**ODL Basic Developing**

Preparing the environment:

Install devel on CentOS machine

Install karaf with ODL:

* Download stable boron dist
* Configure its feature conf file
* Run bin/start (make sure it works by running “bundle:list” on bin/client).

First day (4\5\17), Hello World:

-Cloning the netvirt/vpnservices/dhcpservice project

-Cleaning it completely, the new project’s structure can be seen in the git repo, initially should contains only POMs in impl and api, and the common directory which is used as a parent.

-Modify POMs to contain the new project’s name instead of netvirt and dhcp, we might want to change the parent of api and impl to ‘binding-parent’ to avoid uneccesary configuration reqs.

-Eliminate uneccesary dependencies, refer to the github repo for exact list.

The Hello World “Project” consists of two main directories, ***impl*** and ***api***.

First we created a YANG module in the ***api***, named ***greeter***, which represents the data structure holding the name we would want to greet.

Each YANG module has the following basic attributes:

* Namespace (usually the fully qualified name of the module)
* Prefix (a shorthand name we could use in the current module)
* Revision (date)
* Container (the core of the module, contains the actual schema)
* (optional) imports of other modules, in which we can also specify a prefix for the import itself, to easily refer to the different imported modules.

A container must hold a *config* boolean (true for configuration related data, false for operational data, mostly provided by southbound), and a description.  
Perhaps a useful explanation I found on *stackoverflow* regarding *config: “*Config is what represents configuration data, usually what will be writable via the northbound agents (CLI, Netconf, Web, etc.), it is also what will be retrieved in a get-config Netconf operation.  
Operational data is status data, data that is not writable via the northbound agents, it will come from a data provider application.

A web client should only be able to do a GET operation on operational data. Because it doesn't make sense to allow a client to change the information about a status.  
For config data it makes sense to have all the operations.”

Following up are data holders. We can either have leaf or a list of data holder or a leaf-list, where a leaf holds a single primitive data type (in our case its a leaf containing a string named *name*).

After writing the YANG module, we tried to build the project by running ‘*mvn –Pq clean install’* from the project’s root folder. We may encounter some errors here regarding dependencies, but they are quite informative and easily debugable.

Assuming we passed the maven build, we should have a jar corresponding to each leaf project AKA **bundle** (which we can identify by checking its <*packaging>* tag in the POM is valued with *bundle*).  
  
This jar is then inserted into the karaf’s deploy library, regardless of its running condition, since the karaf supports “hot” jar deployments.  
  
We can then verify our progress so far by entering the RESTCONF API (**NEED TO INSERT LINK HERE**), and looking for the module’s name on the list, trying to operate with it using the REST API.

Having that sorted out, we can write the listener which will write a message to the LOG upon every change in the greeter that resides in the MD-SAL(?).

The ***impl*** directory contains two main components, the java code and the blueprint.

First, inside the blueprint XML we define the the listener class we’re about to construct, ***HelloListener***, alongside with:

* Constructor’s arguments (defined as an **argument** tag)
* Init function, which is invoked after the listener’s constructor
* Destroy function, which is invoked before listener’s destruction

Then there’s the Java file itself, in which we implement the class:

|  |
| --- |
| public class HelloListener extends AsyncDataTreeChangeListenerBase<SayHelloToName, HelloListener> |
|  | implements AutoCloseable |

***AsyncDataTreeChangeListenerBase*** has a matching abstract method for every REST command, so the implementation is pretty straighforward for each case.

***SayHelloToName*** is the contrainer in the YANG we created earlier, which is generated into Java code.

The logger is a static member of the class, initialized as follows:

private static final Logger LOG = LoggerFactory.getLogger(HelloListener.class);

After completing the *c’tor, start, add* and *update* functions, we can already send PUT https calls, and see the result in the log at (**INSERT PATH**).

Remaining questions so far:

* Q:How does the class gets instantiated during in case of “hot” deployment? Does the Karaf lisents to incoming jars?
* A: It has some form of polling machanism over the bundles, basically addressing the blueprints. Requires the code itself to be hot deployable.
* Q: ***InstanceIdentifier*** is a bit unclear, not sure whether its binded to an instance, or a class.
* A: It is binded to a class. It is more required when addressing complex YANG models.
* Q:What is the functionality of the***DataBroker*** class, it definitely relates to the listener’s registry but hard to see exactly how.
* A: DataBroker is an MDSAL creature, required to communicate and transfer data between our Java code to the MDSAL instantiated models.

Second and Third days (7-8\5\17), ODL-Calculator:

Three sub-projects:

* **odl-calculator-api:**
  + Has two APIs:
    - calculator-api, hold the input query string
    - calculator-operational-api, holds the final result
  + One listener implementation
    - Public class CalculatorOutputListener extends AsyncDataTreeChangeListenerBase<OdlCalculatorOpenrationalApi, CalculatorOutputListener> implements AutoCloseable  
      This listener waits for the OdlCalculatorOpenrationalApi to be modified, and outputs the result into the karaf log.
* **odl-calculator-parser:** 
  + Has a single API:
    - Parser-api, hold a query list, where each query holds a string. Should have been implemented with leaf-list. Holds the parsed query.
  + And a single Listener:
    - Public class CalculatorInputListener extends AsyncDataTreeChangeListenerBase<OdlCalculatorApi, CalculatorInputListener> implements AutoCloseable.  
      This listener waits for a query to be inserted into the OdlCalculatorApi, it then parses it into a string list, and creates an instance of parser-api which hold the parsed query.
* **odl-calculator-impl**:
  + Has a single listener:
    - Public class ParserListener extends AsyncDataTreeChangeListenerBase<ParserApi, ParserListener> implements AutoCloseable.   
      This listener waits for a parsed query to be inserted into the Parser southbound, calculates the output, and writes it to the OdlCalculatorOpenrationalApi.

The most important thing we learnt here, which did not exist in the previous project, is how to instantiate a YANG model and inject it with data, using the code instead of the REST calls. It also revealed the place where we actually need to dataBroker, instead of just using it at the initial registration. We can refer to both the parser and the impl project for the full code.

The “punch” line for writing data is the following:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | | MDSALUtil.syncWrite(databroker,LogicalDatastoreType.CONFIGURATION,  InstanceIdentifier.builder(ParserApi.class).build(), dataObject); | | The first three arguments are pretty trivial, we need the dataBroker of the current class, the type of datastore we are about to write to (determined by the *config* directive in the YANG model), and an instance indentifier for the YANG model.  Building data object is the non static part, where we need to use the build function to instantiate every sub element in the model. |  | | |
|  |  |