OpenDaylight Application Developers' Tutorial

This tutorial is intended for developers new to SDN application development. Some familiarity with Java and IDEs such as Eclipse will prove useful, though isn't absolutely necessary.

1. Setup

To get started, download and set up the SDN Hub Tutorial VM in Virtualbox or VMware. The actual source for this tutorial application is available at https://github.com/sdnhub/SDNHub_Opendaylight_Tutorial

2. Quickstart

- Boot up our tutorial VM with OpenDaylight already installed. The complete source code for the Hydrogen Release is at /home/ubuntu/opendaylight, and the tutorial application that we will work with is located in /home/ubuntu/SDNHub_OpenDaylight_tutorial directory.
- If you're interested in setting up the OpenDaylight installation from scratch, follow the instructions here
- Let's now create a simple (virtual) network topology using Mininet and Open vSwitch. In a Terminal window, run the following command, which starts a network emulation environment to emulate 1 switch with 3 hosts, that attempts to connect to a remote controller.

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

- You will now see a mininet terminal. Begin a ping between two of the hosts. The ping fails because there is no intelligence in the switch to learn the MAC addresses of each host and forward traffic to the correct switch ports

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
```

- Let us now add that intelligence. Open a Terminal tab (Ctrl-Shift-T) and go into the SDNHub_Opendaylight_Tutorial folder and run the following commands to build the example application we will walk through for the tutorial – a Hub / L2 learning switch:

```
ubuntu@ubuntu:~$ cd SDNHub_Opendaylight_Tutorial
ubuntu@ubuntu:~/SDNHub_Opendaylight_Tutorial$ cd commons/parent
ubuntu@ubuntu:~/SDNHub_Opendaylight_Tutorial/commons/parent$ mvn clean install
```

- This command uses Apache Maven to build the tutorial code. "mvn" is the maven command that compiles code based on the pom.xml file in that directory. "clean" is not essential if you do not wish to clean the temporary build files. "install" is essential for compilation.
- The way Maven works is by resolving dependencies between packages. The tutorial_L2_forwarding (the hub/switch code we will edit) depends on the Opendaylight controller package, so Maven will download the pre-compiled jar files from Opendaylight.org, resolve dependencies, and so on. This is much, much faster than compiling the tutorial code within the source of the Opendaylight controller.
  - **Note:** to speed up subsequent compilations, you can run "mvn install -DskipTests -DskipIT -nsu". The "nsu" is short for no-snapshot-updates. It ensures that the compilation does not download definitions from nexus.opendaylight.org

- Next, let's run the controller itself

```
ubuntu@ubuntu:~/SDNHub_Opendaylight_Tutorial/commons/parent$ cd
../../distribution/opendaylight/target/distribution.tutorial_L2_forwarding-1.0.0-SNAPSHOT-osgipackage/opendaylight
ubuntu@ubuntu:~/SDNHub_Opendaylight_Tutorial/distribution/opendaylight/target/distribution.tutorial_L2_forwarding-1.0.0-
SNAPSHOT-osgipackage/opendaylight$ ./run.sh
```

- The above command starts all the OSGi bundles installed as jar files in the plugins directory. Once all bundles are started, you will be able to check the status of bundles using the "ss" command

```
osgi> ss tutorial
"Framework is launched."
id    State       Bundle
36    ACTIVE      org.sdnhub.tutorial_L2_forwarding.sdnhub.tutorial_L2_forwarding_0.5.0.SNAPSHOT
```

- Since OpenDaylight today does not have a way to specific which modules get precedence over other moduels, we require that the SimpleForwarding and ARPHandler modules be stopped using the "stop" OSGi command. This way, all packet forwarding decisions will be left to the tutorial_L2_forwarding module.

```
osgi> ss simple
"Framework is launched."
id    State       Bundle
64    RESOLVED    org.opendaylight.controller.samples.simpleforwarding_0.4.1

osgi> ss arp
```

```
"Framework is launched."
id    State       Bundle
200    ACTIVE      org.opendaylight.controller.arphandler_0.5.1

osgi> stop 200
```

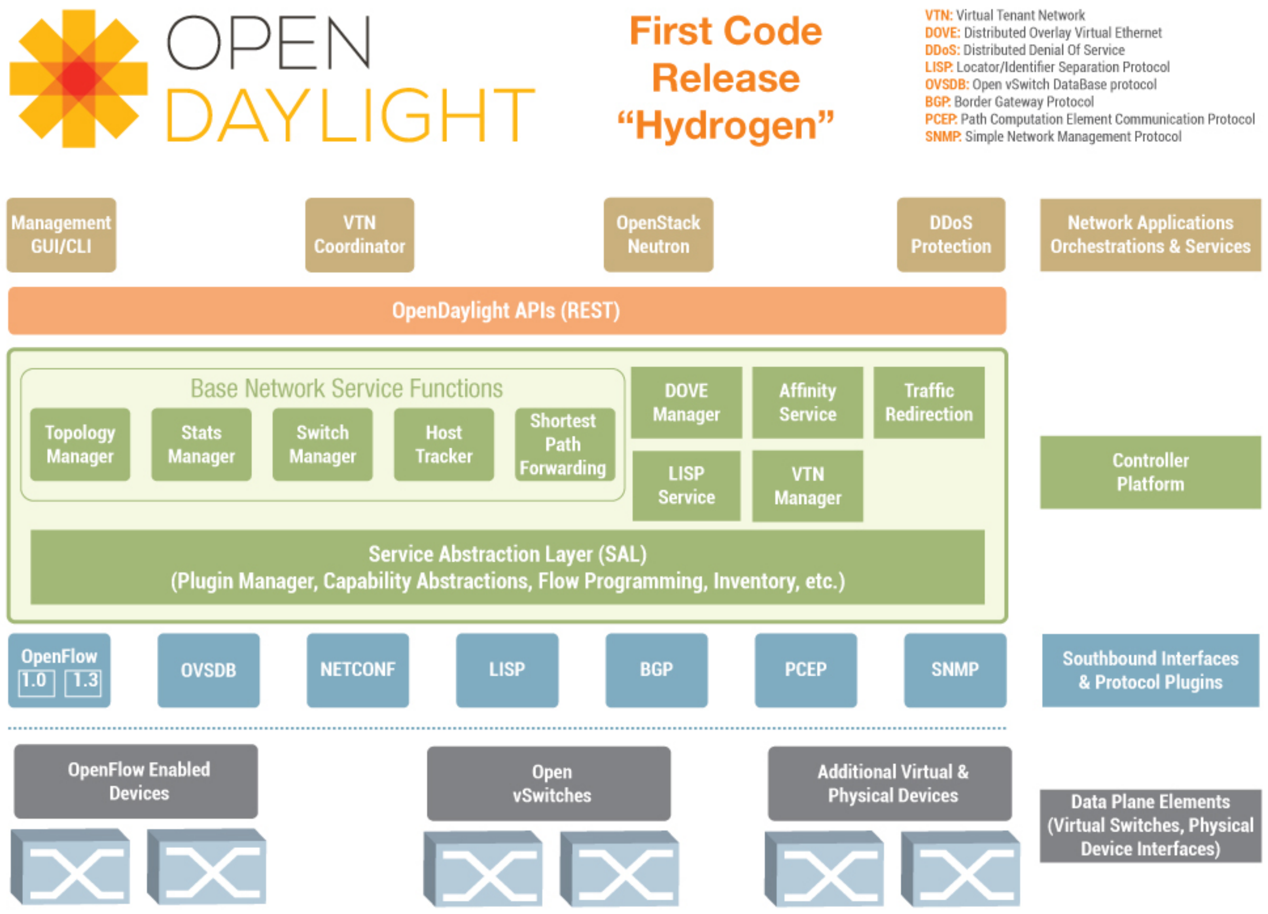
- After the controller finishes loading, go back to the mininet terminal. You will see that the pings now succeed

```
From 10.0.0.1 icmp_seq=17 Destination Host Unreachable
From 10.0.0.1 icmp_seq=18 Destination Host Unreachable
64 bytes from 10.0.0.2: icmp_req=19 ttl=64 time=94.1 ms
64 bytes from 10.0.0.2: icmp_req=20 ttl=64 time=21.2 ms
64 bytes from 10.0.0.2: icmp_req=21 ttl=64 time=21.9 ms
```

- You will have noticed that the pings are actually taking longer than they should for a learning switch (sub-1ms) for such a simple topology. This is because the controller is currently functioning in “Hub” mode. In the next few sections, we will walk through how we can turn the hub into a switch.

3. Introduction to Open Daylight

Before we jump into the code, a high-level overview of the Opendaylight controller is in order


OPEN DAYLIGHT
First Code Release "Hydrogen"
VTN: Virtual Tenant Network
DOVE: Distributed Overlay Virtual Ethernet
DDoS: Distributed Denial Of Service
LISP: Locator/Identifier Separation Protocol
OVSDB: Open vSwitch DataBase protocol
BGP: Border Gateway Protocol
PCEP: Path Computation Element Communication Protocol
SNMP: Simple Network Management Protocol
Management GUI/CLI
VTN Coordinator
OpenStack Neutron
DDoS Protection
Network Applications Orchestrations & Services
OpenDaylight APIs (REST)
Base Network Service Functions
Topology Manager
Stats Manager
Switch Manager
Host Tracker
Shortest Path Forwarding
DOVE Manager
Affinity Service
Traffic Redirection
LISP Service
VTN Manager
Controller Platform
Service Abstraction Layer (SAL)
(Plugin Manager, Capability Abstractions, Flow Programming, Inventory, etc.)
OpenFlow 1.0 1.3
OVSDB
NETCONF
LISP
BGP
PCEP
SNMP
Southbound Interfaces & Protocol Plugins
OpenFlow Enabled Devices
Open vSwitches
Additional Virtual & Physical Devices
Data Plane Elements (Virtual Switches, Physical Device Interfaces)

OpenDayLight uses the following software tools/paradigms. It is important to become familiar with them. The Service Abstraction Layer (SAL) is your friend for most development aspects.

- **Maven**: OpenDayLight uses Maven for easier build automation. Maven uses pom.xml (Project Object Model for this bundle) to script the dependencies between bundles and also to describe what bundles to load on start.
- **OSGi**: This framework in the backend of OpenDayLight allows dynamically loading bundles and packaged Jar files, and binding bundles together for information exchange.
- **Java interfaces**: Java Interfaces are used for event listening, specifications and forming patterns. This is the main way in which specific bundles implement call-back functions for events and also to indicate awareness of specific state.

The controller platform has certain core bundles, each of which export important services through Java interfaces. Here is a brief list of important ones that come in handy while developing network services. See **this link** for more information.

Bundle	Exported interface	Description
arphandler	IHostFinder	Component responsible for learning about host location by handling ARP.
hosttracker	IfIptoHost	Track the location of the host relatively to the SDN network.
switchmanager	ISwitchManager	Component holding the inventory information for all the known nodes (i.e., switches) in the controller.
topologymanager	ITopologyManager	Component holding the whole network graph.
usermanager	IUserManager	Component taking care of user management.
statisticsmanager	IStatisticsManager	Component in charge of using the SAL ReadService to collect several statistics from the SDN network.
sal	IReadService	Interface for retrieving the network node's flow/port/queue hardware view
sal	ITopologyService	Topology methods provided by SAL toward the applications
sal	IFlowProgrammerService	Interface for installing/modifying/removing flows on a network node
sal	IDataPacketService	Data Packet Services SAL provides to the applications
web	IDaylightWeb	Component tracking the several pieces of the UI depending on bundles installed on the system.

4. Basic steps to writing an OpenDayLight Application

For beginners, we recommend using Eclipse for viewing and editing the source code, and command-line for compiling and running the controller. Open Eclipse using the desktop shortcut.

We have already setup the Eclipse environment for you. But, if you wish to do it from scratch, you should follow these instructions to setup Maven-Eclipse integration. After that you can and go to File->Import. Choose 'Maven > Existing Maven Projects'. On the next screen, choose as the Root Directory ~/ubuntu/SDNHub_Opendaylight_Tutorial/. Select the pom.xml that it finds, click on 'Add project(s) to working set' below, and click 'Finish'.

Once Eclipse is setup, you should now see 'sdnhub.tutorial_L2_forwarding' on your Project explorer pane. Double-click on src/main/java > org.opendaylight.controller.tutorial_L2_forwarding > TutorialL2Forwarding.java and follow the walk-through below.

Most controller platforms expose some native features to allow these key features:

A. Ability to listen to asynchronous events (e.g., PACKET_IN, FLOW_REMOVED) and to register call back functions (e.g., receiveDataPacket)
B. Ability to parse incoming packets (e.g., ARP, ICMP, TCP) and fabricate packets to send out into the network
C. Ability to create and send an OpenFlow/SDN message (e.g., PACKET_OUT, FLOW_MOD, STATS_REQUEST) to the programmable dataplane.
D. Managing decision making modules on the same controller

With OpenDayLight you can achieve all of those using a combination of SAL implementation classes and listener interfaces.

A. Receiving events

There are listener interfaces for several types of events including events thrown by modules like TopologyManager. For instance, to allow a module to receive packets sent by the switch to the controller, the class needs to implement IListenDataPacket:

```
public class TutorialL2Forwarding implements IListenDataPacket
```

When the class implements that *IListenDataPacket* interface, any packet that arrives at a node without a pre-cached rule will result in a callback of *receiveDataPacket*. So overriding that method allows the application to get a copy of the packet.

```
public PacketResult receiveDataPacket(RawPacket inPkt) { ... }
```

One other feature that was useful was generating the list of ports in a node/switch. Assuming *incoming_node* is the switch whose ports I want to list, I use the SwitchManager class as follows:

```
Set<NodeConnector> nodeConnectors =
this.switchManager.getUpNodeConnectors(incoming_node);
```

B. Parsing packets

Once the packet is received, you can decode the packet using the *dataPacketService*. Usage of command is shown below. Today, according to the *org.opendaylight.controller.sal.packet* package, we can inspect the packet headers the following packet types: ARP, Ethernet, ICMP, IPv4, LLDP, TCP, UDP.

```
Packet formattedPak = this.dataPacketService.decodeDataPacket(inPkt);
```

Along with the packet header fields, you can extract the incoming node (i.e., switch) and node-connector (i.e., switchport) using the following:

```
NodeConnector incoming_connector = inPkt.getIncomingNodeConnector();
```

We use the following two useful commands to extract header details to use in the MAC learning switch:

```
byte[] srcMAC = ((Ethernet)formattedPak).getSourceMACAddress();
```

```
long srcMAC_val = BitBufferHelper.toNumber(srcMAC);
```

C. Sending messages to switch

Once the decisions are taken and the controller decides that a packet needs to be forwarded, I can use the *dataPacketService* to send a packet out of a node-connector (switchport). In the following example, we resend the incoming packet out of node-connector ‘p’:

```
RawPacket destPkt = new RawPacket(inPkt);
destPkt.setOutgoingNodeConnector(p);
this.dataPacketService.transmitDataPacket(destPkt);
```

Besides a PACKET_OUT, we can also perform a FLOW_MOD style rule insertion into a switch. For that, we use the Match, Action and Flow classes. Here is an example of how to create a match rule where the IN_PORT matches the value extracted from the PACKET_IN:

```
Match match = new Match();
match.setField(MatchType.IN_PORT, incoming_connector);
```

There are several other fields you can match one. Following are what I see in the *org.opendaylight.controller.sal.match*: “inPort”, “dlSrc”, “dlDst”, “dlVlan”, “dlVlanPriority”, “dlOuterVlan”, “dlOuterVlanPriority”, “dlType”, “nwTOS”, “nwProto”, “nwSrc”, “nwDst”, “tpSrc”, “tpDst”.

In the package *org.opendaylight.controller.sal.action*, I see the following supported actions: “drop”, “loopback”, “flood”, “floodAll”, “controller”, “interface”, “software path”, “harware path”, “output”, “enqueue”, “setDlSrc”, “setDlDst”, “setVlan”, “setVlanPcp”, “setVlanCif”, “stripVlan”, “pushVlan”, “setDlType”, “setNwSrc”, “setNwDst”, “setNwTos”, “setTpSrc”, “setTpDst”, “setNextHop”. Here is an example of creating an action list for the flow rule, for OUTPUT to an*outgoing_connector* that is pre-calculated.

```
List<Action> actions = new ArrayList<Action>();
actions.add(new Output(outgoing_connector));
```

Once the match rule is formed and action list is created, we can create a Flow and add it to a particular switch. In this case, the rule is inserted to the switch where the original packet arrived:

```
Flow f = new Flow(match, actions);
Status status = programmer.addFlow(incoming_node, f);
if (status.isSuccess())
    logger.trace("Installed flow {} in node {}", f, incoming_node);
else
    //What to do in case of flow insertion failure?
```

D. Managing multiple modules

As highlighted in the Quick-start, it is possible that there are multiple modules contesting for the same events and want to act upon it. OpenDaylight today does not provide a precedence order among modules. This requires some coordination among modules to ensure there is no conflict.

Given that, it is possible for a module to signal to the platform when it has finished processing a packet. Once the packet is appropriately handled, the controller application has 3 choices on returning status back to the SAL. It can be

- return PacketResult.CONSUME: Packet has been processed and none in the chain after us should handle it
- return PacketResult.KEEP_PROCESSING: Packet has been processed, but still pass onto other bundles
- return PacketResult.IGNORED: Packet has been ignored and ready to pass to other bundles

Now that you're updated on the theory, you are now ready to create a new module.

5. Sample application: Learning Switch

For the purposes of this tutorial, you should attempt to convert the hub application to a MAC learning switch.
build a hub and/or a MAC learning switch using the above code snippets. The main logic for hub and learning switch are **available here**.

Modify the implementation of the application *tutorial_L2_forwarding* in the SDNHub_OpenDaylight_tutorial directory by editing TutorialL2Forwarding.java. Convert the Hub module to the learning switch. We have provided hints on what needs to change. Once you successfully compile and run, you will see that mininet ping succeeds (**Note**: RTT time is less than 1ms once flow rules are inserted, unlike in the hub case):

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
From 10.0.0.1 icmp_seq=4 Destination Host Unreachable
64 bytes from 10.0.0.2: icmp_req=5 ttl=64 time=0.529 ms
64 bytes from 10.0.0.2: icmp_req=6 ttl=64 time=0.133 ms
64 bytes from 10.0.0.2: icmp_req=7 ttl=64 time=0.047 ms
```

The OSGi console shows the following sample log:

```
2013-06-13 21:00:31.844 PDT [SwitchEvent Thread] INFO o.o.c.t.i.TutorialL2Forwarding
- Installed flow Flow[match = Match[[DL_DST(00:00:00:00:00:01,null),
IN_PORT(OF|2@OF|00:00:00:00:00:00:00:01,null)]],
actions = [OUTPUT[OF|1@OF|00:00:00:00:00:00:00:01]], priority = 0, id = 0,
idleTimeout = 0, hardTimeout = 0] in node OF|00:00:00:00:00:00:00:01
```

Solutions

In the tutorial_L2_forwarding internal directory, you will see three solutions files:

- *TutorialL2Forwarding_singleswitch.solution*: L2 MAC Learning switch that pushes flow_mod rules to the switch based on the packet_in. It only tracks MAC for a single switch.
- *TutorialL2Forwarding_multiswitch.solution*: L2 MAC Learning switch that pushes flow_mod rules to the switch based on the packet_in. It tracks MAC for multiple switches independently.
- *TutorialL2Forwarding_statelesslb.solution*: Stateless load-balancer code where you can statically set the list of servers and their IP addresses. When traffic arrives for the virtual IP of the load-balancer, it selects the server in a round-robin basis and sends request to a different server after doing IP header rewriting in both directions.

6. Workflow, dependencies and information sharing

Maven and OSGI

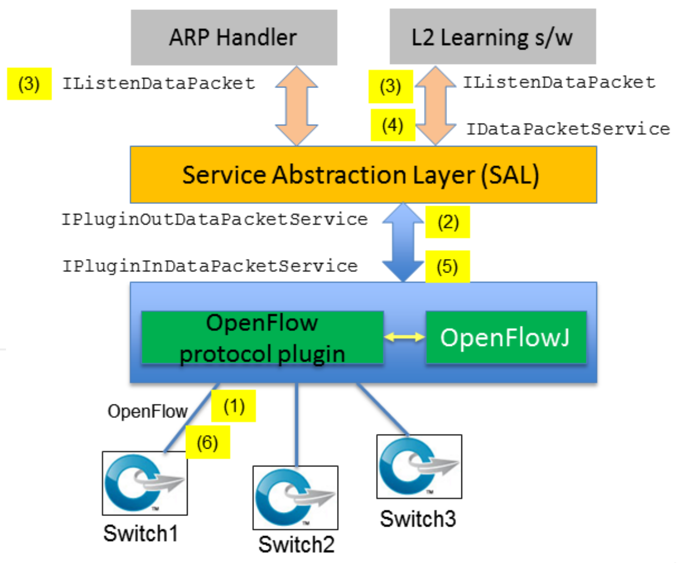
- **pom.xml**: Generally the pom.xml is used to indicate the inter-bundle dependencies. The syntax for that must be fairly apparent from the format of the pom.xml in the arphandler directory
- **Activator.java**: This handles the configuration of the dependencies during run-time. The configureInstance function in the bottom of Activator.java of each bundle has two main tasks:
  - tells the list of Java interfaces implemented by a bundle 'X' using *setInterface* call
  - registers other bundles that the bundle 'X' uses using *add* call
- **set/unset binding**: The bundle 'X' gets a reference to the instantiated object of other needed bundle 'Y' through the *setY* and *unsetY* calls in the main class. For example, TutorialL2Forwarding needs data from SwitchManager. This is accomplished by defining two methods *setSwitchManager* and *unsetSwitchManager*. These are called by OSGi and object references are appropriately passed.

Life of a packet

In OpenDaylight, the SAL is incharge of all plumbing between the applications and the underlying plugins. Here is an illustration of the life of a packet.

- A packet arriving at Switch1 will be sent to the appropriate plugin managing the switch
- The plugin will parse the packet, generate an event for SAL
- SAL will dispatch the packet to the modules listening for DataPacket
- Module handles packet and sends packet_out through IDataPacketService

- SAL dispatches the packet to the modules listening for DataPacket
- OpenFlow message sent to appropriate switch



7. Debugging

If we want to debug your code, you need to start OpenDayLight including the parameter "-debug":

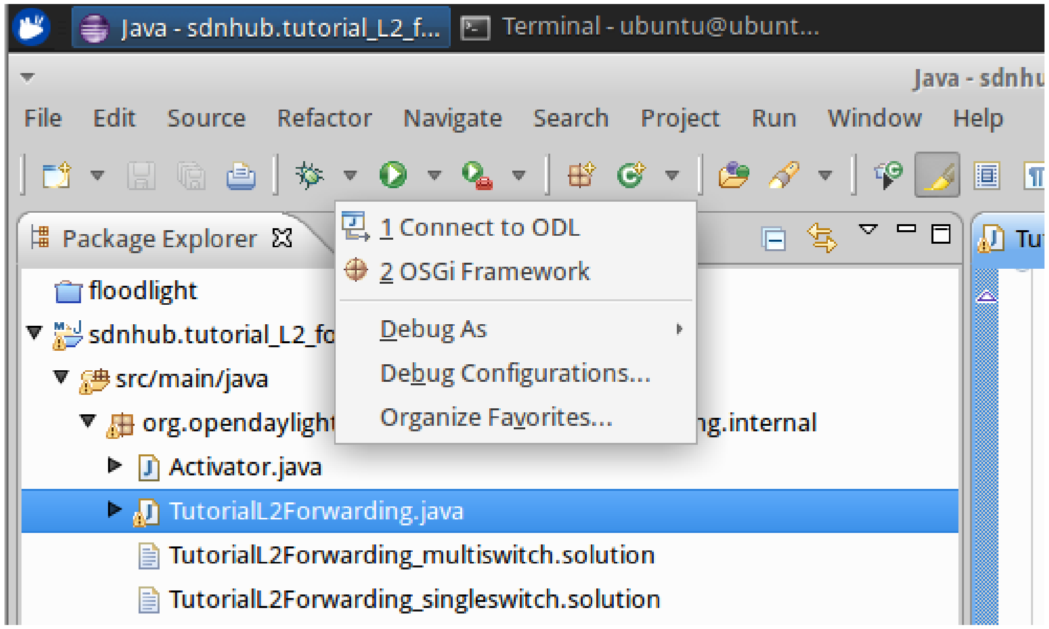
```
ubuntu@ubuntu:~/SDNHub_Opendaylight_Tutorial/commons/parent$ cd
../../distribution/opendaylight/target/distribution.tutorial_L2_forwarding-
1.0.0-SNAPSHOT-osgipackage/opendaylight


ubuntu@ubuntu:~/SDNHub_Opendaylight_Tutorial/distribution/opendaylight/target/distribution.tutorial_L2_forwarding-1.0.0-
SNAPSHOT-osgipackage/opendaylight$ ./run.sh -debug
```

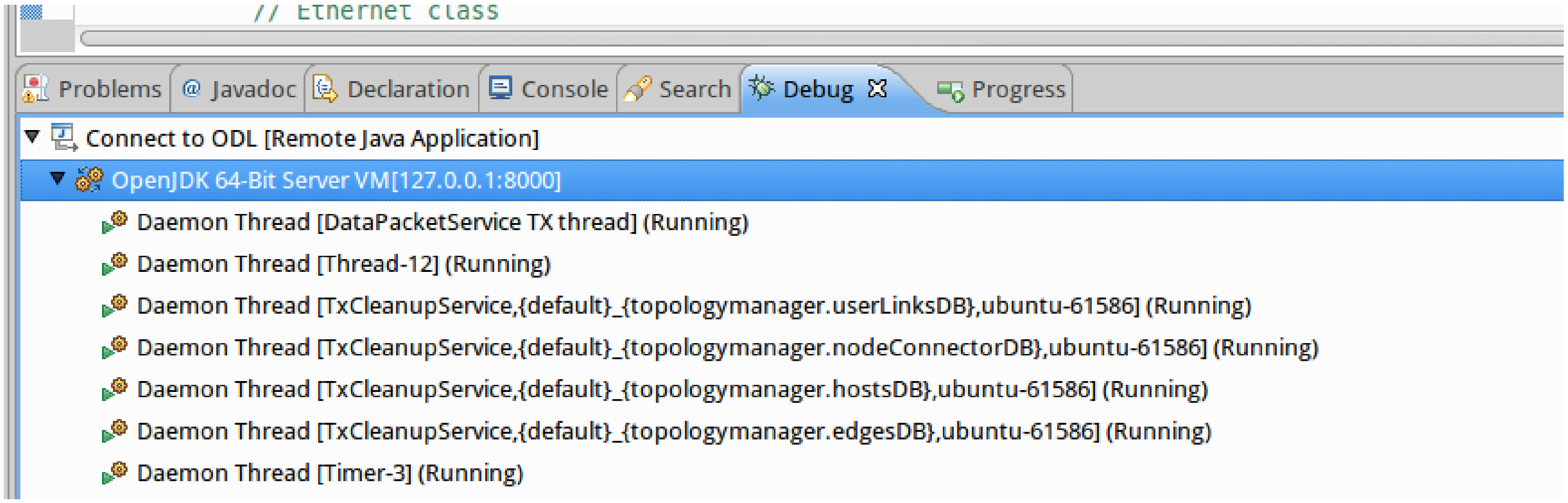
"-debug" will allow us to connect to OSGI (listening at port 8000) for debugging purposes.

Also you don't need to relaunch OpenDayLight or reload your module every time you made changes to your code, you just need to save them using eclipse and it will automatically update the module.

For Debug the code you can call "Connect to ODL" and it will allow to debug your code using all the eclipse benefits.


Java - sdnhub.tutorial_L2_f...
Terminal - ubuntu@ubunt...
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
1 Connect to ODL
2 OSGi Framework
Debug As
Debug Configurations...
Organize Favorites...
floodlight
sdnhub.tutorial_L2_fo
src/main/java
org.opendaylight
ng.internal
Activator.java
TutorialL2Forwarding.java
TutorialL2Forwarding_multiswitch.solution
TutorialL2Forwarding_singleswitch.solution

If everything works fine, you should be able to see the Debug tab:



```
Connect to ODL [Remote Java Application]
  OpenJDK 64-Bit Server VM[127.0.0.1:8000]
    Daemon Thread [DataPacketService TX thread] (Running)
    Daemon Thread [Thread-12] (Running)
    Daemon Thread [TxCleanupService,{default}_{topologymanager.userLinksDB},ubuntu-61586] (Running)
    Daemon Thread [TxCleanupService,{default}_{topologymanager.nodeConnectorDB},ubuntu-61586] (Running)
    Daemon Thread [TxCleanupService,{default}_{topologymanager.hostsDB},ubuntu-61586] (Running)
    Daemon Thread [TxCleanupService,{default}_{topologymanager.edgesDB},ubuntu-61586] (Running)
    Daemon Thread [Timer-3] (Running)
```

Copyright 2013 Software-defined Networking Hub

Terrifico Theme powered by WordPress