MD-SAL

Release 8.0.11

OpenDaylight Project

Mar 03, 2022

CONTENTS

CHAPTER

ONE

OVERVIEW

The Model-Driven Service Adaptation Layer (MD-SAL) is message-bus inspired extensible middleware component that provides messaging and data storage functionality based on data and interface models defined by application developers (i.e. user-defined models).

The MD-SAL:

- Defines a **common-layer, concepts, data model building blocks and messaging patterns** and provides infrastructure / framework for applications and inter-application communication.
- Provide common support for user-defined transport and payload formats, including payload serialization and adaptation (e.g. binary, XML or JSON).

The MD-SAL uses **YANG** as the modeling language for both interface and data definitions, and provides a messaging and data-centric runtime for such services based on YANG modeling.

The MD-SAL provides two different API types (flavours):

- **MD-SAL Binding:** MD-SAL APIs which extensively uses APIs and classes generated from YANG models, which provides compile-time safety.
- **MD-SAL DOM:** (Document Object Model) APIs which uses DOM-like representation of data, which makes them more powerful, but provides less compile-time safety.

Note: Model-driven nature of the MD-SAL and **DOM**-based APIs allows for behind-the-scene API and payload type mediation and transformation to facilitate seamless communication between applications - this enables for other components and applications to provide connectors / expose different set of APIs and derive most of its functionality purely from models, which all existing code can benefit from without modification. For example **RESTCONF Connector** is an application built on top of MD-SAL and exposes YANG-modeled application APIs transparently via HTTP and adds support for XML and JSON payload type.

CHAPTER

TWO

BASIC CONCEPTS

Basic concepts are building blocks which are used by applications, and from which MD-SAL uses to define messaging patterns and to provide services and behavior based on developer-supplied YANG models.

Data Tree All state-related data are modeled and represented as data tree, with possibility to address any element / subtree

- **Operational Data Tree** - Reported state of the system, published by the providers using MD-SAL. Represents a feedback loop for applications to observe state of the network / system.
- **Configuration Data Tree** - Intended state of the system or network, populated by consumers, which expresses their intention.

Instance Identifier Unique identifier of node / subtree in data tree, which provides unambiguous information, how to reference and retrieve node / subtree from conceptual data trees.

Notification Asynchronous transient event which may be consumed by subscribers and they may act upon it.

RPC asynchronous request-reply message pair, when request is triggered by consumer, send to the provider, which in future replies with reply message.

Note: In MD-SAL terminology, the term 'RPC' is used to define the input and output for a procedure (function) that is to be provided by a provider, and mediated by the MD-SAL, that means it may not result in remote call.

CHAPTER

THREE

MESSAGING PATTERNS

MD-SAL provides several messaging patterns using broker derived from basic concepts, which are intended to transfer YANG modeled data between applications to provide data-centric integration between applications instead of API-centric integration.

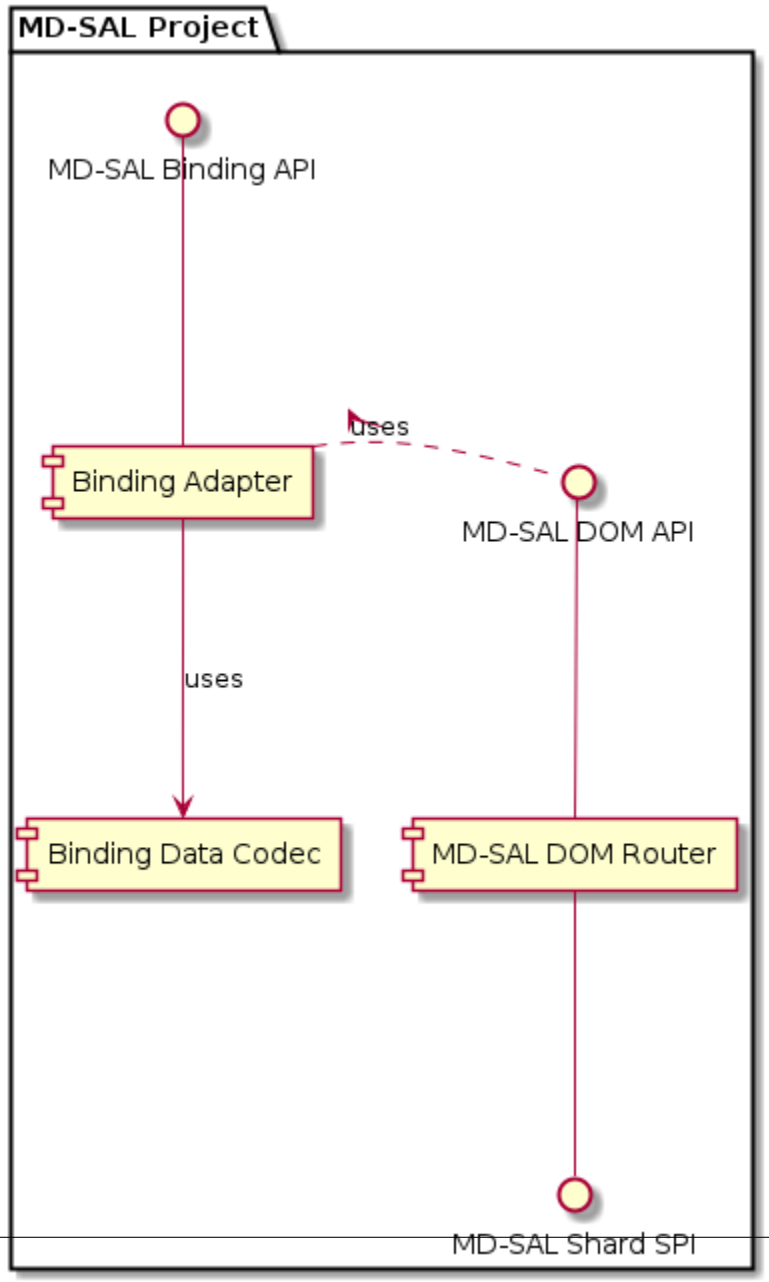
- **Unicast communication**
  - **Remote Procedure Calls** - unicast between consumer and provider, where consumer sends **request** message to provider, which asynchronously responds with **reply** message.
- **Publish / Subscribe**
  - **Notifications** - multicast transient message which is published by provider and is delivered to subscribers.
  - **Data Change Events** - multicast asynchronous event, which is sent by data broker if there is change in conceptual data tree, and is delivered to subscribers.
- **Transactional access to Data Tree**
  - Transactional **reads** from conceptual **data tree** - read-only transactions with isolation from other running transactions.
  - Transactional **modification** to conceptual **data tree** - write transactions with isolation from other running transactions.
  - **Transaction chaining**

CHAPTER

FOUR

TABLE OF CONTENTS

4.1 Architecture



4.2 Conceptual Data Tree

4.2.1 Terminology

Data Tree An instantiated logical tree that represents configuration or operational state data of a modeled problem domain (for example, a controller or a network)

Data Tree Consumer A component acting on data, after this data are introduced into one or more particular subtrees of a Data Tree.

Data Tree Identifier A unique identifier for a particular subtree of a Data Tree. It is composed of the logical data store type and the instance identifier of the subtree's root node. It is represented by a `DOMDataTreeIdentifier`.

Data Tree Producer A component responsible for providing data for one or more particular subtrees of a Data Tree.

Data Tree Shard A component responsible for providing storage or access to a particular subtree of a Data Tree.

Shard Layout A longest-prefix mapping between Data Tree Identifiers and Data Tree Shards responsible for providing access to a data subtree.

4.2.2 Basic Concepts

Data Tree is a Namespace

The concept of a data tree comes from RFC6020. It is is vaguely split into two instances, configuration and operational. The implicit assumption is that **config implies oper**, i.e. any configuration data is also a valid operational data. Further interactions between the two are left undefined and the YANG language is not strictly extensible in the number and semantics of these instances, leaving a lot to implementation details. An outline of data tree use, which is consistent with the current MD-SAL design, is described in draft-kwatsen-netmod-opstate.

The OpenDaylight MD-SAL design makes no inherent assumptions about the relationship between the configuration and operational data tree instances. They are treated as separate entities and they are both fully addressable via the `DOMDataTreeIdentifier` objects. It is up to MD-SAL plugins (e.g. protocol plugins or applications) to maintain this relationship. This reflects the asynchronous nature of applying configuration and also the fact that the intended configuration data may be subject to translation (such as template configuration instantiation).

Both the configuration and operational namespaces (data trees) are instances of the Conceptual Data Tree. Any data item in the conceptual data tree is addressed via a `YangInstanceIdentifier` object, which is a unique, hierarchical, content-based identifier. All applications use the identifier objects to identify data to MD-SAL services, which in turn are expected to perform proper namespace management such that logical operation connectivity is maintained.

Identifiers versus Locators

It is important to note that when we talk about Identifiers and Locators, we **do not** mean URIs and URLs, but rather URNs and URLs as strictly separate entities. MD-SAL plugins do not have access to locators and it is the job of MD-SAL services to provide location independence.

The details of how a particular MD-SAL service achieves location independence is currently left up to the service's implementation, which leads to the problem of having MD-SAL services cooperate, such as storing data in different backends (in-memory, SQL, NoSQL, etc.) and providing unified access to all available data. Note that data availability is subject to capabilities of a particular storage engine and its operational state, which leads to the design decision that a `YangInstanceIdentifier` lookup needs to be performed in two steps:

1. A longest-prefix match is performed to locate the storage backend instance for that identifier
2. Masked path elements are resolved by the storage engine

Data Tree Shard

A process similar to the first step above is performed today by the Distributed Data Store implementation to split data into Shards. The concept of a Shard as currently implemented is limited to specifying namespaces, and it does not allow for pluggable storage engines.

In context of the Conceptual Data Tree, the concept of a Shard is generalized as the shorthand equivalent of a storage backend instance. A Shard can be attached at any (even wildcard) `YangInstanceIdentifier`. This contract is exposed via the `DOMShardedDataTree`, which is an MD-SAL SPI class that implements an `YangInstanceIdentifier -> Shard` registry service. This is an omnipresent MD-SAL service, Shard Registry, whose visibility scope is a single OpenDaylight instance (i.e. a cluster member). **Shard Layout** refers to the mapping information contained in this service.

Federation, Replication and High Availability

Support for various multi-node scenarios is a concern outside of core MD-SAL. If a particular scenario requires the shard layout to be replicated (either fully or partially), it is up to Shard providers to maintain an omnipresent service on each node, which in turn is responsible for dynamically registering `DOMDataTreeShard` instances with the Shard Registry service.

Since the Shard Layout is strictly local to a particular OpenDaylight instance, an OpenDaylight cluster is not strictly consistent in its mapping of `YangInstanceIdentifier` to data. When a query for the entire data tree is executed, the returned result will vary between member instances based on the differences of their Shard Layouts. This allows each node to project its local operational details, as well as the partitioning of the data set being worked on based on workload and node availability.

Partial symmetry of the conceptual data tree can still be maintained to the extent that a particular deployment requires. For example the Shard containing the OpenFlow topology can be configured to be registered on all cluster members, leading to queries into that topology returning consistent results.

4.2.3 Design

Data Tree Listener

A Data Tree Listener is a data consumer, for example a process that wants to act on data after it has been introduced to the Conceptual Data Tree.

A Data Tree Listener implements the DOMDataTreeListener interface and registers itself using DOMDataTreeService.

A Data Tree Listener may register for multiple subtrees. Each time it is invoked it will be provided with the current state of all subtrees that it is registered for.

.DOMDataTreeListener interface signature

```
public interface DOMDataTreeListener extends EventListener {

    void onDataTreeChanged(Collection<DataTreeCandidate> changes, // (1)
        Map<DOMDataTreeIdentifier, NormalizedNode<?, ?>> subtrees);

    void onDataTreeFailed(Collection<DOMDataTreeListeningException> causes); // (2)
}
```

1. Invoked when the data tree to which the Data Tree Listener is subscribed to changed. *changes* contains the collection of changes, *subtrees* contains the current state of all subtrees to which the listener is registered.
2. Invoked when a subtree listening failure occurs. For example, a failure can be triggered when a connection to an external subtree source is broken.

Data Tree Producer

A Data Tree Producer represents source of data in system. Data TreeProducer implementations are not required to implement a specific interface, but use a DOMDataTreeProducer instance to publish data (i.e. to modify the Conceptual Data Tree).

A Data Tree Producer is exclusively bound to one or more subtrees of the Conceptual Data Tree, i.e. binding a Data Tree Producer to a subtree prevents other Data Tree Producers from modifying the subtree.

- A failed Data Tree Producer still holds a claim to the namespace to which it is bound (i.e. the exclusive lock of the subtree) until it is closed.

DOMDataTreeProducer represents a Data Tree Producer context

- allows transactions to be submitted to subtrees specified at creation time
- at any given time there may be a single transaction open.
- once a transaction is submitted, it will proceed to be committed asynchronously.

.DOMDataTreeProducer interface signature

```
public interface DOMDataTreeProducer extends DOMDataTreeProducerFactory, AutoCloseable {
    DOMDataWriteTransaction createTransaction(boolean isolated); // (1)
    DOMDataTreeProducer createProducer(Collection<DOMDataTreeIdentifier> subtrees); //␣
↪(2)
}
```

1. Allocates a new transaction. All previously allocated transactions must have been either submitted or canceled. Setting *isolated* to *true* disables state compression for this transaction.
2. Creates a sub-producer for the provided *subtrees*. The parent producer loses the ability to access the specified paths until the resulting child producer is closed.

Data Tree Shard

- **A Data Tree Shard** is always bound to either the `OPERATIONAL`, or the `CONFIG` space, never to both at the same time.
- **Data Tree Shards** may be nested, the parent shard must be aware of sub-shards and execute every request in context of a self-consistent view of sub-shards liveness. Data requests passing through it must be multiplexed with sub-shard creations/deletions. In other words, if an application creates a transaction rooted at the parent Shard and attempts to access data residing in a sub-shard, the parent Shard implementation must coordinate with the sub-shard implementation to provide the illusion that the data resides in the parent shard. In the case of a transaction running concurrently with sub-shard creation or deletion, these operations need to execute atomically with respect to each other, which is to say that the transactions must completely execute as if the sub-shard creation/deletion occurred before the transaction started or as if the transaction completed before the sub-shard creation/deletion request was executed. This requirement can also be satisfied by the Shard implementation preventing transactions from completing. A Shard implementation may choose to abort any open transactions prior to executing a sub-shard operation.
- **Shard Layout** is local to an OpenDaylight instance.

- **Shard Layout** is modified by agents (registering / unregistering Data Tree Shards) in order to make the Data Tree Shard and the underlaying data available to plugins and applications executing on that particular OpenDaylight instance.

Registering a Shard with the Conceptual Data Tree

Note: Namespace in this context means a Data Tree Identifier prefix.

1. **Claim a namespace** - An agent that is registering a shard must prove that it has sufficient rights to modify the subtree where the shard is going to be attached. A namespace for the shard is claimed by binding a Data Tree Producer instance to same subtree where the shard will be bound. The Data Tree Producer must not have any open child producers, and it should not have any outstanding transactions.
2. **Create a shard instance** - Once a namespace is claimed, the agent creates a shard instance.
3. **Attach shard** - The agent registers the created shard instance and provides in the reigstration the Data Tree Producer instance to verify the namespace claim. The newly created Shard is checked for its ability to cooperate with its parent shard. If the check is successful, the newly created Shard is attached to its parent shard and recorded in the Shard layout.
4. **Remove namespace claim** (optional) - If the Shard is providing storage for applications, the agent should close the Data Tree Producer instance to make the subtree available to applications.

Important: Steps 1, 2 and 3 may fail, and the recovery strategy depends on which step failed and on the failure reason.

4.3 Incremental Backup

4.3.1 Terminology

Source Waits for a Sink to connect. After it does, registers a DTCL and starts sending all changes to the Sink.

Sink Connects to the Source and asks for changes on a particular path in the datastore(root by default). All changes received from the Source are applied to the Sink's datastore.

DTCL Data Tree Change Listener is an object, which is registered on a Node in the datastore andnotified if said node(or any of its children) is modified.

4.3.2 Concept

Incremental Backup vs Daexim

The concept of Incremental Backup originated from Daexim drawbacks. Importing the whole datastore may take a while since it triggers all the DTCLs. Therefore using Daexim as a mechanism for backup is problematic, since the export/import process needs to be executed quite frequently to keep the two sites synchronized.

Incremental Backup simply mirrors the changes made on the primary site to the secondary site one-by-one. All that's needed is to have a Source on the primary site, which sends the changes and a Sink on the secondary site which then applies them. The transport mechanism used is Netty.

Replication (works both for LAN and WAN)

Once the Sink is started it tries to connect to the Source's address and port. Once the connection is established, the Sink sends a request containing a path in the datastore which needs to be replicated. Source receives this request and registers DTCL on said path. Any changes the listener receives are then streamed to the Sink. When Sink receives them he applies them to his datastore.

In case there is a network partition and the connection goes down, the Source unregisters the listener and simply waits for the Sink to reconnect. When the connection goes UP again and the Sink reconnects, the Source registers the DTCL again and continues replicating. Therefore even if there were some changes in the Source's datastore while the connection was down, when the Sink reconnects and Source registers new DTCL, the current initial state will be replicated to the Sink. At this point they are synchronized again and the replication can continue without any issue.

- **Features**
  - **odl-mdsal-replicate-netty**

```
<dependency>
  <groupId>org.opendaylight.mdsal</groupId>
  <artifactId>odl-mdsal-replicate-common</artifactId>
  <classifier>features</classifier>
  <type>xml</type>
</dependency>
<dependency>
  <groupId>org.opendaylight.mdsal</groupId>
  <artifactId>odl-mdsal-replicate-netty</artifactId>
  <classifier>features</classifier>
  <type>xml</type>
</dependency>
```

Configuration and Installation

1. **Install the features on the primary and secondary site**

```
feature:install odl-mdsal-replicate-netty odl-mdsal-replicate-common
```

2. **Enable Source (on the primary site)**

```
config:edit org.opendaylight.mdsal.replicate.netty.source
config:property-set enabled true
config:update
```

All configuration options:

- enabled <true/false>
- listen-port <port> *(9999 is used if not set)*
- keepalive-interval-seconds <amount> *(10 is used if not set)*
- max-missed-keepalives <amount> *(5 is used if not set)*

3. **Enable Sink (on the secondary site)** *In this example the Source is at 172.16.0.2 port 9999*

```
config:edit org.opendaylight.mdsal.replicate.netty.sink
config:property-set enabled true
```

(continues on next page)

(continued from previous page)

```
config:property-set source-host 172.16.0.2
config:update
```

All configuration options:

- enabled <true/false> *(127.0.0.1 is used if not set)*
- source-host <address> *(127.0.0.1 is used if not set)*
- source-port <port> *(9999 is used if not set)*
- reconnect-delay-millis <reconnect-delay> *(3000 is used if not set)*
- keepalive-interval-seconds <amount> *(10 is used if not set)*
- max-missed-keepalives <amount> *(5 is used if not set)*

Switching Primary and Secondary sites

Sites can be switched simply by disabling the configurations and enabling them in the opposite direction.

Example deployment

Running one ODL instance locally and one in Docker

1. **Run local ODL**

```
karaf-distribution/bin/karaf
```

Karaf Terminal - Start features

- features-mdsal - core MDSAL features
- odl-mdsal-replicate-netty - netty replicator
- odl-restconf-nb-bierman02 - we'll be using Postman to access datastore
- odl-netconf-clustered-topolog - we will change data of some netconf devices

```
feature:install features-mdsal odl-mdsal-replicate-netty odl-restconf-nb-
↪bierman02 odl-netconf-clustered-topolog
```

Start Source

```
config:edit org.opendaylight.mdsal.replicate.netty.source
config:property-set enabled true
config:update
```

2. **Run Dockerized Karaf distribution**

To get access to Karaf Terminal running in Docker you can use:

```
docker exec -ti $(docker ps -a -q --filter ancestor=<NAME-OF-THE-DOCKER-
↪IMAGE>) /karaf-distribution/bin/karaf
```

Start features in the Docker's Karaf Terminal

```
feature:install features-mdsal odl-mdsal-replicate-netty odl-restconf-nb-
↪bierman02 odl-netconf-clustered-topolog
```

Start Sink - Let's say the Docker runs at 172.17.0.2 meaning it will find the local Source is at 172.17.0.1

```
config:edit org.opendaylight.mdsal.replicate.netty.sink
config:property-set enabled true
config:property-set source-host 172.17.0.1
config:update
```

3. **Run Postman and try modifying the Source's datastore**

   Put data to the local datastore:

   - Header

   ```
   PUT http://localhost:8181/restconf/config/network-topology:network-
   ↪topology/topology/topology-netconf/node/new-netconf-device
   ```

   - Body

   ```
   <node xmlns="urn:TBD:params:xml:ns:yang:network-topology">
     <node-id>new-netconf-device</node-id>
     <host xmlns="urn:opendaylight:netconf-node-topology">127.0.0.1</host>
     <port xmlns="urn:opendaylight:netconf-node-topology">16777</port>
     <username xmlns="urn:opendaylight:netconf-node-topology">admin</
   ↪username>
     <password xmlns="urn:opendaylight:netconf-node-topology">admin</
   ↪password>
     <tcp-only xmlns="urn:opendaylight:netconf-node-topology">false</tcp-
   ↪only>
     <reconnect-on-changed-schema xmlns="urn:opendaylight:netconf-node-
   ↪topology">false</reconnect-on-changed-schema>
     <connection-timeout-millis xmlns="urn:opendaylight:netconf-node-
   ↪topology">20000</connection-timeout-millis>
     <max-connection-attempts xmlns="urn:opendaylight:netconf-node-topology
   ↪">0</max-connection-attempts>
     <between-attempts-timeout-millis xmlns="urn:opendaylight:netconf-node-
   ↪topology">2000</between-attempts-timeout-millis>
     <sleep-factor xmlns="urn:opendaylight:netconf-node-topology">1.5</
   ↪sleep-factor>
     <keepalive-delay xmlns="urn:opendaylight:netconf-node-topology">120</
   ↪keepalive-delay>
   </node>
   ```

   Get the data locally

   - Header

   ```
   GET http://localhost:8181/restconf/config/network-topology:network-
   ↪topology/
   ```

   Get the data from the Docker. The change should be present there.

   - Header

```
GET http://172.17.0.2:8181/restconf/config/network-topology:network-
↪topology/
```

4.4 MD-SAL Binding Query Language User Guide

4.4.1 Feature Overview

Query language based API for work with YANG based models. MD-SAL component provides a binding query language to interact with the underlying data store. This API provides an easy and type-safe mechanism for retrieving and processing data from generated DOM based on queries. On the DOM layer the expression can be transmitted, and it gives the possibility to move the execution to the storage backend. This can reduce app/backend interchange data. This API is a part of the MD-SAL component and can be found inside the *org.opendaylight.mdsal.binding.api.query* package.

4.4.2 Query structure

- *QueryExpression* - Built sequence-based expression. QueryExpression is similar to an SQL query expression. While SQL operates on tables and rows, QueryExpression operates on a subtree. Creates by *QueryFactory*.
- *QueryExecutor* - Interface to execute query expression and retrieve execution result.
- *QueryResult* - Result execution of *QueryExpression* by *QueryExecutor*.

Query result will contain Objects which can be represented using the next methods:

- *stream* - Returns sequential Stream of values from the query result.
- *parallelStream* - Returns parallel Stream of values from the query result.
- *getValues* - Returns List of generic Objects from the query result.
- *getItems* - Returns List of Items(Object and *InstanceIdentifier*) from query result.

4.4.3 Query Usage

A QueryExpression is built up of three items, which specify *what* to search, looking for *something* matching a *predicate*. This is similar in structure to being an SQL query: FROM *what* SELECT *something* WHERE *predicate*.

Query execution workflow:

Query Factory -> Root Path -> Query Builder -> (Extract child node -> Matcher) -> build

- *Query Factory* - Primary entry point for creating Query.
- *Root Path* - Specify Subtree root path for start from the query. This corresponds to the *what* part of a query. Just as with SQL tables, this path has to point to at most one item.
- *Query Builder* - Intermediate builder stage, which allows providing a specification of the query. On query completed call _build_ method finalize the creation of simple query.
- *Extract child node* - Add a child path component to the query specification of what needs to be extracted. This constitutes an intermediate step of specifying the *something* part of *what* needs to be found.
- *Matcher* - Specify a matching pattern for request using Leaf's getter method and appropriate matcher. This constitutes the *predicate* part of the query. Every candidate has to match pattern for the request.

Child node can be specified in the next ways:

- using child container *class*;
- using an exact match in a keyed list using *List* and *Key* types;
- using child case *class* and child *class*;

Leaf's value can be retrieved passing method reference from container type.

Query engine supports *Empty, string, int8, int16, int32, int64, uint8, uint16, uint32, uint64, Identity, TypeObject* leaf's value types. Appropriate MatchBuilder pattern applied according to leaf's value type.

4.4.4 Examples

Create a simple executor:

```
QueryExecutor executor = SimpleQueryExecutor.builder(CODEC)
    .add(new FooBuilder()
        .setSystem(BindingMap.of(
            new SystemBuilder().setName("SystemOne").setAlarms(BindingMap.of(
                new AlarmsBuilder()
                    .setId(Uint64.ZERO)
                    .setCritical(Empty.getInstance())
                    .setAffectedUsers(BindingMap.of()).build(),
                new AlarmsBuilder()
                    .setId(Uint64.ONE)
                    .setAffectedUsers(BindingMap.of()).build()))
                .build(),
            new SystemBuilder().setName("SystemTwo").setAlarms(BindingMap.of(
                new AlarmsBuilder()
                    .setId(Uint64.ZERO)
                    .setCritical(Empty.getInstance())
                    .setAffectedUsers(BindingMap.of(
                    )).build())).build()))
        .build())
    .build();
```

Create query expression and execute it using executor above:

```
QueryExpression<System> query = new DefaultQueryFactory(CODEC).
↪querySubtree(InstanceIdentifier.create(Foo.class))
    .extractChild(System.class)
    .matching()
    .leaf(System::getName).contains("One")
    .build();
final QueryResult result = executor.executeQuery(query);
List items = result.getItems();
```

This expression will retrieve System node with name containing "One" from DOM tree.

```
QueryExpression<Alarms> query
= new DefaultQueryFactory(CODEC).querySubtree(InstanceIdentifier.create(Foo.class))
    .extractChild(System.class)
    .extractChild(Alarms.class)
    .matching()
    .leaf(Alarms::getId).valueEquals(Uint64.ZERO)
```

(continues on next page)

(continued from previous page)

```
    .build();
final QueryResult result = executor.executeQuery(query);
List items = result.getItems();
```

The result of this query expression will be a list of two items - Alarms with Id of ZERO.

4.5 MD-SAL Binding Query Language Developer Guide

Note: Reading this section is likely useful as it contains an overview of MD-SAL Binding query language in OpenDaylight and a how-to use it for retrieving data from data storage.

4.5.1 Retrieving data from storage

MD-SAL has two ways (operations) of retrieving data from storage: read-like and query-like operations.

Read-like operation

The method **read** of ReadTransaction interface.

```
<T extends DataObject> FluentFuture<Optional<T>> read(LogicalDatastoreType store,␣
↪InstanceIdentifier<T> path);
```

The method reads data from the provided logical data store located at the provided path. If the target is a subtree, then the whole subtree is read (and will be accessible from the returned DataObject). So we are getting DataObject which we need to process in code for getting relevant data:

```
FluentFuture<Optional<Foo>> future;
try (ReadTransaction rtx = getDataBroker().newReadOnlyTransaction()) {
    future = rtx.read(LogicalDatastoreType.CONFIGURATION, InstanceIdentifier.create(Foo.
↪class));
}
Foo haystack = future.get().orElseThrow();
Object result = null;
for (System system : haystack.nonnullSystem().values()) {
    if (needle.equals(system.getAlias())) {
        result = system;
        break;
    }
}
```

Note: The structure of the Foo container is here.

Query-like operation

The method **execute** of QueryReadTransaction interface.

```
<T extends DataObject> FluentFuture<QueryResult<T>> execute(LogicalDatastoreType store,␣
↪QueryExpression<T> query);
```

The method executes a query on the provided logical data store for getting relevant data. So we are getting result which we need for future business logic processing. Before running the method execute we need to prepare a query with the match predicates. For example, we want to find in Foo container the System with alias **target-needle**:

```
String needle = "target-needle";
QueryExpression<System> query = factory.querySubtree(InstanceIdentifier.create(Foo.
↪class))
    .extractChild(System.class)
        .matching()
            .leaf(System::getAlias).valueEquals(needle)
        .build();
```

The method **querySubtree** creates a new **DescendantQueryBuilder** for a specified root path. It's intermediate query builder stage, which allows the specification of the query result type to be built up via **extractChild(Class)** and **extractChild(Class, Class)** methods. They used to specify which object type to select from the root path. Once completed, use either **build()** to create a simple query, or **matching()** to transition to specify predicates. There is a bunch of overloaded methods **leaf** which based on the type of arguments returns specific match builders:

- ValueMatchBuilder methods:

ValueMatch<T> nonNull();
ValueMatch<T> isNull();
ValueMatch<T> valueEquals(V value);

- ComparableMatchBuilder extends ValueMatchBuilder and adds methods:

ValueMatch<T> lessThan(V value);
ValueMatch<T> lessThanOrEqual(V value);
ValueMatch<T> greaterThan(V value);
ValueMatch<T> greaterThanOrEqual(V value);

- StringMatchBuilder extends ValueMatchBuilder and adds methods:

ValueMatch<T> startsWith(String str);
ValueMatch<T> endsWith(String str);
ValueMatch<T> contains(String str);
ValueMatch<T> matchesPattern(Pattern pattern);

After creation of query, we can use it in the method execute of QueryReadTransaction interface:

```
FluentFuture<QueryResult<System>> future;
try (ReadTransaction rtx = getDataBroker().newReadOnlyTransaction()) {
    future = ((QueryReadTransaction) rtx).execute(LogicalDatastoreType.CONFIGURATION,␣
↪query);
}
QueryResult<System> result = future.get();
```