of data. (See Discovering Things and Meta-Data.)

In the Real-world scenario 2A example, the flowId filter is used to filter the incoming data.

5.3.1 FlowId Filter

The flowId filter in the Thing properties ensures that the Thing subscribes only to data matching a filter expression. Applications will ONLY get data that matches their interest by exploiting the fact that flowId’s are typically expressing the context of data.

The temperature dashboard in this scenario is only interested in data for the current floor. A flowId filter is (optionally) defined by creating an inputSettings property.

A Thing properties configuration with a flowId filter for a Thing:

```
{
  "id": "8838c6bf-43dd-4534-9be6-47f190537b23",
  "classId": "TemperatureDashboard:com.adlinktech.example:v1.0",
  "contextId": "floor1.dashboard1",
  "description": "Dashboard Thing that shows temperature data from sensors on floor1",
  "inputSettings": {
    "name": "temperature",
    "filters": {
      "flowIdFilters": ["floor1.*"]
    }
  }
}
```

The temperature dashboard performs the read call on the Thing.
5.3.2 `flowIdFilters` expressions

The syntax for expressions that can be used in the `flowIdFilters` field in a Thing properties definition:

- The `flowIdFilters` consists of 1 or more filter strings separated by a comma
- A wildcard character (*,?) can be used: * matches 0..n occurances of any character; ? matches 1 occurrence of any character
- Examples of valid expressions for `flowIdFilters`:
  - “flowIdFilters”: [“flow1”]
  - “flowIdFilters”: [“flow2.abc”]
  - “flowIdFilters”: [“flow2”, “id*”]
  - “flowIdFilters”: [“flow1?”, “flow2*”]
This section contains a description of real-world use-cases for the Edge SDK.

Scenario 4 covers a generic gateway service that forwards Thing-data to a connected system.

Scenario 5 describes an implementation of a browser tool that displays meta-data for Things that are discovered on the Data River.

6.1 Real-world scenario 4: A Gateway Service

This example describes the concepts behind a gateway service that forwards (replicates) Thing-data to a connected subsystem, e.g. the Microsoft Azure cloud. The task of this service is to replicate data from the Data River to the connected system: basically the digital-twins on the Data River are replicated within that system. The purpose of this example scenario is:

- Show how to deal with dynamic inputs.
- Explain the usage of \texttt{read\_next} for fair processing all data on such a dynamic input.
- Understanding different aspects of each individually received sample.

The scenario is based on the smart-camera use-case that is mentioned earlier in this guide in the section on \textit{Flow ID} where camera(s) publish (telemetry-type) observations of boxes that each have a unique bar-code. The system in this example consists of two stations: \textit{Station 1} and \textit{Station 2}. A station in this context is a location of a ‘pallet’ where these boxes are going to be stacked upon.

Station 1 is a basic setup:

- A single smart-camera that reads barcodes and reports the \((x,y,z)\) position of the box for each barcode. The camera \textit{Thing} has a contextId value ‘station1.camera1’
- A light-sensor that periodically publishes the illuminance of the environment and raises an alarm-event when its too dark. The contextId for this sensor \textit{Thing} is ‘station1.lightsensor’

Station 2 is more sophisticated has 2 camera’s to increase the coverage i.e. robustness:

- Two smart-cameras, with contextId ‘station2.camera1’ and ‘station2.camera2’
- A light-sensor (same behavior as the camera from station 1) with contextId ‘station2.lightsensor’
6.1.1 Dynamic Inputs

Typically a gateway service will forward a selection of the data in a system. This selection could for example be based on a configured list of TagGroups or by discovering available TagGroups and apply logic to select the ones that should be forwarded. In this example we assume that all TagGroups that have context ‘com.adlinktech.example’ need to be forwarded by the gateway service. The resulting ThingClass definition for the gateway service Thing is:

```json
{
    "name": "GatewayService",
    "context": "com.adlinktech.example",
    "version": "v1.0",
    "description": "ADLINK Edge SDK Example Gateway Service",
    "inputs": [{
        "name": "dynamicInput",
        "tagGroupId": "*:com.adlinktech.example:*"
    }]
}
```

For both the name and the version parts of the TagGroup reference a wildcard ‘*’ is used, so this input will match data for any version of any TagGroup that is in the ‘com.adlinktech.example’ context. This makes the input of the gateway service a dynamic input. Once the gateway service is running, it starts receiving data for the specified subset of TagGroups from Things that are already connected to the Data River. The gateway service will also automatically discover new data flows from emerging Things that are connected in a later stage and it will receive data for the flows that have a TagGroup matching the expression of the input.
The syntax for expressions that can be used in the TagGroupId field in a ThingClass definition:

- The expression consists of three components name, context and version, using a colon as the separator: name:context:version
- All three components need to be provided
- A wildcard character (*,?) can be used within any of the three components: * matches 0..n occurrences of any character; ? matches 1 occurrence of any character
- Examples of valid expressions for TagGroupId:
  - *:com.adlinktech.example:v1.0
  - *:com.adlinktech.*:v1.?
  - Temperature:*:*
  - *Temperature*:.*:*
  - *:*:v1.0
  - *:*:*

### 6.1.2 Processing data using `read_next`

For reading data from its dynamic input, the gateway service uses the `read_next` method. Although technically the data could also be read by using the regular `read` method or by using a `DataAvailableListener`, reading data with `read_next` helps in ‘fair’ processing all data on a dynamic input when multiple Things produce data of TagGroups that match the input. The `read_next` will automatically ‘cycle’ over the data flows, returning samples from the next flow that has data available (in case the default infinite blocking time is used). This prevents starvation of data-flows, which is e.g. likely to occur when dealing with a combination high- and low-frequency dataflows in a system.

```cpp
vector<DataSample<IOT_NVP_SEQ>> msgs =
  m_thing.read_next<IOT_NVP_SEQ>("dynamicInput");
```

Data from the Data River might come-in at a higher volume than the connected subsystem can handle, e.g. due to limitations in bandwidth or update frequency. In this case batching and down-sampling the data may be required. For data-flows with data of the kind Telemetry or State, we don’t want to bother the Things that produce the data with this problem. Because for these kinds-of-data the most recent sample will reflect the latest value or state, whenever the gateway can send data, it should send the freshest data available (rather than older samples that the gateway service may have received).

Using `read_next` helps implementing this pattern in the gateway service: the service reads data using `read_next` and stores the samples in a batch per data-flow (when a Thing produces a single flow this would result in a batch per Thing). When the batch is ‘full’ (e.g. reached the maximum message size that can be forwarded) the oldest samples in the batch can be dropped and will be replaced by the newer data, until the batch is due to be sent to the subsystem. Assuming the gateway service will treat data from each source equally important, it can just perform the `read_next` at any pace to cycle over the flows for its dynamic input and fill the batches with the data. Because a batch for a data-flow will always be filled with the most recent data, it can be forwarded at any time, when there is capacity available to send data to the connected subsystem.

For data of kind Event, it may be inevitable to have flow-control when it is high-volume data (more than the uplink to the external system can handle). This will impact the Thing that writes the data, as described in the section on QoS Profiles in the Introduction.
6.1.3 Meta-data for received data

The example implementation for the gateway service that is included in the Edge SDK outputs a number of fields for each incoming data-flow:

- The contextId that identifies the Thing that writes the data, and its ThingClass and (Thing) Id
- The flowId of the data-flow
- The name of the TagGroup and its associated QoS
- The sample count for the data-flow

In a real-world implementation of this gateway service scenario, the information shown here can be used to replicate Things that live on the Data River in a connected subsystem. The discoverable meta-data for Things, ThingClasses and TagGroups can be used to create the Thing instances in the connected system. For batching the data in the gateway service (as described in the previous section), the key for a batch is constructed from (1) the Id of the Thing that writes the data, (2) the TagGroup associated with the data and (3) the flowId of the data-flow. Where this example code only counts the number of samples for each key, in a real-world scenario the data samples would be batched and sent-over to the connected system.

6.1.3.1 Flow State

Each incoming data-sample has a flow-state that indicates the state of the data-flow a sample is a part of.

The state of a flow can be either:

- alive: Where the Thing that writes the data indicates that more data will be written for this flow.
- purged: No more data will be written for this data-flow.

In the smart-camera example implementation, when a camera loses track of an observed object (simulated by loosing track of a barcode after a fixed duration), the following purge method is called:

```cpp
do {
    IOT_NVP_SEQ data = {
        IOT_NVP(string("barcode"), barcode_v),
        // set other fields in data sample
    };

    // Write data
    m_thing.write("observation", getFlowId(barcode), data);
} while (/* check elapsed time */);

// Purge the data-flow
m_thing.purge("observation", getFlowId(barcode));
```

When the writer purges a data-flow, a receiver (i.e. the gateway service) gets an updated flow-state value in its samples. If there is unread data available in the receiver Thing’s queue, any read call samples made after the purge has the flow state set to purged. If all data was already read when the purge happened, the reader will get an empty sample with the ‘purged’ flow state.

The gateway service is using the flow-state to determine if a camera has lost track of an observed object.

The code for getting the flow-state of a flow in the gateway service implementation:

```cpp
auto flowState = msg.getFlowState();
```

(continues on next page)
6.2 Real-world scenario 5: Dynamic Browsing

This scenario describes the implementation of a tool that displays meta-information for all *Things*, and all *ThingClass* and *TagGroup* definitions that are discovered on the *Data River*.

Two approaches are supported to get this data:

1. add listeners for discovery of the meta-data (see *Discovering Things and Meta-Data*).
2. exploit the registries for discovered *Things*, *ThingClasses* and *TagGroups*. This example is implemented using approach (2).

To have some sample data in the Data River, the code for this scenario includes two applications (‘Generator A’ and ‘Generator B’) that both create a number of *Things* that represent sensors in the generator system. This code is similar to the code from scenario 1 and details will not be covered here. The remainder of this section will focus on the code of the ThingBrowser application, the application that displays the meta-information.

---

![Diagram](image)

Fig. 2: Scenario 5: Dynamic Browsing
6.2.1 Registries

When a Thing is created using the Edge SDK, it gets registries for locally registered entities (Things, ThingClasses and TagGroups) and it gets registries for discovered entities. The registries for locally registered entities are:

- **TagGroupRegistry** - a registry that contains all locally registered TagGroups
- **ThingClassRegistry** - contains locally registered ThingClasses
- **ThingRegistry** - contains Things that are created locally

The registries for discovered entities are automatically populated with discovered meta-data:

- **DiscoveredTagGroupRegistry** - a registry that is populated by discovered TagGroups
- **DiscoveredThingClassRegistry** - populated with discovered ThingClasses
- **DiscoveredThingRegistry** - populated with Things discovered on the Data River

The ThingBrowser application registers a listener for discovered Things, which is called when a Thing is discovered on the Data River.

The following code example exploits the registries for discovered ThingClasses and TagGroups to retrieve the metadata for discovered Things. When a Thing is discovered, its ThingClass and TagGroups used in this ThingClass may not be discovered yet. The ThingBrowser will retry getting these for a maximum of 3 seconds:

```cpp
class NewThingDiscoveredListener : public ThingDiscoveredListener {
    DataRiver &m_dataRiver;
    DiscoveredThingClassRegistry m_thingClassRegistry = createThingClassRegistry();

    // Constructor
    NewThingDiscoveredListener(DataRiver &dataRiver): m_dataRiver(dataRiver) {
    }

    DiscoveredThingClassRegistry createThingClassRegistry() {
        return m_dataRiver.getDiscoveredThingClassRegistry();
    }

    // New Thing discovered
    void notifyThingDiscovered(const DiscoveredThing& thing) {
        bool thingClassFound = false;
        int retryCount = 30;

        while (!thingClassFound && retryCount-- > 0) {
            Try {
                ThingClass thingClass = m_thingClassRegistry.findThingClass(
                    thing.getClassId().getName() +
                    "":"" + thing.getClassId().getContext() +
                    "":"" + thing.getClassId().getVersionTag());
                cout << "ThingClass: " << thingClass.getId().getName() << endl;
                thingClassFound = true;
            }
            catch(const InvalidArgumentError&) {
                // ThingClass not found
            }

            // Sleep 100ms before retry
            this_thread::sleep_for(chrono::milliseconds(100));
        }
    }
}
```

6.2. Real-world scenario 5: Dynamic Browsing
For the inputs and outputs of the ThingClass, the TagGroups are retrieved from the DiscoveredTagGroupRegistry in a similar way. This code fragment shows getting the input TagGroups:

```cpp
DiscoveredTagGroupRegistry tagGroupReg = dataRiver.getDiscoveredTagGroupRegistry();
vector<InputTagGroup> inputs = thingClass.getInputTagGroups();
for (InputTagGroup input : inputs) {
    bool tagGroupFound = false;
    int retryCount = 30;

    while (!tagGroupFound && retryCount-- > 0) {
        try {
            TagGroup tagGroup = tagGroupReg.findTagGroup(input.getInputTagGroup());
            cout << tagGroup.getName() << ": ";
            tagGroupFound = true;
        } catch (const InvalidArgumentError&) {
            // TagGroup not found
        }

        // Sleep 100ms before retry
        this_thread::sleep_for(chrono::milliseconds(100));
    }
}
```
QOS PROFILE OVERRIDE

To control the Quality-of-Service (QoS), a TagGroup has an associated QoSProfile. The Edge SDK has a set of pre-defined QoSProfiles, which can be extended with custom profiles.

The QoS profile override feature allows for services to override the default QoS profiles for local Reader(s).

The inputs that are defined in a Reader’s ThingClass, can be overridden to use a custom QoS profile in the Reader’s Thing configuration properties.

Custom QoS profile use case examples:

- The telemetry QoS profile is used where the most recent data is more relevant than old data. The keep last history depth for <datareader_qos> is set to 1. However, there may be cases where the keep last history depth value needs to be set at a value > 1. For example, a gateway service might want to apply a > 1 keep last history depth for telemetry type data, to increase the tolerance to handle bursts of data, or cope with reactivity limitations.

- A tool such as the DataRiver browser, might want to have the ability to ‘peek’ into the data, with minimal impact on existing data flows. A custom QoS profile could override the event QoS profile so that it would not block event publishers.

The actions required to override a QoS profile are provided in the following sections.

Note: Example code snippets are taken from example S1_ConnectSensor.

7.1 Create Custom QoS Profile

Steps to create a custom QoS profile:

- Determine the TagGroup of your local Reader. A Reader’s inputs and associated TagGroups are specified in a ThingClass’s json definition file.

  Example: The “tagGroupId” for the “temperature” input is defined in the ThingClass definition file, TemperatureDisplayThingClass.json.

```
{
  "name": "TemperatureDisplay",
  "context": "com.adlinktech.example",
  "version": "v1.0",
  "description": "temperature sensor data",
  "inputs": [{
    "name": "temperature",
    "tagGroupId": "Temperature:com.adlinktech.example:v1.0"
  }
```
(continues on next page)
• Determine which QoS profile requires an override. The \textit{TagGroup} definition specifies which “qosProfile” is used.

Example: The “qosProfile” for the TagGroup defined in TemperatureTagGroup.json is: telemetry

```json
{
    "name": "Temperature",
    "context": "com.adlinktech.example",
    "qosProfile": "telemetry",
    "version": "v1.0",
    "description": "ADLINK Edge SDK Example Temperature TagGroup",
    "tags": [{
        "name": "temperature",
        "description": "Temperature",
        "kind": "FLOAT32",
        "unit": "°C"
    }]
}
```

• Locate QoS profiles
  – QoS profiles are defined as xml files. They are found in the \texttt{EDGE_SDK_HOME/qos} directory.

• Copy an existing QoS profile, and rename it.
  – A copy of telemetry.xml is renamed to: telemetryOverrideKeepLast.xml

• Open custom QoS profile xml in an editor.
  – Example: Open telemetryOverrideKeepLast.xml in a text editor.

• Update the \texttt{<qos_profile name> in your custom QoS profile xml file to match the file name.}

  ```xml
  <qos_profile name="telemetryOverrideKeepLast">
  
  ```

• Update the \texttt{<datareader_qos>} attributes as required for your local reader. The override mechanism can only be applied to the inputs of readers.

Example: Override and set history depth value = 10

```xml
<qos_profile name="telemetryOverrideKeepLast">
  <datareader_qos>
    <durability>
      <kind>VOLATILE_DURABILITY_QOS</kind>
    </durability>
    <reliability>
      <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
    <destination_order>
      <kind>BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS</kind>
    </destination_order>
    <history>
      <kind>KEEP_LAST_HISTORY_QOS</kind>
      <depth>10</depth>
    </history>
  </datareader_qos>
</qos_profile>
```
• Add custom QoS profile xml file to the `EDGE_SDK_HOME/qos` directory.

### 7.2 Update Reader’s `Thing` Configuration Properties

To override the QoS profile for a local Reader’s input, the Reader’s `Thing` configuration properties .json file requires a “qosOverride” entry for the input.

A “qosOverride” entry is specified using the following format:

```json
"inputSettings": [
    {
        "name": "temperature",
        "qosOverride": [{
            "tagGroupId": "Temperature:com.adlinktech.example:v1.0",
            "originalQosProfile": "telemetry",
            "newQosProfile": "telemetryOverrideKeepLast"
        }]
    }
]
```

Where:

• “name” : Name of input to override (defined in reader `ThingClass` definition).

• “qosOverride” : Override the “originalQosProfile” with the “newQosProfile” for a “tagGroupId”.

• “tagGroupId” : tagGroupId for input (defined in reader `ThingClass` definition). The syntax for expressions is the same as that for dynamic inputs. See More Real-World Scenarios.

• “originalQosProfile” : Original QoS profile for `TagGroup` with tagGroupId (defined in `TagGroup` definition).

• “newQosProfile” : Name of custom QoS profile.

**Note:** If the “originalQosProfile” specified does not match the “qosProfile” of the `TagGroup` with “tagGroupId”, the QoS profile will not be applied and a WARNING will be logged.

Example: The “temperature” input is defined in the `ThingClass` definition file, TemperatureDisplayThingClass.json.

```json
{
    "name": "TemperatureDisplay",
    "context": "com.adlinktech.example",
    "version": "v1.0",
    "description": "temperature sensor data",
    "inputs": [{
        "name": "temperature",
        "tagGroupId": "Temperature:com.adlinktech.example:v1.0"
    }]
}
```

Example: Update Reader’s `Thing` Configuration Properties to override the QoS profile for the input named “temperature”.

```json
{
    "id": "ada27527-4345-4242-890f-215ade35500c",
    "classId": "TemperatureDisplay:com.adlinktech.example:v1.0",
    "contextId": "example1.display",
}
```

(continues on next page)
"description": "shows temperature data",
"inputSettings": [
    {
        "name": "temperature",
        "qosOverride": [
            {
                "tagGroupId": "Temperature:com.adlinktech.example:v1.0",
                "originalQosProfile": "telemetry",
                "newQosProfile": "telemetryOverrideKeepLast"
            }
        ]
    }
]
8.1 Overview

Things exchange data in the form of samples (or messages), defined by TagGroups.

The Google Protocol Buffer (GPB) TagGroup feature allows for the specification of TagGroups using google protocol buffers instead of JSON files. Generated GBP data types are used to represent messages in Edge SDK applications.

The TagGroup implementation using JSON files relies on an IOT_VALUE class. Once a TagGroup type is defined in JSON and released, it cannot be changed.

Using GPB has two main advantages. First, it provides you will a native API to your TagGroup data: a class with methods to read and write values by name.

Second, GPB messages can be evolved in a way that allows old applications and new applications to communicate, even if they used different generations of the same definition. In the context of EdgeSDK, an existing TagGroup data type can be updated without breaking existing application code.

When using GPB to define TagGroups, the Edge SDK apis used in the application code will differ slightly from the apis used when TagGroups are defined with JSON.

Note: Currently, the GPB TagGroup feature is only provided for Edge SDK C++.

8.1.1 What are Google Protocol Buffers (GPB)?

The GPB support for Edge SDK is based on GPB proto3. The Google documentation (https://developers.google.com/protocol-buffers/docs/overview) provides an overview of the key concepts:

“Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the “old” format.”

“You specify how you want the information you’re serializing to be structured by defining protocol buffer message types in .proto files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs.”

“The message format is simple – each message type has one or more uniquely numbered fields, and each field has a name and a value type, where value types can be numbers (integer or floating-point), booleans, strings, raw bytes, or even (as in the example above) other protocol buffer message types, allowing you to structure your data hierarchically. You can specify optional fields, required fields, and repeated fields. You can find more information about writing .proto files in the Protocol Buffer Language Guide.”
“Once you’ve defined your messages, you run the protocol buffer compiler for your application’s language on your .proto file to generate data access classes. These provide simple accessors for each field (like name() and set_name()) as well as methods to serialize/parses the whole structure to/from raw bytes – so, for instance, if your chosen language is C++, running the compiler on the above example will generate a class called Person. You can then use this class in your application to populate, serialize, and retrieve Person protocol buffer messages.”

Useful Links

- https://developers.google.com/protocol-buffers/docs/proto3
- https://developers.google.com/protocol-buffers/docs/overview

8.1.2 GPB Limitations

- **Message Size**

  The google documentation indicates that:

  “Protocol Buffers are not designed to handle large messages. As a general rule of thumb, if you are dealing in messages larger than a megabyte each, it may be time to consider an alternate strategy.”
  https://developers.google.com/protocol-buffers/docs/techniques#large-data

  Therefore, GPB is **NOT** the preferred solution for defining TagGroups for larger messages such as video frames or video files. In our investigations, the cost of copying such payloads during serialization/deserialization of Protobuf messages becomes noticeable with large payloads, when compared to using JSON/NVP_SEQ TagGroups.

8.1.3 Examples

Each Edge SDK example has two versions:

- An example version with TagGroup definitions in JSON file format
  
  For example, S1_ConnectSensor

- An example version with TagGroups defined with GPB format
  
  For example, S1_ConnectSensorProtobuf

In addition, a **TypeEvolution** example is provided, where the same application is evolved over 3 generations, with different TagGroup formats.

8.2 GPB TagGroup .proto Files

A TagGroup is defined in a .proto file. The .proto file is then used to generate data access classes using the protoc compiler.

One C++ header file will be generated for each .proto file processed.
8.2.1 Convert existing TagGroup JSON files to .proto files

A json2proto executable is provided to convert an existing TagGroup JSON file to a .proto file.

```
ADLINK JSON to .proto

positional arguments:
  jsonfile TagGroup JSON file

optional arguments:
  -h, --help         show this help message and exit
  -s SUFFIX, --suffix SUFFIX
                     Suffix to append to existing context
  --stdout           Write output to stdout instead of a .proto file
  --outdir OUTDIR    Specify output directory. Default is the directory
                     containing the input file
```

**Note:** Before running the json2proto executable in a terminal, run the Edge SDK config_env_variables script.

8.2.2 Define new .proto files

A TagGroup definition is defined in a .proto file. Examples are provided to demonstrate syntax.

Example: TemperatureTagGroup.proto file

```proto
syntax = "proto3";

// Import the DataRiver descriptor extension
import "adlinktech/datariver/descriptor.proto";

// namespace for data type to be generated
package com.adlinktech.example.protobuf;

message Temperature {
  option(.adlinktech.datariver.tag_group) = {
    qosProfile: "telemetry"
    description: "ADLINK Edge SDK Example Temperature TagGroup"
  };

  float temperature = 1 [{.adlinktech.datariver.field_options} = {
    description: "Temperature"
    unit: "°C"
  }];
};
```

Example: ObservationTagGroup.proto file

```proto
syntax = "proto3";

import "adlinktech/datariver/descriptor.proto";

package com.adlinktech.example.protobuf;
(continues on next page)```
message Observation {
  option(.adlinktech.datariver.tag_group) = {
    qosProfile: "telemetry"
    description: "ADLINK Edge SDK Example Camera observation TagGroup"
  };

  string barcode = 1 {
    description: "Barcode"
    unit: "n/a"
  };

  int32 position_x = 2 {
    description: "x coordinate of position"
    unit: "n/a"
  };

  int32 position_y = 3 {
    description: "y coordinate of position"
    unit: "n/a"
  };

  int32 position_z = 4 {
    description: "z coordinate of position"
    unit: "n/a"
  };
};

8.3 Generate Data Access Classes

For each supported Edge SDK language, a protoc compiler plug-in is required to generate Edge SDK GPB data access classes. The required protoc compiler and protobuf libraries are included with the Edge SDK installation.

8.3.1 Invoking protoc for C++

Usage: protoc [OPTION] PROTO_FILES

Parse PROTO_FILES and generate output based on the options given:

-IPATH, --proto_path=PATH Specify the directory in which to search for imports. May be specified multiple times; directories will be searched in order. If not given, the current working directory is used.

--version Show version info and exit.

-h, --help Show this text and exit.

--encode=MESSAGE_TYPE Read a text-format message of the given type from standard input and write it in binary to standard output. The message type must be defined in PROTO_FILES or their imports.

--decode=MESSAGE_TYPE Read a binary message of the given type from standard input and write it in text format to standard output. The message type must be defined in PROTO_FILES or their imports.

--decode_raw Read an arbitrary protocol message from
standard input and write the raw tag/value pairs in text format to standard output. No PROTO_FILES should be given when using this flag.

--descriptor_set_in=FILES Specifies a delimited list of FILES each containing a FileDescriptorSet (a protocol buffer defined in descriptor.proto). The FileDescriptor for each of the PROTO_FILES provided will be loaded from these FileDescriptorSets. If a FileDescriptor appears multiple times, the first occurrence will be used.

-oFILE, Writes a FileDescriptorSet (a protocol buffer, defined in descriptor.proto) containing all of the input files to FILE.

--descriptor_set_out=FILE defined in descriptor.proto) containing all of the input files to FILE.

--include_imports When using --descriptor_set_out, also include all dependencies of the input files in the set, so that the set is self-contained.

--include_source_info When using --descriptor_set_out, do not strip SourceCodeInfo from the FileDescriptorProto. This results in vastly larger descriptors that include information about the original location of each decl in the source file as well as surrounding comments.

--dependency_out=FILE Write a dependency output file in the format expected by make. This writes the transitive set of input file paths to FILE.

--error_format=FORMAT Set the format in which to print errors. FORMAT may be 'gcc' (the default) or 'msvs' (Microsoft Visual Studio format).

--print_free_field_numbers Print the free field numbers of the messages defined in the given proto files. Groups share the same field number space with the parent message. Extension ranges are counted as occupied fields numbers.

--plugin=EXECUTABLE Specifies a plugin executable to use. Normally, protoc searches the PATH for plugins, but you may specify additional executables not in the path using this flag. Additionally, EXECUTABLE may be of the form NAME=PATH, in which case the given plugin name is mapped to the given executable even if the executable’s own name differs.

--cpp_out=OUT_DIR Generate C++ header and source.

--csharp_out=OUT_DIR Generate C# source file.

--java_out=OUT_DIR Generate Java source file.

--js_out=OUT_DIR Generate JavaScript source.

--objc_out=OUT_DIR Generate Objective C header and source.

--php_out=OUT_DIR Generate PHP source file.

--python_out=OUT_DIR Generate Python source file.
--ruby_out=OUT_DIR
@<filename>

Generate Ruby source file.
Read options and filenames from file. If a relative file path is specified, the file will be searched in the working directory. The --proto_path option will not affect how this argument file is searched. Content of the file will be expanded in the position of @<filename> as in the argument list. Note that shell expansion is not applied to the content of the file (i.e., you cannot use quotes, wildcards, escapes, commands, etc.). Each line corresponds to a single argument, even if it contains spaces.

Note: In order to run protoc in a terminal, configure the environment variables by running the Edge SDK config_env_variables script.

8.3.2 Generated output C++

A .proto file can contain one or more TagGroup definition.

In order to use these tag groups in your C++ code, you must run protoc with both the --cpp_out and --cppdatariver_out parameters. This can be done in separate invocations, or in a single invocation of protoc. Alternately, you can use CMake commands to run protoc as part of a CMake build.

Both the --cpp_out and --cppdatariver_out parameters generated two files for each .proto file. The generated files maintain the base of the .proto file (the name with the .proto extension removed), and generated the following files:

- .pb.h - header for standard C++ access. Generated by --cpp_out
- .pb.cc - source for standard C++ access. Generated by --cpp_out
- .dr.h - header for DataRiver C++ access. Generated by --cppdatariver_out
- .dr.cpp - source for DataRiver C++ access. Generated by --cppdatariver_out

Example:

.proto file: TemperatureTagGroup.proto

DataRiver header file: TemperatureTagGroup.dr.h DataRiver source file: TemperatureTagGroup.dr.cpp Standard header file: TemperatureTagGroup.pb.h DataRiver source file: TemperatureTagGroup.pb.cc

8.3.3 Generating C++ with CMake

Although you can use the protoc command directly, if you are using the CMake build system generator, you can make use of Google-provided CMake commands to generate C++ files from your .proto files. You are not obligated to use CMake. However, it may make building your application easier.

The CMake command to compile .proto files is protobuf_generate. It becomes available after you have imported ThingAPI with the find_package command. Each of the CMakeLists.txt files in the protobuf examples demonstrates the use of this command.

8.3. Generate Data Access Classes
Because `protobuf_generate` only allows specifying one ‘out’ parameter to `protoc` at a time, a typical ThingAPI project will use the command twice: once to specify `--cpp_out`, and once to specify `--cppdatariver_out`. The following code is from the `CMakeLists.txt` file in the `S1_ConnectSensorProtobuf` example:

```cmake
set(PROTO_SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/definitions)  
sset(PROTOC_OUTPUT_DIR ${CMAKE_CURRENT_BINARY_DIR})

# Compile .proto files for C++ protobuf_generate(LANGUAGE cpp  
# PROTOS ${PROTO_SOURCE_DIR}/TemperatureTagGroup.proto OUT_VAR PB_FILES  
# PROTOC_OUT_DIR ${PROTOC_OUTPUT_DIR})

# Compile .proto files for DataRiver C++ extensions protobuf_generate(LANGUAGE cppdatariver  
# PROTOS ${PROTO_SOURCE_DIR}/TemperatureTagGroup.proto OUT_VAR DR_FILES  
# ${CPPDATARIVER_EXTENSIONS} # let protoc know what extension are generated PRO-  
# TOC_OUT_DIR ${PROTOC_OUTPUT_DIR})
```

The initial `set()` commands contain the locations where the `.proto` file is located (`PROTO_SOURCE_DIR`) and where the generated C++ code will be located (`PROTOC_OUTPUT_DIR`).

The first `protobuf_generate` command specifies `--cpp_out` to `protoc` by setting `LANGUAGE` to `cpp`. More than one `.proto` file can be listed after the `PROTOS` keyword. In this case, only one `.proto` file is specified. The `OUT_VAR` keyword names a variable whose contents will contain a list of all file generated. In this case, `PB_FILES` will contain the full path of `TemperatureTagGroup.pb.h` and `TemperatureTagGroup.pb.cc`. The `PROTOC_OUT_DIR` keyword tells `protoc` where to put the generated C++ output.

The second `protobuf_generate` command specifies `--cppdatariver_out` to `protoc`. Most of its keywords and values serve the same purpose as in the first command. The one exception is the text `$({CPPDATARIVER_EXTENSIONS})`. This is variable created by finding the ThingAPI package. It is a convenience variable that contains the following text:

```
GENERATE_EXTENSIONS .dr.h .dr.cpp
```

By using it in the `protobuf_generate` command you are telling the command which file extensions of the files generated by `cppdatariver`. This enables the command to correctly determine which files will be generated and thus set the `DR_FILES` variable.

The rest of the `CMakeLists.txt` is pretty standard. There are two exceptions. First, targets that need access to the generated C++ files must include the `OUT_VAR` variables (`PB_FILES` and `DR_FILES`, in this case) in their list of files to be compiled. For example, you will see this target definition:

```cmake
add_executable(s1_temperaturesensor src/TemperatureSensor.cpp $<PB_FILES> $<DR_FILES>)
```

Second, targets that include the generated header files must know where to find those files. In the `S1` example, you will see the following command:

```cmake
target_include_directories(s1_temperaturesensor PRIVATE $<PROTOC_OUTPUT_DIR>)
```

This tells CMake that include files for the `s1_temperaturesensor` target can be found in the `PROTOC_OUTPUT_DIR` (which was described above). This is the location of the protoc compiler generated C++ code.
8.4 Application Code API

Using the Edge SDK API with Google Protobuf TagGroups is very similar to using the API with JSON TagGroups. This section highlights the things you must do differently.

8.4.1 Accessing Generated C++ classes

In order to access the generated C++ classes, it is sufficient to include the .dr.h file generated by the cppdatariver generator. If you use the CMake generate_protobuf() commands, then the path to the generated header follows the same relative path structure as your .proto has with respect to your project.

For example, a project-relative .proto file path:

```
definitions/TemperatureTagGroup.proto
```

results in an include statement for .dr.h:

```
#include "definitions/TemperatureTagGroup.dr.h"
```

Generated C++ classes are in a namespace generated from the package statement in the .proto file. For example:

Proto file package: com.adlinktech.example

```
Generated C++ namespace for types: com::adlinktech::example
```

You can use appropriate C++ statements to avoid having to type fully qualified names in your code. For example, this will make all types in a generated namespace available for unqualified use:

```
using namespace ::com::adlinktech::example;
```

8.4.2 Registering Protobuf TagGroups

A protobuf TagGroup is registered differently from a JSON tag group. The cppdatariver protoc generator creates a Helper class for each generated C++ class that includes a static method registerWithDataRiver(). For example, if you have a generated C++ class Temperature, the helper class would be TemperatureHelper, and you would register the Temperature tag group with the following statement:

```
DataRiver dataRiver = ...;
TemperatureHelper::registerWithDataRiver(dataRiver);
```

ThingClasses and ThingProperties continue to be registered with JSON files and appropriate JSON registries.

Protobuf tag groups continue to have three-part ‘id strings’ formed from ‘name:context:version’. If not explicitly specified in the .proto file, context is derived from ‘package’ and version is ‘v1.0’. As an example, consider this .proto file:

```
syntax = "proto3";

import "adlinktech/datariver/descriptor.proto";

package com.adlinktech.example.protobuf;

message Temperature {
  option (.adlinktech.datariver.tag_group) = {
    qosProfile: "telemetry"
    description: "ADLINK Edge SDK Example Temperature TagGroup"
  }
}
```

(continues on next page)
Here, the `taggroup` id is ‘Temperature:com.adlinktech.example.protobuf:v1.0’. In the option section of the Temperature message, 'context' and 'version' could be explicitly specified. The following is equivalent to the above, but uses explicit specifications of context and version:

```cpp
message Temperature {
  option(.adlinktech.datariver.tag_group) = {
    qosProfile: "telemetry"
    description: "ADLINK Edge SDK Example Temperature TagGroup"
    context: "com.adlinktech.example.protobuf"
    version: "v1.0"
  };
}
```

### 8.4.3 Use ThingEx instead of Thing

Methods to access protobuf tag group data are all available from a new C++ class, `com::adlinktech::datariver::ThingEx`, for extended Thing. No API methods return a ThingEx. Instead, you merely need to declare a ThingEx variable, and assign it an existing Thing:

```cpp
using namespace com::adlinktech::datariver;
DataRiver dataRiver = ...;

// DataRiver::createThing() returns Thing.
// We can access it as a ThingEx simply by assigning it to a ThingEx
ThingEx newAndImprovedThing = dataRiver.createThing();
```

### 8.4.4 Handling variant data: VLoanedDataSamples and VDataSample

One of the main purposes of ThingEx is to let you read and write protobuf data, but also continue to be to read and write legacy data encoded in the IOT_NVP_SEQ datatype. It is perfectly reasonable to have a `ThingClass` input that return both IOT_NVP_SEQ data and protobuf data. To do this, new read and read_next methods were required; once that did not return `std::vector<IOT_NVP_SEQ>`.

ThingEx includes read and read_next methods, but they return a new data type: `VLoanedDataSamples`. `VLoanedDataSamples` is a C++ collection-like class (it is iterable) that can contain both IOT_NVP_SEQ data and protobuf data. `VLoanedDataSamples` also manages the memory associated with the data read, ensuring that is automatically released once you are done with it. (This is why the name includes ‘loaned’). You can treat a `VLoanedDataSamples` collection like most C++ collection types. You can iterate over it:

```cpp
VLoanedDataSamples samples = ...;
for(auto sample : samples) {
  // do something with each sample
}
```

You can also use methods like `begin()` and `end()` to get C++ random-access iterators:
Elements of a VLoanedDataSamples collection are of type VDataSample. They are similar to the original DataSample, but their contents is variable.

The method getWireFormat() returns one of WireFormat::NVP_SEQ or WireFormat::PROTOBUF, telling you what kind of data is contained in the sample. If you had a ThingClass input which returned both NVP_SEQ and PROTOBUF data (as the TypeEvolution, Gen2_Sensor_Dashboard TemperatureDashboard.cpp application does, you could process received samples as follows:

```cpp
VLoanedDataSamples samples = ...;
VLoanedDataSamples::const_iterator it = samples.begin();
while (it < samples.end()) {
    // so something with *it
    // OR
    // use it->XXX to access methods of the current sample
    ++it;
}
```

Even more useful is the method isCompatible, which lets you test whether the data in a VDataSample and be assigned to a particular variable. Again, inspired by Gen2_Sensor_Dashboard, TemperatureDashboard.cpp:

```cpp
Temperature tempData; // a protoc-generated class
const IOT_NVP_SEQ& nvpSeqSample;
if (sample.isCompatible(tempData)) {
    // process protobuf Temperature data
} else if (sample.isCompatible(nvpSeqSample)) {
    // process IOT_NVP_SEQ data
} else {
    // Well, this should never happen,
    // unless a future version introduces a new WireFormat
}
```

Once you know a variable is compatible with the data in a VDataSample, you can use one of the get() method overloads to access it. There are two overloads of get(). The first sets a non-const variable passed to it. This format is appropriate for protoc-generated types, as there is implicitly data movement involved as the binary protobuf data is deserialized into the a protoc-generated class. For example, you would read the Temperature data as follows:

```cpp
Temperature tempData // a protoc-generated class
if (sample.isCompatible(tempData)) {
    sample.get(tempData);
    // use tempData accessor methods:
    tempData.temperature();
}
```

The second form of get() returns a const reference to the data contained in the VDataSample. This is most suitable of WireFormat::NVP_SEQ data, where no transformation is required, and for the sake of efficiency, copying should be avoided. This is how you would unpack a received IOT_NVP_SEQ:

```cpp
Temperature tempData // a protoc-generated class
if (sample.isCompatible(tempData)) {
    sample.get(tempData);
} else {
    // Well, this should never happen,
    // unless a future version introduces a new WireFormat
}
```


```cpp
const IOT_NVP_SEQ& nvpSeqSample;
if(sample.isCompatible(nvpSeqSample)) {
    nvpSeqSample = get<IOT_NVP_SEQ>();
}
```

If you care about avoiding unnecessary data copies, be careful about the type declaration of variables like `nvpSeqSample`. The following, very similar code, results in a copy:

```cpp
IOT_NVP_SEQ nvpSeqSample;
if(sample.isCompatible(nvpSeqSample)) {
    nvpSeqSample = get<IOT_NVP_SEQ>(); // copy
}
```

Finally, you may use the second overload of `get()` for protobuf data, if you do not have the protoc-generated C++ class available. This might be the case with a dynamic `ThingClass` input. In that case, you could use the `::com::adlinktech::datariver::GPB_PAYLOAD_VEC` data type. (This is a `std::vector<std::uint8_t>`, representing an undecoded protobuf.) Your code might look like this:

```cpp
const GPB_PAYLOAD_VEC& undecodedProtobuf;
if(sample.isCompatible(undecodedProtobuf)) {
    undecodedProtobuf = get<GPB_PAYLOAD_VEC>();
    // See later in the documentation for how
    // to use undecodedProtobuf.
}
```

### 8.4.5 ThingEx read(), read_next() and DataAvailableListenerEx

With an understanding of `VLoanedDataSamples` and `VDataSample`, you are ready to use the ThingEx versions of `read()` and `read_next()`. These methods return `VLoanedDataSamples`, and you would use the techniques, above, to navigate them.

Similarly, you can create a `DataAvailableListenerEx` subclass, whose `notifyDataAvailable()` method receives a `VLoanedDataSamples` object. You would add the listener to a ThingEx object using the familiar `addListener()` methods, and remove it using `removeListener()`.

### 8.4.6 Writing Protobuf data

When using Google Protobuf generated C++ classes, you must use the ThingEx::write. The generated .dr.h file takes care of the rest. Your code might look like this:

```cpp
ThingEx m_thing = ...; // probably assigned somewhere else
Temperature sensorData; // a generated protobuf C++ message class
sensorData.set_temperature(32);
m_thing.write("temperature", sensorData);
```
8.4.7 Advanced: Dealing with unknown protobuf types

The EdgeSDK is powerful. One capability is to defined *ThingClass* inputs that are dynamic. That is, they match *TagGroup* definitions that are not local to your application. If you have such a dynamic input, and it matches protobuf data, then you need to know how to unpack that data. Most of this involves using standard Google protobuf APIs. However, the com::adlinktech::datariver::TagGroup class provides some methods that help with this, too.

Recall that earlier, we had the following code snippet:

```cpp
const VDataSample& sample = ...;
const GPB_PAYLOAD_VEC& undecodedProtobuf;
if(sample.isCompatible(undecodedProtobuf)) {
    undecodedProtobuf = get<GPB_PAYLOAD_VEC>();
    // See later in the documentation for how
    // to use undecodedProtobuf.
}
```

The problem is, how do we decode this serialized protobuf? We start by asking the TagGroup associated with the sample, by calling *getTagGroup()* and then query *TagGroup::protobufBuildInto* to construct a google::protobuf::DescriptorPool:

```cpp
const VDataSample& sample = ...;
const GPB_PAYLOAD_VEC& undecodedProtobuf;
if(sample.isCompatible(undecodedProtobuf)) {
    undecodedProtobuf = get<GPB_PAYLOAD_VEC>();
    // create a DescriptorPool with the TagGroup's
    // descriptors...
    using namespace ::google::protobuf;
    DescriptorPool p;
    const Descriptor* d = sample.getTagGroup().protobufBuildInto(&p);
    // TODO: find and use the descriptor load the data
}
```

This creates a stack-based DescriptorPool into which we load the descriptor for the data. In a real world application, you might want the DescriptorPool to have a longer life time in order to avoid repeatedly loading the same descriptors. Be aware that you cannot count on using a single DescriptorPool; different *TagGroups* may have incompatible descriptors. In general, each Descriptor should be loaded into a separate DescriptorPool. The method *TagGroup::getProtobufDescriptorHash()* returns a string representation of a 128-bit hash of the TagGroup descriptor. It is safe to assume that this value can uniquely identify a DescriptorPool, and could thus serve as a key into a string-to-DescriptorPool map.

Apart for optimization DescriptorPool creation, the last step is to deserialize the protobuf, and examine its values. The method *TagGroup::getProtobufFullTypeName()* lets you find the *TagGroup* Descriptor in cached descriptor pool. Regardless of how you find the descriptor, your code would then proceed to load data from the received buffer:

```cpp
const GPB_PAYLOAD_VEC& undecodedProtobuf = ...;
// obtained from protobufBuildInto or by look-up in a pool
const Descriptor* d = ...;
// allocated locally, or stored with the descriptor pool
DynamicMessageFactory* factory = ...;
// Use the factory to retrieve a 'prototype' of the Message class
const Message *prototype = factory->GetPrototype(d);
// allocate a Message we can actually manipulate
Message *m = prototype->New();
// load the undecoded protobuf into 'm'
m->ParseFromArray(undecodedProtobuf.data(), undecodedProtobuf.size());
// TODO: use google::protobuf::Reflection to examine 'm'
```

(continues on next page)
This code is complex. You write simple, but inefficient code without resorting to memory allocation. However, in a real world situation, you will likely want to cache DescriptorPool, DynamicMessageFactory instances. You need to be understand the Google APIs, and, in particular, the life-time requirements for types like DescriptorPool and DynamicMessageFactory. (DynamicMessageFactory holds pointers to data owned by DescriptorPool; the factory must be destroyed before the DescriptorPool.) Exercise caution as you start this adventure.

One last thing. Type evolution means that a given TagGroup id may have more than one descriptor. The above code randomly selects one of those descriptors. You can access all of the evolutions of a particular tag group by calling DataRiver::getDiscoveredTagGroupRegistry() and then DiscoveredTagGroupRegistry::findTagGroups. (Note the final ‘s.’) This later method will return all the available TagGroup definitions. Each will have a distinct value for getProtobufDescriptorHash() and, of course, unique descriptors.

You can read more about type evolution, below.

8.5 Type Evolution

Over time, it may become necessary to evolve the TagGroups for applications(s).

A TypeEvolution example is provided in EdgeSDK_HOME/examples/cpp, where the same application is evolved over 3 generations, with different TagGroup formats.

- The first generation uses TagGroups defined with JSON format.
- The second generation changes a TemperatureSensor to write a protobuf TagGroup. The Dashboard reads samples from generation 1 and generation 2.
- The third generation of the application uses google protobuf type evolution to add a new field to the protobuf temperature TagGroup. The Dashboard can read data from all three versions of a sensor.
- Included in the example is a run.cmake file that launches various generations of sensors and dashboards, and change logs for each application generation.
- A README file provides additional instructions.

8.5.1 How a generation of applications evolve from JSON/NVP_SEQ to Protobuf

- Create a new tag group, defined in a .proto file. It must have a different ‘id’ from the original.
- Use methods for distinguishing NVP_SEQ samples from Protobuf samples.

```cpp
VLoanedDataSamples msgs = m_thing.read("temperature", 1000);

// Process samples
for (const VDataSample& msg : msgs) {
    auto flowState = msg.getFlowState();
    if (flowState == FlowState::ALIVE) {
        // translate everything into a Temperature object.
        Temperature tempData;
        std::string sensorGeneration;
        if (msg.isCompatible(tempData)) {

(continues on next page)```
sensorGeneration = "Gen2";
msg.get(tempData);
else {
    sensorGeneration = "Gen1";
    // unpack the temperature from an IOT_NVP_SEQ
    const IOT_NVP_SEQ& dataSample = msg.get<IOT_NVP_SEQ>();
    try {
        // Get temperature value from sample
        for (const IOT_NVP& nvp : dataSample) {
            if (nvp.name() == "temperature") {
                tempData.set_temperature(nvp.value().iotv_float32());
            }
        }
    }
    catch (exceptions e) {
        cerr << "An unexpected error occurred: " << e.what() << endl;
        continue;
    }
}
// Show output
cout << "Temperature data received for flow "
    << msg.getFlowId() << "(" << sensorGeneration << ")": "
    << fixed << setw(3) << setprecision(1)
    << tempData.temperature() << endl;
}

8.5.2 How a generation of applications evolve a Protobuf TagGroup

- Google provides rules for updating a message type. Please refer to:
  https://developers.google.com/protocol-buffers/docs/proto3#updating
- TagGroup “id” DOES NOT CHANGE, if you have followed Google’s rules.
- When new applications receive messages from old applications, default values are provided for any fields that
  have been added. Fields that have been removed will not be received.
- When old applications receive messages from new applications, default values will be provided for any fields
  that have been removed. Fields that have been added will not be received.

8.6 Real-world scenario 2: Connect a dashboard (GPB)

This scenario is the same as the Real-world scenario 2, except that the TagGroup is defined using google protocol
buffers (GBP).

In the Real-world scenario 2 example, the application code uses a read selector to select the desired subset of data.
8.6.1 Temperature TagGroup .proto

The TagGroup for Temperature is specified using a .proto file.

Example: TemperatureTagGroup.proto file

```protobuf
syntax = "proto3";
import "adlinktech/datariver/descriptor.proto";
package com.adlinktech.example.protobuf;

message Temperature {
  option(.adlinktech.datariver.tag_group) = {
    qosProfile: "telemetry"
    description: "ADLINK Edge SDK Example Temperature TagGroup"
  };
  float temperature = 1 [(_.adlinktech.datariver.field_options) = {
    description: "Temperature"
    unit: "°C"
  }];
}
```

8.6.2 ThingClass Definitions

The ThingClass definitions have updated references to the new protobuf TagGroup.

The ThingClass for the dashboard defines the ‘Temperature’ TagGroup as input:

```json
{
  "name": "TemperatureDashboard",
  "context": "com.adlinktech.example.protobuf",
  "version": "v1.0",
  "description": "dashboard that shows temperature sensor data",
  "inputs": [{
    "name": "temperature",
    "tagGroupId": "Temperature:com.adlinktech.example.protobuf:v1.0"
  }]
}
```

The ThingClass for the sensor defines the ‘Temperature’ TagGroup as input:

```json
{
  "name": "TemperatureSensor",
  "context": "com.adlinktech.example.protobuf",
  "version": "v1.0",
  "description": "temperature sensor",
  "outputs": [{
    "name": "temperature",
    "tagGroupId": "Temperature:com.adlinktech.example.protobuf:v1.0"
  }]
}
```

Note: The ThingClass “context” has changed, requiring updates to the “classId” of the dashboard and sensor config-

8.6. Real-world scenario 2: Connect a dashboard (GPB)
uration properties files. (TemperatureDashboardProperties.json and TemperatureSensor*Properties.json)

## 8.6.3 Include

The protocol buffer compiler will generate the data access classes for the TagGroup .proto files. The access classes provide accessors for each field and can be used in the application code to populate, serialize and retrieve protocol buffer messages.

The generated DataRiver TagGroup header files must be included in the application code in order to reference the required TagGroup data types.

```cpp
#include "definitions/TemperatureTagGroup.dr.h"

// Note: Include statements for <IoTDataThing.hpp>
// and <thing_IoTData.h> no longer required
```

## 8.6.4 Register TagGroup

In the generated TemperatureTagGroup.dr.h header file, a Helper class provides api to register the TagGroup on the DataRiver.

```cpp
// Register TagGroup using .proto generated types.
::com::adlinktech::example::protobuf::TemperatureHelper::registerWithDataRiver(m_dataRiver);
```

## 8.6.5 ThingEx

When creating a Thing on the DataRiver where a TagGroup is defined using GBP, a ThingEx class instance is returned. (As opposed to a Thing class instance.)

```cpp
// Create a Thing based on properties specified in a JSON resource file.
JSONThingProperties tp;
tp.readPropertiesFromURI(m_thingPropertiesUri);
ThingEx m_thing = m_dataRiver.createThing(tp);
```

## 8.6.6 Write

The temperature sensor performs the write call on the ThingEx instance, passing in a Temperature data type as an argument.

```cpp
void writeSample(const float temperature) {
    ::com::adlinktech::example::protobuf::Temperature sensorData;
sensorData.set_temperature(temperature);
m_thing.write("temperature", sensorData);
}
```
8.6.7 Read

The temperature dashboard performs the read call on the ThingEx instance. The application code for the GPB TagGroup differs from the code for the JSON format:

- usage of generated `Temperature` data type
- usage of `ThingEx` (instead of Thing)
- usage of `SelectorEx` (instead of Selector)
- usage of `VLoadedDataSamples`
- usage of `VDataSample` (instead of DataSample)

```cpp
using ::com::adlinktech::example::protobuf::Temperature;

string selection = floor + ".*";
ThingEx::SelectorEx selector = m_thing.select("temperature").flow(selection);

// Read data using selector
VLoadedDataSamples msgs = selector.read(1000);

// Process samples
for (const VDataSample& msg : msgs) {
    auto flowState = msg.getFlowState();
    if (flowState == FlowState::ALIVE) {
        Temperature dataSample;
        msg.get(dataSample);
        float temperature = dataSample.temperature();
    }
}
```
By default, data on the Data River is not encrypted and applications that create Things and connect to the Data River do not have to provide authentication. This means that any application can connect to a Data River instance, create Things and receive data for any TagGroup via the Thing inputs. In addition, all data (including discovery data) on the Data River can be read on network level (e.g. using a network sniffing tool). Depending on the requirements for an IoT system, securing the Data River may be required. The Edge SDK provides the means to protect data on the Data River and to restrict which application (thus what Things) have access to the river (i.e. are allowed to create a Data River instance object to connect). Optionally, to provide or restrict access to TagGroups, access control rules can be specified.

9.1 Security model

Securing the Data River means providing:

- **Authentication of applications** - Verify the identity of applications that connect to the Data River. Authentication uses PKI (Public Key Infrastructure) with a pre-configured shared Certificate Authority (see Public Key Infrastructure).

- **Access Control** - Enforce policy decisions on what data (TagGroups and discovery data) an authenticated application (and the Things it creates) can read from or write to the Data River.

- **Encryption** - Provides confidentiality of data on the Data River, by encoding data samples (of a certain TagGroup) or meta-data (used for discovery) in such a way that only authorized applications can decode it and read the contents.

- **Integrity** - The assurance of the accuracy and consistency of data samples on the Data River, including verification of the identity of a sender.
9.2 Security configuration

Configuring security is done by providing specific settings to Data River applications. This is transparent to applications (and their Things) that connect to the river: the application is not aware of any security related aspects and there is no need to change a Data River application when enabling security for the Data River. The Edge SDK includes a tool called securitycomposer that generates the required configuration set for each application in a system that is using a secured Data River.

The next section describes the configuration files and their options that can be used with the securitycomposer tool.

For details on running the securitycomposer command line interface tool, see Using the Configuration tool.

9.2.1 Configuration options

The securitycomposer tool transforms input configuration (in Json format) to the required configuration files for a specific application (actually for a specific Data River instance object that is created by this application).

Assuming that all applications have their own identity, each application that connects to this secure Data River needs its own specific configuration. The securitycomposer tool is run for each participating application, and provides application-specific settings in each run. Each application configuration is written to an output directory, including the certificates that are required during runtime.

The set of input files for the tool consists of:

- A Json file that contains generic security settings for the Data River instance that will be secured. See Data River security settings.
- The certificate of the certificate authority (CA) for identity management. The CA is not used by the security composer tool, but is required during runtime by the application that is connecting to the secured Data River. See Public Key Infrastructure.
• The certificate and private key of the CA for permissions management: the private key of this CA is used by the security composer tool to sign the generated configuration files. During runtime the application needs only the certificate of this CA, for validating the signature of the permissions files. The private key of this CA should never be available in a runtime system, see also the remark in the next section.

• A Json file with security settings that are specific for an application that connects to a secured Data River. See Application security settings.

• The certificate and the private key for the identity of the application, that will be used during runtime for identification of the application. For more information on obtaining this identity certificate and key, see Public Key Infrastructure.

Fig. 2: Security Composer tool

9.2.1.1 Data River security settings

The security related settings that are global for a specific Data River are stored in a Json file that will be reused for generating the configuration files for all applications that connect to this Data River. In the example ‘SecureDataRiver’ that is bundled with the Edge SDK, an example of this file can be found (the file /examples/SecureDataRiver/config/src/data_river_config.json). This Json file contains 3 sections:

• certificationAuthority: contains the URI of the identity CA certificate and the URI of the permissions CA certificate and private key (all in PEM file format). The private key of the permissions CA is used to sign the configuration files to prevent unauthorized modifications in this configuration. The certificates for both CAs are copied to the output directory, because during runtime the Data River security implementation will use these to verify other applications’ identity and permissions.

Note: It is important to store the private key of the CA’s in a secure location, because anyone with access to these files is able to create new identities (that can be used to connect to a secured Data River) and create their own signed permissions file which can result in unauthorized access to data in the system.

• discoveryProtection: this node contains a single setting (enabled, boolean) that enables or disables the protection of the meta-data that is exchanged over the Data River. This meta-data enables applications to discover other Things connected to the Data River as well as the TagGroups they produce and consume. When set to true,
meta-data will be encrypted and only authenticated applications can decrypt and read the data. When enabled, it is also not possible for other Data River based applications (or any other network tooling) to inject (malicious) meta-data that will be processed by the secured Data River participants.

- **dataProtection**: contains rules for protecting TagGroup data. This section contains a default rule, that is used when no other (more specific) rules apply to a TagGroup. Each rule consists of a TagGroupId (except for the default rule) and a policy element. The TagGroupId is in the form `name:context:version` and specifies the TagGroup the rule applies to, which may contain wildcard characters * (zero or more characters) and ? (any single character). The policy part of a rule specifies if access control is enabled for reading and/or writing data for this TagGroup or set of TagGroups. These access control settings do not specify which applications can read or write the data, these rules just indicate that reading or writing for this TagGroup(s) is restricted (or not, if disabled) to applications that have explicit permission (see Application security settings). The policy element also includes a protection setting to specify the kind of protection that will be applied to the data. This can be one of:

  - **sign** - The data samples for the TagGroup(s) are signed. This enables the Thing that receives data to verify the identity of the sender. The data is not encrypted, so it can be read at network level, e.g. using a network sniffing tool.

  - **encrypt** - Data samples for the TagGroup(s) are both signed and encrypted. The sender of the data can be verified and the data cannot be read by unauthorized nodes (a network sniffer or an unauthorized Edge SDK based application).

  - **none** - The data is not signed or encrypted, which means that the data for this TagGroup is not protected.

Enabling discovery or data protection impacts the performance of the system. Different factors will affect the actual performance hit when using discovery protection or data protection: the hardware (CPU) an application is running on, the characteristics of the system (number of Things, sample frequency and sample size) and the configured kind of protection (encrypting data requires more CPU processing that only signing data). Protecting discovery generally has lower impact on the performance of a system than protection TagGroup data, because the amount and frequency of discovery data is typically lower for discovery data.

### 9.2.1.2 Application security settings

Some of the security settings are specific to an application (and the Data River instance object it creates). These settings include the identity of the participant and the permissions on TagGroups, in case there is a restriction on access to a TagGroup configured in the global Data River security settings. In the bundled ‘SecureDataRiver’ example, two examples of this configuration file can be found for both the sensor and display application.

The Json file with application specific settings contains 3 sections:

- **authentication**: This node contains the URIs for the identity certificate and the associated private key (both in PEM file format).

- **permissions**: A list of rules that apply to a (set of) TagGroup(s). Each entry in the list specifies the TagGroup expression (for the syntax see dataProtection setting in the previous section) and the read and/or write policy (‘allow’ or ‘deny’) for this application. These rules are only applicable in case the TagGroup has restricted read and/or write access configured at Data River level. The permissions are checked by the Data River when an application reads or writes data of a TagGroup.

- **defaultPermission**: The default rule that is used when none of the other rules apply to the TagGroup and operation (read or write) being executed.
9.2.2 Using the Configuration tool

The tool for generating security configuration is shipped with the Edge SDK and is available at the following location:

```bash
$EDGE_SDK_HOME/tools/securitycomposer[/exe]
```

This tool has a command-line interface and the following command line options are supported:

--datariver-config=<file>

The path of a file that contains generic security settings for the Data River instance. This is a required parameter.

--app-config=<file>

The path of a file that contains the application specific configuration file. This is a required parameter.

--app-name=<name>

Name of the application for which the security configuration is created. The application name will be used as part of the file name for the generated configuration files. This is a required parameter.

--output-path=<path>

Directory where the generated configuration will be saved. This is a required parameter. If the provided path is not an existing directory, the securitycomposer tool will create the required directories.

The generated configuration files will be written to the specified output directory. Some of the output files are shared by all participating applications, and some files are application specific and therefore prefixed with the supplied application name:

- **Shared configuration output:**
  - governance.p7s
  - identity_ca_cert.pem
  - permissions_ca_cert.pem

- **Application specific output:**
  - `<app-name>_datariver_config.xml`
  - `<app-name>_identity_cert.pem`
  - `<app-name>_identity_priv_key.pem`
  - `<app-name>_permissions.p7s`

The main configuration file for a Data River enabled application is the `<app-name>_datariver_config.json` file. When creating the Data River instance object, the application needs to provide an URI to this file:

```cpp
DataRiver river = DataRiver::getInstance(  
   "file:///home/myapp/config/myapp_datariver_config.xml"
);  
```

The other configuration files are referenced by this Data River configuration XML and should be placed in the same directory (because all references are relative paths).
9.3 Public Key Infrastructure

The system that provides public-key encryption and digital signature services is known as a public-key infrastructure (PKI). The purpose of a PKI is to manage keys and certificates. By managing keys and certificates through a PKI, an organization establishes and maintains a trustworthy networking environment. Some concepts from a PKI that are relevant for the Secure Data River implementation are explained next.

9.3.1 Key concepts

**Public Key Cryptography**: Each application has a key pair, generated during the initial certificate deployment process. The key pair is comprised of a public key, which is shared, and a private key, which is not shared. Data is encrypted with the application's public key and decrypted with their private key. Digital signatures, used for non-repudiation, authentication and data integrity, are also generated using public key cryptography.

**Identity Certificate** is an electronic document used to prove the ownership of a public key. The certificate includes information about the key, information about the identity of its owner (called the subject), and the digital signature of an entity that has verified the certificate's contents (called the issuer). If the signature is valid, and the software examining the certificate trusts the issuer, then it can use that key to communicate securely with the certificate's subject.

**Certificate Authority** issues certificates and acts as the chief agent of trust. When issuing a certificate, the CA signs the certificate with its private key in order to validate it. During electronic transactions the CA also confirms that certificates are still valid. Certificates may be revoked for various reasons. This process is usually accomplished through the use of a Certificate Revocation List (CRL) which is a list of the certificates that have been revoked.

9.3.2 Obtaining CAs and application identity

The security composer tool requires certificates for the identity CA and the permissions CA (including its private key) as part of its input for generating the secure Data River configuration. These CAs can be obtained in several ways, e.g. by generating self-signed certificates using OpenSSL or by using a third-party PKI solution.

The identity CA can issue application identities (certificate and private key pairs). The security composer tool does not generate the application certificate and key. This needs to be done before running the tool, e.g. by using OpenSSL tools or a third-party PKI suite. The resulting certificate and private key for the application identity are supplied to the security composer as input.

The permissions CA is used to protect the security configuration files from unauthorized alterations. The security composer tool uses the private key of the permissions CA to add a digital signature to these configuration files, which includes a hash of the configuration XML. The Data River will validate this signature using the permissions CA public key (that is part of the CA's certificate).
As you develop your Data River applications, you may need to understand the activity on your Data River. The ADLINK Edge SDK includes tools that enable you to monitor Data River activity:

- The Log Viewer tool enables you to view log messages from any ADLINK Edge SDK 1.1 or later application attached to your Data River.
- The Thing Browser tool enables you to collect information on all Things that are active on your Data River.
- The Edge SDK Diagnostic Tool enables the detection of common problems and issues found when running EdgeSDK based applications, and offers troubleshooting tips and solutions to errors.

10.1 Log Viewer

The Log Viewer captures all log messages produced by all Things connected to your Data River.

10.1.1 Starting the Log Viewer

To start the Log Viewer, execute the following command from a command shell:

```
$EDGE_SDK_HOME/tools/logviewer
```

Optionally, include a command line argument indicating the running time of the Log Viewer (in seconds). If no running time is specified on the command line, then the Log Viewer runs indefinitely.

10.1.2 Specifying a Data River configuration

The Log Viewer connects to the Data River with the following environment variables:

- If `ADLINK_DATARIVER_URI` is defined, then it is passed to `DataRiver::getInstance()` as the Data River configuration file.
- Otherwise, the mandatory environment variable `EDGE_SDK_HOME` will be used to select the default Data River configuration.
10.1.3 Log Viewer Output

The following is an example of the log viewer output while running the S1_ConnectSensor example:

```
Starting LogViewer
2019-09-23 18:15:42.123 UTC : vagrant logviewer(9784) v1.2.0 : [INFO] Successfully acquired license
2019-09-23 18:15:42.123 UTC : vagrant logviewer(9784) v1.2.0 : [WARNING] Environment variable with name "OSPL_URI" is already set, overriding with new value
2019-09-23 18:15:42.123 UTC : vagrant logviewer(9784) v1.2.0 : [INFO] DataRiver connection created. Config: "file:///home/vagrant/thingsdk-debug/etc/config/default_datariver_config_v1.2.xml"
2019-09-23 18:15:42.125 UTC : vagrant temperaturedisplay(9783) v1.2.0 : [INFO] Successfully acquired license
2019-09-23 18:15:42.125 UTC : vagrant temperaturedisplay(9783) v1.2.0 : [WARNING] Environment variable with name "OSPL_URI" is already set, overriding with new value
2019-09-23 18:15:42.125 UTC : vagrant temperaturedisplay(9783) v1.2.0 : [INFO] DataRiver connection created. Config: "file:///home/vagrant/thingsdk-debug/etc/config/default_datariver_config_v1.2.xml"
LogViewer stopped
Closing LogViewer
```

Apart from the starting and stopping messages, each line contains a single log entry. Fields in the log entry are separated by the following character sequence: space-colon-space. The fields are:

- a time stamp, in a `yyyy-mm-dd hh:mm:ss.ddd` in the Universal Coordinated Time (UTC) time zone.
- the running application, including host name, application name and process ID.
- the log message, prefixed by its severity enclosed in square brackets.

Looking at the example log entries, we see:

- time stamp entries for July 10, 2019 at approximately 18:15:42 UTC.
- two processes `logviewer` and `temperaturedisplay` with PIDs 9784 and 9783, respectively.
- both processes are running on a common host named `vagrant`.
- each process has produced three log messages: two `INFO` messages and one `WARNING` message.

10.1.4 Effect of Log Viewer on Data River performance

The presence of a Log Viewer on your Data River can cause extra network traffic. If your DataRiver has many active Things that are producing log messages, this extra traffic may be significant. Instead of having the Log Viewer running constantly, use the Log Viewer for short periods of time.

10.2 Thing Browser

The Thing Browser captures details of each Thing connected to your Data River.
10.2.1 Starting the Thing Browser

To start the Thing Browser, execute the following command from a command shell:

```
$EDGE_SDK_HOME/tools/thingbrowser
```

Optionally, you may include a command line argument, indicating the running time of the Thing Browser, in seconds. If no running time is specified on the command line, then the Log Viewer will run indefinitely.

10.2.2 Specifying a Data River configuration

The Thing Browser connects to the Data River with the following environment variables:

- If `ADLINK_DATARIVER_URI` is defined, then it is passed to `DataRiver::getInstance()` as the Data River configuration file.
- Otherwise, to select the default Data River configuration, it uses the mandatory `EDGE_SDK_HOME` environment variable.

10.2.3 Thing Browser Output

The following is an example of the Thing Browser output while running the S1_ConnectSensor example:

```
Starting ThingBrowser

example1.display [Thing]
  Owned by:  vagrant temperaturedisplay(12739) v1.1.0
  Thing ID:  ada27527-4345-4242-890f-215ade35500c
  Context:  example1.display
  Description: Edge SDK example 1 display that shows temperature data
  TemperatureDisplay:com.adlinktech.example:v1.0 [ThingClass]
    Description: ADLINK Edge SDK Example Display that shows temperature sensor inputs:
      temperature: Temperature:com.adlinktech.example:v1.0 [TagGroup]
        Description: ADLINK Edge SDK Example Temperature TagGroup
        QosProfile: telemetry
        Tags:
          temperature : Temperature (kind: FLOAT32 | unit: °C)
    outputs:
      <none>

example1.sensor [Thing]
  Owned by:  vagrant temperaturesensor(12740) v1.1.0
  Thing ID:  79EC6787DA88
  Context:  example1.sensor
  Description: Edge SDK example 1 temperature sensor
  TemperatureSensor:com.adlinktech.example:v1.0 [ThingClass]
    Description: ADLINK Edge SDK Example Temperature sensor inputs:
      <none>
    outputs:
      temperature: Temperature:com.adlinktech.example:v1.0 [TagGroup]
        Description: ADLINK Edge SDK Example Temperature TagGroup
        QosProfile: telemetry
```
(continues on next page)
10.2.4 Effect of Thing Browser on Data River performance

The Thing Browser has little impact on overall Data River performance as it reads data that is already published on the Data River for other purposes.

10.3 Edge SDK Diagnostic Tool

The Diagnostic Tool is a tool packaged with the Edge SDK whose purpose is to automate the detection of common problems and issues found when running Edge SDK based applications, and offer troubleshooting tips and solutions to errors.

The Diagnostic tool makes use of various sub-tools to provide diagnostic information.

The Diagnostic tool, takes the sub-tool’s analysis results, collects them, sorts them, and displays the found issue reports to the user. Issue reports generated by sub-tools generally all contain an issue severity (“Error”, “Warning”, etc), a detailed description of the issue detected, and offer a solution to the user to follow to solve the issue. For example, if an Edge SDK application fails to communicate to other applications on the DataRiver (due to the wrong network interface being selected), a sub-tool that specializes in scanning Edge SDK log files will flag an occurrence of that as warning and construct an issue report to deliver to the Diagnostic Tool for viewing.

The sub-tools that are supported:

- logfinder: searches local file system for opensplice log files, parses them for known patterns and outputs diagnostic results
- logviewer: captures all log messages produced by all Things connected to your DataRiver

10.3.1 Diagnostic Tool Message Types

The Diagnostic Tool uses two message types, defined in Google Protocol Buffer’s .proto format. These are located in tools/diagnostic_tool/proto. In summary toolinfo.proto defines properties that describe some program (a sub-tool) that can be executed on the command line, with some configurable parameters, whose purpose is to analyze some aspect of an Edge SDK domain and report on errors and other issues.

A toolresults.proto holds a list of IssueReports, which a sub-tool is expected to produce as output. See the .proto files for a field by field documentation on what each does. These .proto files are provided in the installation so as to enable the creation of new sub-tools at any time.

10.3.2 Sub-Tools

The Diagnostic Tool gets information on how to run sub-tools based on JSON files located in the tools/diagnostic_tool/tools folder. Each file contains a single JSON encoded object of a ToolInfo message (toolinfo.proto), which the Diagnostic Tool will automatically parse on startup. For every ToolInfo entry found in the tools folder, the Diagnostic Tool can run that tool and collect its output on completion. New tools can be defined and added to this folder at any time.
10.3.3 Using the Diagnostic Tool

The Diagnostic Tool, uses a command line menu interface.

To launch the tool, run “diagnostic_tool” in a terminal where the Edge SDK environment variables have been configured:

```
> diagnostic_tool
```

Use the “help” (-h or –help) command to get command options:

```
-h
OPTIONS
--dump : Run selected tools immediately, and print results to stdout
--tools <toolnames> : Sets the tools selected to be run to <toolnames>,
    where <toolnames> is comma separated list of names as
    they appear in the correspongin toolinfo JSON file.
    ex. --tools logfinder,logviewer
--tool_params <list>: Sets the parameters to be passed to each named tool.
    Format of <list> is:
    <tool_name>:<param_name>=<value>[,<param_name>=<value>...]
    [<tool_name>:<>...]
    ex. --tool_params logfinder:search_dirs=/path/to/file,
        duration=5 logviewer:duration=5
```

Example usage:

- select which tools to run using the “tools” command
- customize the parameters for each tool execution using the “params” command
- run the sub-tools using the “run_diagnostics” command
- list the returned issues using the “list_issues” command

**Note:** In order to run “diagnostic_tool” in a terminal, configure the environment variables by running the Edge SDK config_env_variables script.
This appendix defines the error, warning and information messages the Edge SDK outputs to the log files.

When starting a program, any messages are written to two files in the current working directory. These files may also include messages from the underlying middleware.

- ospl-error.log - contains error messages
- ospl-info.log - contains warning and info messages

A.1 Error Messages

**E_EXCEPTION_IN_DISPATCHER** [Ignoring %s thrown during Dispatcher event processing: %s] A registered listener callback operation threw an uncaught exception while it was being notified by the Dispatcher. Examine your code to make sure your registered callbacks catch the exception before reaching the end of the callback.

**E_MISSING_OUTPUT_TAG_GROUP** [Output tag group "%s" was not found in the system] A call to Thing::write() used in invalid output name. Examine your code and compare it to the ThingClass definition associated with the Thing.

**E_MISSING_INPUT_TAG_GROUP** [Input "%s" declared explicitly but the tag group "%s" was not found in the system] A call to Thing::read() or Thing::read_next() used an invalid input name. Examine your code and compare it to the THingClass definition associated with the Thing.

**E_DUPLICATE_THING_CLASS** [Failed to add thing class: %s. A conflict thing class with the same key detected in the system.] An attempt was made to register a ThingClass on the DataRiver that already existed, and that had a different definition. Examine all applications on your DataRiver that register the ThingClass, and ensure that they share a common definition.

**E_DUPLICATE_TAG_GROUP** [Failed to add tag group: %s. A conflict tag group with the same key detected in the system.] An attempt was made to register a TagGroup on the DataRiver that already existed, and that had a different definition. Examine all applications on your DataRiver that register the TagGroup, and ensure that they share a common definition.

**E_CONNECTION_CLOSED** [DataRiver connection has already been closed.] A DataRiver method was invoked after DataRiver::close() was called. Re-examine your code.

**E_THING_CLASS_NOT_FOUND** [Thing Class "%s" was not found] During a call to DataRiver::createThing(), it was discovered that the ThingClass referred to did not exist in the DataRiver. Ensure that your application is correctly registering the ThingClass before creating a Thing.

**E_DUPLICATE_THING** [Failed to add thing: %s, %s. A conflict thing with the same key detected in the system.] During a call to DataRiver::createThing(), it was discovered that the thing already exists on the DataRiver. Examine all your applications to ensure that each Thing created has a unique identity.
E_INVALID_INPUT_NAME [An invalid inputName, “%s” was passed to Thing::read or Thing::read_next.] A call to Thing::read() or Thing::read_next() contained an invalid ‘inputName’. Consult the Thing class definition, and use one of the input names specified in the definition.

E_WRONG_NUMBER_OF_TAGS [The number of tag values provided does not match the number of tags defined in the output tag group.] The data passed to Thing::write() contained the wrong number of tags. It must include all of the tags defined in the corresponding TagGroup definition.

E_MISSING_TAG_VALUE [Value for tag “%s” is missing] The data passed to Thing::write() was missing a tag of the specified name. Ensure that your code creates a value for each tag defined in the corresponding TagGroup definition.

E_MISSING_SUBTYPE [Subtype “%s” not provided.] The data passed to OutputHandler::validate() contained a subtype that was not referenced in the declared type in the corresponding TagGroup during setup of the OutputHandler.

E_BAD_TAG_VALUE [Value of tag “%s” does not match the type specified in the tag group definition.] The data passed to Thing::write() contained a tag value that did not conform to the declared type in the corresponding TagGroup. Correct your code to use the appropriate value type.

E_EMPTY_FLOW_ID [The value of flowId cannot be empty.] A call to Thing::write() included an empty flowId. A flowId may not be empty.

E_INVALID_OUTPUT_NAME [Output name “%s” was not found.] A call to Thing::write() referenced an output name that was not defined in the corresponding ThingClass definition.

E_MALFORMED_REFERENCE_ID [The reference id provided is not well-formed] A thing class reference was malformed. It should be of the form: name:context:version

E_JSON_PARSE_ERROR [Error parsing JSON document at line %d, column %d: %s] During JSON parsing, an error was discovered. Correct the error.

E_INVALID_TAGGROUPID_REF [tagGroupId has invalid format. “name:context:version” expected.] A tagGroupId has an invalid format. It should be of the form: name:context:version

E_INVALID_CLASSID_REF [thingClassId has invalid format. “name:context:version” expected.] A thing class has an invalid format. It should be of the form: name:context:version

E_INVALID_THINGID [thingId has invalid format. The value should match regular expression %s] A thingId value contains an invalid character

E_INVALID_REF_NAME [name has invalid format. The value should match regular expression %s] An invalid reference format was found. The reference format should be name:context:version.

E_INVALID_REF_CONTEXT [context has invalid format. The value should match regular expression %s] The value of context field contains an invalid character

E_INVALID_REF_VERSION [version has invalid format. The value should match regular expression %s] The value of version field contains an invalid character

E_INVALID_REF_QOS [qos has invalid format. The value should match regular expression %s] The value of qos field contains an invalid character

E_INVALID_JSON_TYPEDEFINITION_REF [Parsing tags “%s” failed. Unable to resolve a typedefinition reference “%s”.%s] During JSON parsing, a typedefinition reference could not be resolved. Examine your JSON.

E_INVALID_JSON_TAGS_REF [Parsing tags “%s” failed. Unable to resolve a tags reference “%s”.%s] During JSON parsing, a tags reference could not be resolved. Examine your JSON.

E_CANNOT_FIND_QOS_PROFILE [Failed to find profile file “%s.xml” in tag group “%s”] A qosProfile reference in a tagGroup definition could not be resolved to a corresponding file located in the qos subdirectory of your Edge SDK installation. Check the value of the EDGE_SDK_HOME environment variable, and the contents of the qos directory.
E_CANNOT_FIND_NEW_QOS_PROFILE  [Failed to find QoS profile file “%s.xml”]  A qosProfile reference in a QoS override definition could not be resolved to a corresponding file located in the qos subdirectory of your Edge SDK installation. Check the value of the EDGE_SDK_HOME environment variable, and the contents of the qos directory.

E_INVALID_JSON_REFERENCE  [Parsing TagGroup “%s” failed. Unable to resolve a %s reference “%s”.%s] During JSON TagGroup parsing, an invalid reference was found. Examine your JSON.

E_EXPECTED_JSON_OBJECT  [Expected a Json Object.] During JSON parsing a JSON object was expected, but some other JSON entity was found instead. Examine your JSON.

E_INVALID_JSON_FIELD_TYPE  [Field “%s” is not of the expected JSON type.] During JSON parsing, the value found in the named field did not conform with its expected data type. Examine your JSON.

E_MISSING_REQUIRED_JSON_FIELD  [Mandatory JSON field “%s” is missing. (Details: “%s”)] During JSON parsing, the named file was found to be missing. Examine your JSON.

E_INVALID_JSON_IOT_TYPE  [The specified type (“%s”) of field “%s” is invalid.] During JSON TagGroup parsing, a tag definition referenced an unrecognized field type. Examine your JSON.

E_BAD_JSON_TAGS_FIELD  [Object must contain only one of “%s” or “%s” fields.%s] During JSON parsing, the referenced JSON object was expected to have exactly one of the named fields. Examine your JSON.

E_UNSUPPORTED_URI_FORMAT  [Unsupported URI protocol “%s”] A URI reference to a TagGroup, ThingClass or Thing properties definition file used an invalid URI protocol. The only protocol supported is file:/. Examine your code.

E_FILE_OPEN_FAILED  [File “%s” could not be opened] An attempt to open the named file failed. Check that the file exists, and that the user running the application has read access to the file.

E_THING_CONFLICT  [Failed to add thing: %s,%s. A conflict thing with the same key detected in the system.] A attempt was made to add a Thing to the DataRiver that was simultaneously registered by another application. Each application should add unique Things to the DataRiver. Examine your applications to ensure that each Thing is created exactly once.

E_THING_CLASS_CONFLICT  [Failed to add thing class: %s. A conflict thing class with the same key detected in the system.] An attempt was made to add a ThingClass to the DataRiver while another application was simultaneously adding a conflicting definition. All applications registering a ThingClass should use identical definitions. Examine the applications involved to ensure they have the correct ThingClass definitions.

E_TAG_GROUP_CONFLICT  [Failed to add tag group: %s. A conflict tag group with the same key was detected in the system.] An attempt was made to add a TagGroup to the DataRiver while another application was simultaneously adding a conflicting definition. All applications registering a TagGroup should use identical definitions. Examine the applications involved to ensure they have the correct TagGroup definitions.

E_CANNOT_OPEN_PROFILE  [Error opening profile “%s” QoS provider “%s”: %s] The named QoS profile file could not be opened. Ensure that the user running your application has read access to the named file.

E_SELECTOR_INVALID_THING_ID  [Selector::thingId failed. An invalid thingId: %s was provided.] An invalid ThingId was passed to Selector::thingId(). Examine and correct your code.

E_SELECTOR_INVALID_FLOW_ID  [Selector::flowId failed. An invalid flowId: %s was provided.] An invalid flowId was passed to Selector::flowId(). Examine and correct your code.

E_DUPLICATED_TAG_NAME  [Multiple tags with same name called “%s” detected in a tag group.] You cannot have two tag groups with same tag name within the same tag group.

E_TAG_GROUP_MISSING_TAG  [No tags found in the tag group/tags definition called “%s”.] At least one tag must be defined in a tag group. Examine and correct your code.

E_INVALID_PATH  [Invalid path “%s”] The provided path could not be converted into an URI
A.1. Error Messages

Edge SDK C++ User Guide, Release 1.4

E_INVALID_INPUT_REF [Input name: “%s” was not found in the class definition: “%s”] The provided input was not found in the input list defined in the ThingClass.

E_INVALID_JSON_RANGE_VALUE [Invalid range value specified: “%s”] An invalid range value was specified for the tag definition.

E_VALUE_OUT_OF_RANGE [The provided value of the Tag: “%s” was not in range specified] A Tag value was not in range specified in the TagGroup tag definition.

C_ADD_THING_CLASS_FAILED [Failed to add thing class: %s. DDS system did not detected the new entity within allocated time limit.] An unexpected error occurred. Gather all available log files and contact ADLINK Technical support.

C_ADD_TAG_GROUP_FAILED [Failed to add tag group: %s. DDS system did not detected the new entity within allocated time limit.] An unexpected error occurred. Gather all available log files and contact ADLINK Technical support.

C_ADD_THING_FAILED [Failed to add thing: %s:%s. DDS system did not detected the new entity within allocated time limit.] An unexpected error occurred. Gather all available log files and contact ADLINK Technical support.

C_READ_ERROR [An unexpected error occurred during Thing::read().] An unexpected error occurred. Gather all available log files and contact ADLINK Technical support.

C_READ_NEXT_ERROR [An unexpected error occurred during Thing::read_next()] An unexpected error occurred. Gather all available log files and contact ADLINK Technical support.

C_WRITE_ERROR [Unexpected DataRiver failure during Thing::write().] An unexpected error occurred. Gather all available log files and contact ADLINK Technical support.

C_WRITE_ERROR_CREATE_OUTPUT_HANDLER [Unexpected DataRiver failure during Thing::write(). Could not create outputHandler for “%s”] An unexpected error occurred. Gather all available log files and contact ADLINK Technical support.

C_DATA_AVAILABLE_LISTENER_ERROR [An unexpected error occurred in DataAvailableListener] An unexpected error occurred. Gather all available log files and contact ADLINK Technical support.

C_UNEXPECTED_ERROR [An unexpected error occurred while connecting to the DataRiver.] An unexpected error occurred. Gather all available log files and contact ADLINK Technical support.

C_LISTENER_THREAD_CREATION_FAILED [Failed to initialize worker thread for listener dispatcher.] An unexpected error occurred during the initialization of dispatcher thread.

C_DATARIVER_CONFIG_PARSE_ERROR [Failed to parse the Data River configuration XML (configuration path: %s, parser error: %s)] An unexpected error occurred when parsing the Data River configuration XML.

C_UNEXPECTED_DDS_ERROR [An error in the underlying DDS system prevented successful execution of the method. DDS exception: “%s”] An unexpected error in the DDS system occurred. Gather all available log files and contact ADLINK Technical support.

F_NO_LICENSE [License for EdgeSDK (feature “%s”) could not be acquired.] An unexpected error occurred. Gather all available log files and contact ADLINK Technical support.
A.2 Warning Messages

W_UNRECOGNIZED_JSON_FIELD [The field name “%s” is not recognized for %s JSON] During JSON parsing, an unrecognized field was found. Examine your JSON and correct it.

W_EMPTY_JSON_FIELD_VALUE [The field “%s” has an empty value in %s JSON] During JSON parsing, an empty value was found in the named field. Examine your JSON.

W_DATARIVER_CONFIG_ALREADY_SET [Environment variable with name “%s” is already set, discarding value from Data River configuration] An variable that was specified in the Data River configuration file is already set in the environment. The value provided in the Data River configuration file will not be used.

W_DATARIVER_CONFIG_OVERWRITTEN [Environment variable with name “%s” is already set, overriding with new value] An variable that was specified in the Data River configuration file is already set in the environment, this value is replaced by the value specified in the Data River configuration because ‘override’ is enabled for this attribute.

W_QOS_OVERRIDE_IGNORED [Runtime QoS override for tag group “%s” is ignored because the original QoS profile specified “%s” does not match with “%s”] A runtime QoS override is ignored if the original tag group specified does not match with tag group’s profile in the system.

W_NO_QOS_PROFILE_TAG_GROUP_IGNORED [A discovered tag group “%s” is ignored because the QoS profile “%s” file does not exist or invalid] A discovered tag group for dynamic input is ignored if the required QoS profile file does not exist or invalid.

A.3 Information Messages

I_NO_ADLINK_LICENSE_VAR [ADLINK_LICENSE environment variable is not set.] During license acquisition, it was discovered that the ADLINK_LICENSE environment variable was not set. ADLINK_LICENSE may be used to point to a license server or a local license file or directory.

I_LIC_NO_ETC [Could not check Edge SDK install subdirectory “etc” for a license because EDGE_SDK_HOME environment variable is not set.] During license acquisition, the software discovered that the EGDE_SDK_HOME environment variable was not set. It is strongly recommended that it be set to the installation directory of EDGE SDK.

I_RLM_INIT_PATH [Call to rlm_init on license path “%s” failed: %s] During license acquisition, the software attempted to access the named directory, but was unable to do so. If you expected the named directory to contain a valid license, examine this directory to confirm its contents are valid.

I_RLM_CHECKOUT [Call to rlm_checkout on license path “%s” failed: %s] During license acquisition, the named directory was checked for a license, but no valid one was found. The software may have continued to look in other locations. If you expected the named directory to contain a valid license, examine this directory to confirm its contents are valid.

I_LICENSE_OK [Successfully acquired license] A license was successfully obtained.

I_INVALID_HANDLE_DURING_READ_NEXT [The read_next() terminated early without going through the entire list due to the invalid instance handle.] During read_next(), the DDS ‘instance handle’ being used to maintain the read_next() position became invalid because the corresponding TagGroup flow instance was purged. The software was forced to abandon the current read_next() pass through the data and start again. No action is required.

I_DATARIVER_CONNECTED [DataRiver connection created. Config: “%s”] The application is successfully connected to the DataRiver.

I_DATARIVER_CLOSED [DataRiver connection closed. Config: “%s”] Connection to the DataRiver is closed.
FEDERATED DEPLOYMENT

The default for the EdgeSDK is to deploy each application in standalone mode (also called single-process) where the lifecycle of the application, its data and the related middleware services are tightly-coupled. It is a simple and easy to use model, but it is not always the most efficient one, as communication between such standalone applications always has to go over DDSI and a related network stack.

Federated deployment means that multiple co-located applications share a single middleware instantiation and with that can benefit from the efficiency, scalability and determinism advantages of communication within a single federation. Federated deployment decouples the lifecycle of application’s data from that of the application itself. Typically, this is exploited by transient and/or persistent data that is managed by one or more durability-services that are deployed as part of federations.

For the EdgeSDK, to enable federated-deployment modify the configuration xml file. (The default file is: default_datariver_config_v1.X.xml.) Edit the following attributes:

- set the variable DDS_SINGLEPROCESS to false
- specifying a sufficiently large DDS_SHMEM_SIZE

Note: Federated deployment requires that the federation is started before applications are started.

Additional documentation on federated deployments can be found in the following links:
https://istkb.adlinktech.com/article/vortex-opensplice-dds-configuration/

B.1 Example: Editing Default Configuration XML File

To enable federated deployment, the following example shows how to modify the default configuration file:

`${EDGE_SDK_HOME}/etc/config/default_datariver_config_v1.2.xml`

Set DDS_SINGLEPROCESS to false

```xml
<!-- Boolean to select the appropriate deployment mode, default is 'true' implying
'standalone' deployment whereas a 'false' value implies federated-deployment
that exploits shared-memory between applications/services on a single machine -->

<setting name="DDS_SINGLEPROCESS" override="true" value="false" />
```
Set DDS_SHMEMSIZE

```xml
  <!-- For federated deployment, this setting specifies the size of the DDS-managed shared-memory segment (e.g. 100M for a 100 Mbyte segment). For standalone mode (DDS_SINGLEPROCESS=true), the default is 0, implying dynamic heap-memory. When the value is >0 in a standalone deployment, a memory-segment of the specified size will be pre-allocated and managed by the DDS middleware (a less common setup that is mainly used for debugging). -->

  <setting name="DDS_SHMEMSIZE" override="true" value="10485760" />
```
C.1 Licensing

Error Message: License for EdgeSDK (feature “OPENSPLICE_THINGSDK”) could not be acquired

Possible Fixes

1. Verify a valid license file (license.lic) has been added to $EDGE_SDK_HOME/etc directory or the ADLINK_LICENSE environment variable is set to the full path and filename of the license file.

2. On Linux, ensure you are following best practices when installing as super-user (root).

   Using EdgeSDK installed by the Linux super user (root)

   **Note:** To install as super user:
   ```
   sudo <path-to-install-program>
   ```

   By default, it installs EdgeSDK under /opt/ADLINK. A non-super-user install is placed in the current user’s home directory: ~/ADLINK.

   To share the EdgeSDK installation amongst multiple users on the same machine, do a super-user install.

   Consuming a super-user install:

   **Best practices:**
   - never consume EdgeSDK as a super-user
   - never attempt to modify the contents of /opt/ADLINK or its subdirectories.

   As a normal user, to set environment variables necessary for compiling and executing EdgeSDK-based applications, run:
   ```
   source /opt/ADLINK/EdgeSDK/1.1.0/config_env_variables.com
   ```

   To modify one of the EdgeSDK examples, copy it to a local directory, then make modifications in your local directory.

   To build one of the EdgeSDK examples, use a cmake ‘out-of-source’ build. Do the following:

   ```
   mkdir ~/s1
   cd ~/s1
   cmake -D CMAKE_BUILD_TYPE=Release /opt/ADLINK/EdgeSDK/1.1.0/examples/cpp/S1_ConnectSensor
   make
   ```
To configure a cmake build against the copy, change the path so that it points to the copied directory.