**OpenDayLight**

**Installation of Boron Distribution version**

According to Eric, currently, we are investigating with Boron first. We can use following command to download the distribution:

wget https://nexus.opendaylight.org/content/repositories/public/org/opendaylight/integration/distribution-karaf/0.5.4-Boron-SR4/[distribution-karaf-0.5.4-Boron-SR4.zip](https://nexus.opendaylight.org/content/repositories/public/org/opendaylight/integration/distribution-karaf/0.5.4-Boron-SR4/distribution-karaf-0.5.4-Boron-SR4.zip)

Then we can unzip and start from bin/karaf. ODL uses Karaf for deploying and managing OSGi bundles.

**Developing FXN Apps**

**Pre requisites**

A development environment with following set up and working correctly from the shell:

* Maven 3.1.1 or later
* Java 7- or Java 8-compliant JDK

1. Set the environments. In my .bash\_aliases, I have the following env variables:

$ cat .bash\_aliases

export JAVA\_HOME=/usr/lib/jvm/java-8-openjdk-amd64

export M2\_HOME=/usr/share/maven

export MAVEN\_OPTS="-Xms256m -Xmx512m" # Very important to put the "m" on the end

source .bashrc

1. Set the settings for default OpenDaylight

cp -n ~/.m2/settings.xml{,.orig} ; wget -q -O - https://raw.githubusercontent.com/opendaylight/odlparent/master/settings.xml > ~/.m2/settings.xml

**Building an example module**

1. Set the env for maven archetype

export GROUP\_ID=org.opendaylight.controller

export ARTIFACT\_ID=opendaylight-startup-archetype

export ARCHETYPE\_VERSION=1.2.4-Boron-SR4

1. Create a project using Maven and an archetype called the *opendaylight-startup-archetype* to pull and generate the project

mvn archetype:generate -DarchetypeGroupId=$GROUP\_ID -DarchetypeArtifactId=$ARTIFACT\_ID -DarchetypeRepository=http://nexus.opendaylight.org/content/repositories/opendaylight.release/ -DarchetypeCatalog=remote -DarchetypeVersion=$ARCHETYPE\_VERSION

1. Update the properties values as follows. Ensure that the groupid and the artifactid is lower case.

Define value for property 'groupId': : org.opendaylight.fxn

Define value for property 'artifactId': : fxn

Define value for property 'version': 1.0-SNAPSHOT: : 1.0.0-SNAPSHOT

Define value for property 'package': org.opendaylight.fxn: :

Define value for property 'classPrefix': ${artifactId.substring(0,1).toUpperCase()}${artifactId.substring(1)}

Define value for property 'copyright': : FXN, Inc.

**App developments**

1. **Define a YANG Model**

From your project directory, go to api/src/main/yang and modify the yang model for your project. Keep the prefix as your project name and define the RPC.

Here is my fxn project

yk@yk-VirtualBox:~/fxn/api/src/main/yang$ cat fxn.yang

module fxn {

yang-version 1;

namespace "urn:opendaylight:params:xml:ns:yang:fxn";

**prefix "fxn"**;

revision "2020-09-15" {

description "Initial revision of fxn model";

}

**rpc fxn-odl** {

input {

leaf name {

type string;

}

}

output {

leaf greeting {

type string;

}

}

}

}

Return to the project/api directory and build your API as follows.

cd ../../../

mvn clean install

Yangtools will generate several files at org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.fxn.rev200915. We should not change these files.

1. **Implement the RPC API**

We need to define the service and implement the service, and register the RPC service. To define the service, go to the impl directory

cd ../impl/src/main/java/org/opendaylight/fxn/impl/

and modify the service provider class. Here is the example of my FxnProvider class:

yk@yk-VirtualBox:~/fxn/impl/src/main/java/org/opendaylight/fxn/impl$ cat FxnProvider.java

/\*

\* Copyright © 2016 Copyright (c) 2020 FXN, Inc. and others. All rights reserved.

\*

\* This program and the accompanying materials are made available under the

\* terms of the Eclipse Public License v1.0 which accompanies this distribution,

\* and is available at http://www.eclipse.org/legal/epl-v10.html

\*/

package org.opendaylight.fxn.impl;

import org.opendaylight.controller.md.sal.binding.api.DataBroker;

import org.opendaylight.controller.sal.binding.api.RpcProviderRegistry;

import org.opendaylight.controller.sal.binding.api.BindingAwareBroker.RpcRegistration;

import org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.fxn.rev200915.FxnService;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class FxnProvider {

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

private final DataBroker dataBroker;

private final RpcProviderRegistry rpcProviderRegistry;

private RpcRegistration<FxnService> serviceRegistration;

public FxnProvider(final DataBroker dataBroker, RpcProviderRegistry rpcProviderRegistry) {

this.dataBroker = dataBroker;

this.rpcProviderRegistry = rpcProviderRegistry;

}

/\*\*

\* Method called when the blueprint container is created.

\*/

public void init() {

serviceRegistration = **rpcProviderRegistry**.addRpcImplementation(FxnService.class, new FxnOdlImpl());

LOG.info("FxnProvider Session Initiated");

}

/\*\*

\* Method called when the blueprint container is destroyed.

\*/

public void close() {

serviceRegistration.close();

LOG.info("FxnProvider Closed");

}

}

At the same place, create a new file to implement the interface FxnService.java, which is generated from Yangtools at the previous step. Here is my example code:

yk@yk-VirtualBox:~/fxn/impl/src/main/java/org/opendaylight/fxn/impl$ cat FxnOdlImpl.java

/\*

\* Copyright © 2016 Copyright (c) 2020 FXN, Inc. and others. All rights reserved.

\*

\* This program and the accompanying materials are made available under the

\* terms of the Eclipse Public License v1.0 which accompanies this distribution,

\* and is available at http://www.eclipse.org/legal/epl-v10.html

\*/

package org.opendaylight.fxn.impl;

import java.util.concurrent.Future;

import org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.fxn.rev200915.FxnService;

import org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.fxn.rev200915.FxnOdlInput;

import org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.fxn.rev200915.FxnOdlOutput;

import org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.fxn.rev200915.FxnOdlOutputBuilder;

import org.opendaylight.yangtools.yang.common.RpcResult;

import org.opendaylight.yangtools.yang.common.RpcResultBuilder;

public class FxnOdlImpl **implements FxnService** {

@Override

public Future<RpcResult<FxnOdlOutput>> fxnOdl(FxnOdlInput input) {

FxnOdlOutputBuilder fxnBuilder = new FxnOdlOutputBuilder();

fxnBuilder.setGreeting("Hello " + input.getName());

return RpcResultBuilder.success(fxnBuilder.build()).buildFuture();

}

}

1. **Modify Blueprint**

Blueprint container: ODL uses blueprint for dependency injection across bundles that run in an OSGi framework. For example, we used RpcProviderRegistry, we can find its interface information from <https://javadocs.opendaylight.org/controller/oxygen/org/opendaylight/controller/sal/binding/api/RpcProviderRegistry.html>. All the controller interfaces at <https://javadocs.opendaylight.org/controller/oxygen/overview-summary.html>

Modify this file for the app as following:

yk@yk-VirtualBox:~/fxn/impl/src/main/resources/org/opendaylight/blueprint$ cat impl-blueprint.xml

<?xml version="1.0" encoding="UTF-8"?>

<!-- vi: set et smarttab sw=4 tabstop=4: -->

<!--

Copyright © 2016 Copyright (c) 2020 FXN, Inc. and others. All rights reserved.

This program and the accompanying materials are made available under the

terms of the Eclipse Public License v1.0 which accompanies this distribution,

and is available at http://www.eclipse.org/legal/epl-v10.html

-->

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"

xmlns:odl="http://opendaylight.org/xmlns/blueprint/v1.0.0"

odl:use-default-for-reference-types="true">

<reference id="dataBroker"

interface="org.opendaylight.controller.md.sal.binding.api.DataBroker"

odl:type="default" />

**<reference id="rpcRegistry"**

**interface="org.opendaylight.controller.sal.binding.api.RpcProviderRegistry"/>**

<bean id="provider"

class="org.opendaylight.**fxn.impl.FxnProvider**"

init-method="init" destroy-method="close">

<argument ref="dataBroker" />

**<argument ref="rpcRegistry" />**

</bean>

</blueprint>

1. **Final Build**
2. Optionally, you can also build the Java classes which will register the new RPC. This is useful to test the edits you have made to these two java files.

cd ../../../../../../../

mvn clean install

1. Return to the top level directory

cd ../

1. Build the entire project again, which will pickup the changes you have made and build them into your project:

mvn clean install -DskipTests

**Execute the App**

Now you are ready to execute your project. Sodium release provides better instruction on testing the app.

1. Run karaf

cd ../karaf/target/assembly/bin

./karaf

1. Wait for the project to load completely. Then view the log to see the loaded FXN Module:

log:display | grep fxn

At the following website for testing, the username/password is admin/admin

<http://localhost:8181/index.html>

RESTCONF documentation at

<http://localhost:8181/apidoc/explorer/index.html>

**Notes:**

* In karaf, we should install the following features

feature:install odl-restconf odl-mdsal-apidocs odl-dlux-core

* At project level, you might use **mvn clean install –DskipTests.** I skipped tests on Boron version. For Sodium, I didn’t skipTests and it is fine.