



# **OpenDaylight SDN controller platform**

**A Degree Thesis**

**Submitted to the Faculty of the**

**Escola Tècnica d'Enginyeria de Telecomunicació de  
Barcelona**

**Universitat Politècnica de Catalunya**

**by**

**BERNAT RIBES GARCIA**

**In partial fulfilment**

**of the requirements for the degree in**

**CIÈNCIES I TECNOLOGIES**

**DE LES TELECOMUNICACIONS**

**Advisor: José Luis Muñoz**

**Barcelona, October 2015**

## **Abstract**

Software Defined Networks (SDN) are presented as a new paradigm with the objective of simplifying the network creation and management. SDN networks are based in the physical separation of the control plane from the forwarding plane, by introducing a centralized logical element, the controller.

OpenDaylight (ODL) project, leaded by Linux, defines an implementation of an SDN controller, which offers an innovation regarding the other SDN solutions: the Service Abstraction Layer (SAL). The SAL is a layer that provides a high level of abstraction, since applications can communicate with network elements without being aware of which network protocols implement them.

SDN controller also contains a set of modules, which contain plugins, that allow the communication with the different network protocols, and applications, that allow to obtain information about the network (topology, resources, etc.).

To achieve this abstraction level, OpenDaylight makes use of a set of third party technologies that leverage the model-driven architecture. This document aims to detail all the technologies that are involved in the development of an OpenDaylight application, together with the internal ODL modules that interact.

By the end of this document, all the required steps to develop an OpenDaylight application will be detailed. Two perspectives will be given:

- Internal: all the modules that are involved in the internal process that OpenDaylight makes when developing an application will be detailed, from its model definition until its deployment.
- Developer: all the required steps to develop an application will be detailed: generating models to define our application, provide its implementations, identifying its dependencies, wiring our application with the SAL, how to deploy it, etc.

This thesis aims to document all the controller's infrastructure, as well as to provide a solid foundation for developing OpenDaylight applications.

## **Resum**

Les Xarxes Definides per Software (SDN) es presenten com un nou paradigma que té com a objectiu simplificar la creació i la gestió de les xarxes. Aquest tipus de xarxes estan basades en la separació del pla de control de la xarxa i del pla de 'forwarding', introduint un element lògic centralitzat que denominem controlador.

El projecte OpenDaylight (ODL), dirigit per Linux, defineix una implementació d'un controlador SDN, que ofereix una innovació respecte les altres solucions SDN del mercat: la Service Abstraction Layer (SAL). La SAL és una capa que permet oferir un alt nivell d'abstracció, ja que les aplicacions podran actuar sobre elements d'una xarxa sense preocupar-se dels diferents protocols de xarxa que hi poden haver.

El controlador OpenDaylight també ofereix un seguit de mòduls, que contenen plugins, que permeten la comunicació amb els protocols de xarxa, i aplicacions, que ens permeten obtenir informació sobre la xarxa (topologia, recursos, etc.).

Per aconseguir aquest nivell d'abstracció, OpenDaylight fa ús d'un seguit de tecnologies externes encarades a l'arquitectura dirigida per models. Aquest document pretén explicar detalladament totes les tecnologies que estan implicades en el desenvolupament d'una aplicació per OpenDaylight, juntament amb els seus mòduls interns que hi interactuen.

Al final del document, s'hauran detallat tots els passos necessaris per a desenvolupar una aplicació per a OpenDaylight des de dues perspectives:

- Interna: es detallaran tots els mòduls implicats en el procés intern que fa OpenDaylight des de que generem els models de la nostra aplicació fins que la despleguem per al seu ús.
- Desenvolupador: es detallaran tots els passos necessaris per a definir una aplicació: generar models que defineixin la nostra aplicació, donar les seves implementacions, identificar les seves dependències, com fer que la SAL reconegui la nostra aplicació, com fer el desplegament d'aquesta, etc.

Aquesta tesi pretén documentar la infraestructura del controlador i oferir una base sòlida per a començar a desenvolupar aplicacions per a OpenDaylight.

## **Resumen**

Las Redes Definidas por Software (SDN) se presentan como un nuevo paradigma con el objetivo de simplificar la creación y la gestión de las redes. Este tipo de redes están basadas en la separación del plano de control y el plano de enrutamiento, introduciendo un elemento lógico centralizado denominado controlador.

El proyecto OpenDaylight (ODL), liderado por Linux, define una implementación de un controlador SDN, que ofrece una innovación respecto a las otras soluciones SDN del mercado: la Service Abstraction Layer (SAL). La SAL es una capa que permite ofrecer un alto nivel de abstracción, ya que las aplicaciones pueden actuar sobre elementos de la red sin tener que preocuparse de los protocolos de red que pueda haber.

El controlador OpenDaylight también ofrece un conjunto de módulos, que contienen plugins, encargados de la comunicación con los protocolos de red) y aplicaciones, que nos permiten obtener información de la red (topología, recursos, etc...)

Para conseguir este nivel de abstracción, OpenDaylight usa un conjunto de tecnologías externas encaradas a la arquitectura dirigida por modelos. Este documento pretende explicar detalladamente todas las tecnologías implicadas en el desarrollo de una aplicación para OpenDaylight, juntamente con los módulos internos que interactúan.

Al final de este documento, se habrán detallado todos los pasos necesarios para desarrollar una aplicación para OpenDaylight usando dos perspectivas:

- Interna: se van a detallar todos los módulos implicados en el proceso interno que OpenDaylight hace desde que generamos los modelos de nuestra aplicación hasta que la desplegamos para su uso.
- Desarrollador: se detallaran todos los pasos necesarios para definir una aplicación: generar los modelos que definen nuestra aplicación, dar sus implementaciones, identificar sus dependencias, como hacer para que la SAL reconozca nuestra aplicación, como hacer el despliegue de ésta, etc.

Esta tesis pretende documentar la infraestructura del controlador y ofrecer una base sólida para empezar a desarrollar aplicaciones para OpenDaylight.

## **Acknowledgements**

I would like to thank my thesis director, Jose Luis Muñoz, for introducing me to the cloud computing technologies and guiding me through the process of this thesis.

I also want to thank my girlfriend for supporting and tolerating me in the stressful situations.

## Revision history and approval record

Revision	Date	Purpose
0	13/10/2015	Document creation
1	14/10/2015	Document revision 1
2	15/10/2015	Document delivery

## DOCUMENT DISTRIBUTION LIST

Name	e-mail
Bernat Ribes Garcia	bernat.ribes.garcia@gmail.com
Jose Luis Muñoz Tapia	jose.munoz@entel.upc.edu

Written by:		Reviewed and approved by:	
Date	13/10/2015	Date	15/10/2015
Name	Bernat Ribes Garcia	Name	Jose Luis Muñoz Tapias
Position	Project Author	Position	Project Supervisor

## **Table of contents**

The table of contents must be detailed. Each chapter and main section in the thesis must be listed in the “Table of Contents” and each must be given a page number for the location of a particular text.

Abstract .....	1
Resum .....	2
Resumen .....	3
Acknowledgements .....	4
Revision history and approval record.....	5
Table of contents .....	6
1.Introduction.....	7
1.1.Topic .....	7
1.2.Topic .....	7
2.State of the art of the technology used or applied in this thesis:.....	8
2.1.Topic .....	8
2.2.Topic .....	8
3.Methodology / project development: .....	9
4.Results .....	10
5.Budget .....	11
6.Environment Impact (Optional) .....	12
7.Conclusions and future development: .....	13
Bibliography:.....	14
Appendices (optional): .....	15
Glossary .....	16

## 1. Introduction

The goal of this document is to provide an executive summary of the project.

### 1.1. Objectives

The aim of this project is to document all the OpenDaylight controller's infrastructure, as well as to provide a solid foundation for developing OpenDaylight applications.

OpenDaylight is a huge controller that depends on a large number of third party technologies. Despite of having some official documentation and developer guides, it lacks of an organized documentation that explains the whole developing process, and how is this internally managed by the controller. This results into a big entrance barrier for newcomers, which present a lot of problems in understanding the whole infrastructure and, as a consequence, in developing applications.

Most of OpenDaylight documentation is specific for a concrete project, so it does not provide a general view of how the things are internally performed in the OpenDaylight and assumes the knowledge of different technologies. Furthermore, OpenDaylight is still a very young controller and as that it is constantly changing, complicating the documentation process since well-trained developers are more focused on developing.

OpenDaylight is aware of the entrance barrier that it presents for newcomers and has started to make some summits presenting low-level documentation. However, it is not enough for breaking that barrier.

This project provides a documentation that includes the whole developing process, introducing all the third party technologies that are involved and how they are integrated with OpenDaylight, as well as it provides a step by step vision that will drive audience to develop its first OpenDaylight application, knowing what they are doing at every moment and how the controller internally manages it.

The following technologies have been documented:

- OpenDaylight controller, with its internal core modules: MD-SAL and config subsystem.
- Model-driven technologies: YANG, NETCONF and RESTCONF.
- Project management tooling: Maven.
- Run-time environment: OSGi and Karaf.
- Java environment: JMX, JConsole, etc.

### 1.2. Requirements and specifications

The requirements of the project are based on conceptual and technical requirements. The conceptual requirements are those that are needed to start understanding how OpenDaylight works.

Technical specifications:



- OpenDaylight Lithium release.
- Eclipse IDE.
- Maven.
- Java 7+.
- LaTeX editor.

#### Conceptual specifications:

- Java programming language.
- Linux kernel-based OS knowledge.
- SDN basics.

### **1.3. Resources**

OpenDaylight software and the third-party technologies that requires are open source and are available in its official web page, referenced in the document. The project contains an installation and configuration guide for each.

### **1.4. Work plan**

#### Work Packages:

Project: OpenDaylight SDN controller platform	WP ref: 1	
Major constituent: Investigate about SDN networks	Sheet n of m	
Short description: In order to understand the whole concept of Software Defined Networks and how OpenDaylight can contribute in it.	Planned start date:	
	Planned end date:	
	Start event: End event:	
Internal task T1: Investigate about Software Defined Networks Internal task T2: Investigate about SDN architecture: controller	Deliverables:	Dates:

Project: OpenDaylight SDN controller platform	WP ref: 2	
Major constituent: Investigate OpenDaylight framework	Sheet n of m	
Short description:  In order to understand the whole OpenDaylight project, emphasizing in the controller itself. Determinate which third-party technologies are involved in its development.	Planned start date:	
	Planned end date:	
	Start event:  End event:	
Internal task T1: OpenDaylight overview  Internal task T2: OpenDaylight controller  Internal task T3: OpenDaylight project details	Deliverables:	Dates:

Project: OpenDaylight SDN controller platform	WP ref: 3	
Major constituent: MD-SAL and Config subsystem	Sheet n of m	
Short description:  In order to understand the core internal components of the OpenDaylight controller. Where abstractions of data, routing and application instantiation are performed.	Planned start date:	
	Planned end date:	
	Start event:  End event:	
Internal task T1: MD-SAL overview  Internal task T2: Config subsystem overview  Internal task T3: MD-SAL and config subsystem in the application developing process	Deliverables:	Dates:

Project: OpenDaylight SDN controller platform	WP ref: 4	
Major constituent: Maven research	Sheet n of m	
Short description:  In order to understand and introduce the project management tool that OpenDaylight uses. How to declare plugins and dependencies, etc.	Planned start date:	
	Planned end date:	
	Start event:  End event:	
Internal task T1: Maven overview  Internal task T2: OpenDaylight plugins & dependencies	Deliverables:	Dates:

Internal task T3: Maven in the application developing process		
---	--	--

Project: OpenDaylight SDN controller platform	WP ref: 5	
Major constituent: Investigate about YANG	Sheet n of m	
Short description: In order to understand and introduce the modelling language that OpenDaylight uses to model the functionality of applications (data, interactions, etc.)	Planned start date:	
	Planned end date:	
	Start event:	
	End event:	
Internal task T1: YANG overview	Deliverables:	Dates:
Internal task T2: YANG to Java mapping		
Internal task T3: YANG in the application developing process		

Project: OpenDaylight SDN controller platform	WP ref: 6	
Major constituent: Investigate about OSGi and Karaf	Sheet n of m	
Short description: In order to understand and introduce the modular run-time framework in which the OpenDaylight controller operates. How are applications deployed, etc.	Planned start date:	
	Planned end date:	
	Start event:	
	End event:	
Internal task T1: OSGi overview	Deliverables:	Dates:
Internal task T2: Karaf overview		
Internal task T3: OSGi and Karaf in ODL framework		

Project: OpenDaylight SDN controller platform	WP ref: 7	
Major constituent: Document the application development	Sheet n of m	
Short description: In order to provide the whole developing process, step-by-step and how is this internally managed by OpenDaylight.	Planned start date:	
	Planned end date:	
	Start event:	
	End event:	

Internal task T1: Developer perspective	Deliverables:	Dates:
Internal task T2: Internal perspective		
Internal task T3: Testing results		

Project: OpenDaylight SDN controller platform	WP ref: 8
Major constituent: Thesis documentation	Sheet n of m
Short description: Documentation of the thesis	Planned start date: Planned end date: Start event: End event:

	March				April				May				June				July				August				September				October	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Investigate about SDN networks																														
Investigate OpenDaylight framework																														
MD-SAL and Config subsystem																														
Maven research																														
Investigate about YANG																														
Investigate about OSGi and Karaf																														
Documentate application development																														
Thesis documentation																														

### 1.5. Project deviations

The initial objective of this project was not to document the whole controller. In fact, we thought that it was smaller than what it really is, so we intended to give its basics and to develop a network-related application to show off its potential.

However, when starting, we figured out that it was really hard to get in and that many things were not clear due to the quality of the basic documentation. So we decided to change the scope of this project, and re-define it as a foundation and initial contribution from which audience can start developing network-related applications.

Before taking this decision, I got in touch with active members OpenDaylight members and they also were aware of that issue, so they encouraged me into making a detailed documentation to raise people awareness and, specially, take off that entrance barriers.

## 2. State of the art of the technology used or applied in this thesis:

### Software Defined Networks

The main principle behind Software-Defined Networking (SDN), is the physical separation of the network control plane from the forwarding plane, by adding a single control plane that controls several devices. This allows to centrally manage the intelligence and the state of the network, and to abstract the complexity of the underlying physical network.

### OpenDaylight

ODL is a highly available, extensible, scalable and multi-protocol controller infrastructure built for SDN deployments. It provides a model-driven service abstraction (service abstraction layer) that allows users to develop applications that easily work across a wide variety of hardware and southbound protocols.

It relies on the following technologies:

- **Maven:** project management tool that simplifies and automates dependencies between a project or different projects. This tooling will help developers to manage all the required plugins and dependencies, as well as to provide a project start-up using its defined archetypes.
- **Java:** it is the programming language that is used to develop applications and features in the OpenDaylight's controller. Developing in Java provides a valuable compile-time safety, as well as an easy way to implement defined services.
- **Open Service Gateway Interface (OSGi):** is the back-end of OpenDaylight as it allows to dynamically load bundles and JAR packages (they compose the applications), and bind modules together for exchanging information.
- **Karaf:** it is an application container built on top of OSGi, which simplifies aspects of packaging and installing applications.
- **YANG:** it is the key-point of the model-driven behaviour in the controller. Developers will use YANG to model an application functionality, and to generate APIs from the defined models, which will be later used to provide its implementations. YANG supports modelling operational and configuration data, as well as RPC and notifications.

### Model-Driven Software / Protocols

They leverage the Model-Driven Software Engineering, which describes a framework based on consistent relationships between (different) models, standardized mappings and patterns that enable model generation and, by extension, code/API generation from models:

- **NETCONF:** is an IETF network management protocol that defines configuration and operational conceptual data stores and a set of Create, Retrieve, Update, Delete (CRUD) operations that can be used to access to them. In addition to the CRUD operations, NETCONF also supports simple Remote Procedure Calls (RPCs) and notification

operations. NETCONF uses XML - base data encoding for the configuration and operational data, as well as for its protocol messages.

- **YANG:** is a modelling language that allows to model configuration and state data in network devices. It also can be used to describe other network constructs, such as services, policies, protocols or subscribers. YANG is tree-structured rather than object-oriented; data is structured into a tree and it can contain complex types. YANG supports constructs to model Remote Procedure Calls and notifications, which makes it suitable for use as an Interface Description Language (IDL) in a model-driven system.

- **RESTCONF:** is a REST-like protocol that provides a programmatic interface over HTTP for accessing data defined in YANG, using the data stores defined in NETCONF. Configuration data and state data are exposed as resources that can be retrieved with the HTTP GET method, while resources representing configuration data can be modified with the HTTP DELETE, PATCH, POST and PUT methods. Data is encoded in either XML or JSON.

### **3. Methodology / project development:**

One of the most main objectives of the project has been the documentation of the core modules of OpenDaylight controller and all the third-party technologies in which ODL mainly relies on. To do so, it has been used the following method.

- Investigate about the module.
- Identify its required third-party technologies.
- Research about the third-party technologies.
- Document the module as well as its impact in the developing process
- Overview of the third-party technologies involved on that particular module.

The other main objective of the project has been to provide a developing guide, detailing step-by-step the required process to develop an OpenDaylight application. To do so, as different modules and third-party technologies are involved, it has been used the following method:

- Research about the interactions between the core OpenDaylight modules.
- Integration with the third-party technologies.
- Starting an OpenDaylight project from 0 and testing the interactions, as well as the obtained results.
- Document the whole process providing two perspectives: developer perspective (code, required dependencies, plugins, etc.) and internal (how our code is being internally interpreted by core modules).

## 4. Results

The result is a documentation that presents OpenDaylight as an SDN solution and details a whole application developing process, introducing all the third party technologies that are involved and how they are integrated with ODL, as well as it provides a step by step vision that drives audience to develop its first application, knowing what they are doing at every moment and how the controller internally manages it.



## 5. Budget

All the software that is used in this project is licensed under a Free Software License (OpenDaylight, Maven, Eclipse...), so it did not add any cost to the thesis.

The operating system used for the development is Ubuntu, which is a free GNU/Linux distribution, so it also does not add any cost to the thesis

The estimated number of hours dedicated to this thesis is 360h. Taking in account that the real cost of a recently graduated engineer, which is 12.5 €/h.

The total cost of the thesis is **4500€**.

## **6. Conclusions and future development:**

### **6.1 Conclusions**

OpenDaylight controller has proved that it is the most successful open-source SDN controller, thanks to the successful collaborations among member companies.

OpenDaylight can be defined by using the following terms: “open”, “multi-vendor”, “multi-project” and “innovation”. As it is able to support network programmability (supports multiple network protocols) via its southbound plugins, a bunch of programmable services.

Furthermore, OpenDaylight presents the Service Abstraction Layer as its key design, which enables the abstraction of services between consumers and providers. In other words, applications that use the controller to communicate with a network, do not have to be aware of which networking protocol is implementing it.

This briefly summarizes the potential that OpenDaylight has. In fact, from its first release in 2014, the number of projects have been growth from 2 to more than 40.

However, at the moment, OpenDaylight does not raise the awareness that a controller with its features deserves. This is essentially because the big entrance barrier that newcomers find when trying to get involved, making it an important drawback for an open community as ODL is.

### **6.2 Future developments**

As a future development for the document created in this thesis, which intend to be the foundation for understanding OpenDaylight capabilities and starting to develop applications, could be the design of any OpenDaylight application that provide services to the network, or a plugin to communicate with specific network protocols. Things that can be developed using OpenDaylight developing framework are unlimited.

For this reason, important cloud computing technologies such as OpenStack, or SDN protocols such as OpenFlow, have shown a strong commitment to integrate with OpenDaylight. An example is that, the latest release, Lithium, has been focused on optimizing the integration with those big technologies.

OpenDaylight is nowadays, the leading SDN solution in the market. As SDN are becoming stronger and are considered as the future, mastering ODL becomes a priceless value.

## **Bibliography:**

- [1] Software defined networking definition.  
<https://www.opennetworking.org/sdn/resources/sdn-definition>.
- [2] The road to SDN  
<https://www.cs.princeton.edu/courses/archive/fall13/cos597E/papers/sdnhistory.pdf>.
- [3] OpenDaylight: MD-SAL architecture.  
[https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:MD-SAL:MD-SAL\\_Document\\_Review:Architecture](https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:MD-SAL_Document_Review:Architecture).
- [4] OpenDaylight: Membership. <https://www.opendaylight.org/membership>.
- [5] OpenDaylight: releases. <https://www.opendaylight.org/software/release-archives>.
- [6] OpenDaylight: TSC charter. <https://www.opendaylight.org/tsc-charter>.
- [7] OpenDaylight: Lithium. <https://www.opendaylight.org/lithium>.
- [8] OpenDaylight: Developer guide.  
<https://www.opendaylight.org/sites/opendaylight/files/bk-developers-guide-20150831.pdf>.
- [9] AD-SAL vs MD-SAL. <http://sdntutorials.com/difference-between-ad-sal-and-md-sal/>.
- [10] OpenDaylight: Project life-cycle. <https://www.opendaylight.org/project-lifecycle-releases>.
- [11] OpenDaylight: The Open Source SDN controller.  
<https://clnv.s3.amazonaws.com/2015/usa/pdf/BRKSDN-2761.pdf>.
- [12] OpenDaylight: Download. <http://www.opendaylight.org/downloads>.
- [13] Official OpenDaylight Git repository. <https://git.opendaylight.org/gerrit/>.
- [14] YANGTools project. [https://wiki.opendaylight.org/view/YANG\\_Tools:Main](https://wiki.opendaylight.org/view/YANG_Tools:Main).
- [15] What is the document object model? <http://www.w3.org/TR/DOM-Level-3-Core/introduction.html>.
- [16] XML DOM tutorial. [http://www.w3schools.com/xml/dom\\_intro.asp](http://www.w3schools.com/xml/dom_intro.asp).
- [17] OpenDaylight: Yang schema and model.  
[https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:YANG\\_Schema\\_and\\_Model](https://wiki.opendaylight.org/view/OpenDaylight_Controller:YANG_Schema_and_Model).
- [18] OpenDaylight: Binding model.  
[https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:Binding\\_Model](https://wiki.opendaylight.org/view/OpenDaylight_Controller:Binding_Model).
- [19] OpenDaylight: Binding-aware components.  
[https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:Binding\\_Aware\\_Components](https://wiki.opendaylight.org/view/OpenDaylight_Controller:Binding_Aware_Components).
- [20] OpenDaylight: Config subsystem.  
[https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:Config:Main](https://wiki.opendaylight.org/view/OpenDaylight_Controller:Config:Main).

[21] Notes on the Config subsystem.

<http://sdntutorials.com/notes-on-config-subsystem-in-opendaylight/>.

[22] OSGi alliance. <http://www.osgi.org/Main/HomePage/>.

[23] OSGi: An overview of its impact on the software life-cycle. <http://java.sys-con.com/node/1765471>.

[24] Apache Karaf. <http://karaf.apache.org/>.

[25] Apache Maven project. <https://maven.apache.org/>.

[26] Maven: The complete reference. <http://books.sonatype.com/mvnref-book/reference/>.

[27] Downloading Maven. <http://maven.apache.org/download.cgi>.

[28] Downloading Eclipse. <https://www.eclipse.org/downloads/>.

[29] Maven software for Eclipse. <http://www.eclipse.org/m2e/>.

[30] Maven tutorial. <http://www.tutorialspoint.com/maven/>.

[31] YANGTools user guide.  
[https://wiki.opendaylight.org/view/YANG\\_Tools:User\\_Guide](https://wiki.opendaylight.org/view/YANG_Tools:User_Guide).

[32] OpenDaylight start-up project archetype.

[https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:MD-SAL:Startup\\_Project\\_Archetype](https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Startup_Project_Archetype).

[33] M. Bjorklund. YANG - a data modeling language for the network configuration protocol (NETCONF), October 2010. RFC6020.

[34] OpenDaylight: YANG to Java mapping.  
[https://wiki.opendaylight.org/view/YANG\\_Tools:YANG\\_to\\_Java\\_Mapping](https://wiki.opendaylight.org/view/YANG_Tools:YANG_to_Java_Mapping).

[35] YANGTools Git repository. <https://github.com/opendaylight/yangtools>.

[36] OpenDaylight: Toaster example overview.  
[https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:MD-SAL:Toaster\\_Tutorial](https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Toaster_Tutorial).

[37] OpenDaylight: Toaster example step-by-step.  
[https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:MD-SAL:Toaster\\_Step-By-Step](https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Toaster_Step-By-Step).

[38] Toaster Git repository.  
<https://github.com/opendaylight/coretutorials/tree/master/toaster>.

Bernat Ribes  
Jose L. Muñoz

UPC Telematics Department

---

OpenDaylight SDN controller platform



# Contents

<b>1</b>	<b>Software Defined Networks</b>	<b>9</b>
1.1	Introduction to Software-Defined Networking . . . . .	9
1.1.1	Motivation . . . . .	9
1.1.2	History and development . . . . .	10
1.1.3	Architecture . . . . .	11
1.2	Model-Driven Software Engineering . . . . .	12
1.3	Model-Driven Network management / Programmability . . . . .	13
<b>2</b>	<b>OpenDaylight</b>	<b>15</b>
2.1	Introduction . . . . .	15
2.2	Architecture . . . . .	16
2.3	Controller . . . . .	17
2.4	Service Abstraction Layer . . . . .	17
	Evolution to Model-Driven Service Adaptation Layer . . . . .	18
2.5	OpenDaylight projects . . . . .	20
2.6	Configuration and installation . . . . .	25
2.6.1	System requirements . . . . .	26
2.6.2	Downloading and installing the Java JDK . . . . .	26
2.6.3	Installing Git . . . . .	26
2.6.4	Running the Lithium distribution . . . . .	26
2.6.5	Installing Karaf features . . . . .	26
2.6.6	DLUX . . . . .	27
<b>3</b>	<b>MDSAL</b>	<b>29</b>
3.1	Introduction . . . . .	29

3.2	MD-SAL format and API types: . . . . .	30
3.2.1	DOM format and APIs . . . . .	30
3.2.2	Binding format and APIs . . . . .	33
3.2.3	RESTCONF interface . . . . .	34
3.3	Basic concepts . . . . .	34
3.4	MD-SAL design . . . . .	35
3.5	Messaging patterns . . . . .	36
3.6	MD-SAL data transactions . . . . .	37
3.6.1	Transaction types . . . . .	37
3.6.2	Write-only and read-write transactions . . . . .	37
	Transaction operations . . . . .	39
	Submitting transactions . . . . .	40
	Transaction local state and isolation . . . . .	41
	Commit failure scenarios . . . . .	42
3.7	MD-SAL RPC routing . . . . .	43
3.7.1	Sending a RPC message . . . . .	44
3.7.2	Global RPCs . . . . .	45
	Global RPC implementation . . . . .	45
3.7.3	Routed RPCs . . . . .	46
	Modelling routed RPCs . . . . .	46
	Implementing routed RPCs . . . . .	47
3.8	MD-SAL notifications routing . . . . .	48
3.8.1	Publishing a notification . . . . .	49
3.8.2	Subscribing for notifications . . . . .	49
3.9	RESTCONF requests . . . . .	50
3.9.1	Supported operations . . . . .	50
<b>4</b>	<b>Config subsystem</b>	<b>53</b>
4.1	Overview . . . . .	53
4.2	Configuration process . . . . .	54
4.2.1	Modules and services . . . . .	54
4.2.2	Declaring an application aware of the config subsystem . . . . .	55
	Providing state data . . . . .	56



4.2.3	Config subsystem models discovery . . . . .	57
4.2.4	Application instantiation . . . . .	58
4.2.5	Wiring a YANG schema to OpenDaylight . . . . .	59
4.2.6	Application reconfiguration . . . . .	60
4.2.7	Pushing initial configuration . . . . .	60
	Using property file . . . . .	61
	Using the config-persister . . . . .	61
4.2.8	Customizing initial configuration . . . . .	62
4.3	Config subsystem APIs and SPIs definitions . . . . .	64
4.3.1	SPIs . . . . .	64
4.3.2	APIs . . . . .	64
4.3.3	Run-time APIs . . . . .	65
4.3.4	JMX APIs . . . . .	65
<b>5</b>	<b>OSGi and Karaf</b>	<b>67</b>
5.1	Open Services Gateway Interface . . . . .	67
5.1.1	Overview . . . . .	67
5.1.2	Service model . . . . .	69
5.1.3	Deployment life-cycle . . . . .	69
5.2	Karaf . . . . .	70
5.2.1	Overview . . . . .	70
5.2.2	Definition of a Karaf feature . . . . .	71
5.2.3	Karaf shell console . . . . .	73
<b>6</b>	<b>Maven</b>	<b>75</b>
6.1	Introduction . . . . .	75
6.2	Installation and configuration . . . . .	76
6.2.1	Maven installation . . . . .	76
6.2.2	Maven integration . . . . .	77
6.3	Introduction to the POM . . . . .	77
6.4	Maven life-cycle: . . . . .	79
6.4.1	Clean lifecycle . . . . .	80
6.4.2	Default life-cycle . . . . .	80

6.4.3	Site life-cycle . . . . .	80
6.5	Repositories . . . . .	81
6.6	Dependencies . . . . .	82
6.6.1	OpenDaylight common dependencies . . . . .	82
6.6.2	Dependency search sequence . . . . .	83
6.7	Plugins . . . . .	84
6.7.1	YANG interpreter plugin . . . . .	85
6.7.2	Maven bundle plugin . . . . .	86
6.8	Creating a project . . . . .	87
6.8.1	OpenDaylight archetype . . . . .	87
<b>7</b>	<b>YANG</b>	<b>91</b>
7.1	Introduction . . . . .	91
7.2	Language overview . . . . .	92
7.2.1	Modules and submodules . . . . .	92
7.2.2	Data modelling basics . . . . .	94
7.2.3	State and configurational data . . . . .	94
7.2.4	Built-in types . . . . .	95
7.2.5	Derived types . . . . .	95
7.2.6	Grouping of reusable nodes . . . . .	96
7.2.7	Choices . . . . .	97
7.2.8	Extending a data model (augmentations) . . . . .	98
7.2.9	Identities . . . . .	99
7.3	Basic data modelling statements . . . . .	99
7.3.1	Leaf nodes . . . . .	99
7.3.2	Leaf-list nodes . . . . .	100
7.3.3	Container nodes . . . . .	101
7.3.4	List nodes . . . . .	103
7.4	RPCs and notifications . . . . .	104
7.4.1	RPC definitions . . . . .	104
7.4.2	Notification definitions . . . . .	105
7.5	YANG to Java mapping . . . . .	106
7.5.1	Name mapping rules . . . . .	106

Package name . . . . .	106
Class and interface names . . . . .	108
Getter and setter names . . . . .	108
7.5.2 Code generation . . . . .	108
Module . . . . .	108
Container . . . . .	110
Leaf . . . . .	111
Leaf-list . . . . .	112
List . . . . .	112
Choice and case . . . . .	114
Grouping and uses . . . . .	115
Identities . . . . .	116
RPCs . . . . .	117
Augmentations . . . . .	118
Notifications . . . . .	119
7.6 OpenDaylight YANG models . . . . .	120
<b>8 Developing an OpenDaylight application</b>	<b>121</b>
8.1 Introduction . . . . .	121
8.1.1 Prerequisites and project structure . . . . .	122
8.2 Part 1: Defining an operational toaster . . . . .	124
8.2.1 Defining a toaster data model . . . . .	124
8.2.2 Implementing the toaster operational data . . . . .	128
8.2.3 Wiring the ToasterImpl service . . . . .	131
8.2.4 Implementing the ToasterImplModule . . . . .	135
8.2.5 Defining the initial XML configuration file . . . . .	136
8.2.6 Getting the operational status of the toaster . . . . .	137
8.3 Part 2: Enabling Remote Procedure Calls (RPCs) . . . . .	139
8.3.1 Defining YANG RPC statements . . . . .	139
8.3.2 Implementing RPC methods . . . . .	140
8.3.3 Invoking RPCs via RESTCONF . . . . .	145
8.4 Part 3: Adding configuration data . . . . .	146
8.4.1 Adding configuration data in the toaster data model . . . . .	146

8.4.2	Changing the configuration data . . . . .	149
8.5	Part 4: Adding state data (JMX access) . . . . .	150
8.5.1	Defining state data models . . . . .	150
8.5.2	Implementing the state data model . . . . .	152
8.5.3	Registering the ToasterImplRuntimeMXBean service . . . . .	153
8.5.4	Accessing state data via JMX . . . . .	153
8.6	Part 5: Adding a ToasterService consumer . . . . .	155
8.6.1	Defining the KitchenService interface . . . . .	155
8.6.2	Defining the KitchenServiceImpl implementation . . . . .	156
8.6.3	Wiring the KitchenServiceImpl . . . . .	157
8.6.4	Defining the initial XML configuration file . . . . .	159
8.6.5	Adding a JMX RPC . . . . .	160
8.6.6	Invoking RPCs via JMX . . . . .	162
8.7	Part 6: Defining notifications . . . . .	163
8.7.1	Defining YANG notification statements . . . . .	163
8.7.2	Publishing notifications . . . . .	164
8.7.3	Wiring our ToasterImpl for notifications . . . . .	166
8.7.4	Implementing notifications in the KitchenServiceImpl . . . . .	166
8.7.5	Wiring the KitchenService for notifications . . . . .	167
8.7.6	Testing notifications . . . . .	168

# Chapter 1

## Software Defined Networks

### 1.1 Introduction to Software-Defined Networking

The main principle behind Software-Defined Networking (SDN) is the physical separation of the network control plane from the forwarding plane, by adding a single control plane to manage all the devices. This allows to centrally manage the intelligence and the state of the network, and to abstract the complexity of the underlying physical network [1].

SDN offers several benefits over traditional networks. On one hand, network elements can be simplified without implying performance losses. On the other hand, SDN improves the flexibility and the performance of the network tasks thanks to its centralization in a single logical element, the controller.

Software-Defined Networking is an emerging architecture that is dynamic, manageable, cost-effective and adaptable; making it ideal for the high-bandwidth and dynamic nature of today's applications. SDN major features:

- **Programmable networks:** the disengagement between control and forwarding planes allow a direct programming of the network without caring about forwarding functions.
- **Flexible development:** that disengagement also allows a fast and adaptable modification of flows and network functions.
- **Centralized management:** SDN architecture is based on a centralized controller that keeps a global vision of the network, and also interacts with it by adding, deleting or modifying flows.
- **Allows to automate the device programming:** SDN allows administrators to configure self-written programs (as SDN does not belong to any software) that speed up the procedures of configuration, optimization and administration of network resources.
- **Based on open standards:** it allows the simplification of the network management as it does not depend on any specific providers, adaptation devices or protocols.

#### 1.1.1 Motivation

SDN addresses the fact that the static architecture of conventional networks is not suitable for the dynamic computing capabilities and the storage needs that we have today. Some of the things that make networks drive to the need for a new paradigm are:

- **Changing traffic patterns:** applications that commonly access to geographically distributed databases and servers, through public and private clouds, require extremely flexible traffic management and access to the bandwidth on demand.
- **The 'consumerization of IT':** the Bring Your Own Device (BYOD) trend requires networks that are both flexible and secure.
- **The rise of cloud services:** users expect on-demand access to applications, infrastructure and other IT resources.
- **"Big data" means more bandwidth:** handling today's mega datasets requires massive parallel processing that is fueling a constant demand for additional capacity and any-to-any connectivity.

Limitations of current traditional networks:

- **Complexity that leads to stasis:** adding or moving devices and implementing network-wide policies are complex, time-consuming, and primarily manual endeavors that risk service disruption. This complexity, discourage networking changes.
- **Inability to scale:** scaling capabilities of traditional networks are not effective with the dynamic traffic patterns in virtualized networks. This problem is even more pronounced in service provider networks, which require a large-scale of parallel processing algorithms and its associated datasets across an entire computing pool.
- **Vendor dependence:** lengthy vendor equipment produce cycles and the lack of open standards and interfaces, limiting the ability of network operators to tailor networks to their individual environments.

SDN benefits:

- Does not depend on the slow innovation and development of hardware, allowing it to be innovated at software's speed, especially in the control plane.
- Optimizes the cost of networking infrastructure.
- Implements the control function in software.
- Simplifies the network equipment.
- Allows the usage of a more complex and costly network algorithms.
- Reduces entrance barriers to innovators and researchers.
- Promotes the innovation of routing algorithms and a better usage of network capabilities.

These benefits have raised the interest of big companies, which are already implementing their own SDN-based solutions. For example, Orange is already making use of SDN, specifically it uses the implementation of the controller that is presented in this document, OpenDaylight.

### 1.1.2 History and development

SDN networks represent a relatively new concept that is still under development. However, it has been matured for years [2].

In the 90's Internet emerged to big audience. This unleashed an effort to improve the network infrastructure in order to manage it in a more efficient way, with the idea of being programmable, and being capable to commute according to the needs in a specific moment.

Internet deployment also caused a huge development of applications, which turned into the need of designing new network protocols to support them. Each protocol had to be standardized by the Internet Engineering Task Force (IETF), making it a slow process for lots of researchers.

As an alternative, some research groups bet for an opening perspective of network control, allowing administrators to reprogram networks. However, conventional networks were not programmable, so active networks were created. Active networks were based in a programmable interface with the capability of showing resources on a particular node and applying concrete actions to a group of defined packets (flows). This kind of network meant a radical approximation to network control. While the development of active networks did not hugely expand, it had resulted into technologies that are being used in SDN, such as the programmatic functions of network elements, network virtualization, etc.

At the beginning of the new millennium, a new concept appeared: the separation of the control plane (where network control functions are located) and the data plane (packet forward functions). Internet traffic had growth and the need of having predictable and reliable networks carried researchers to find new approaches for the network's management. Switchers and routers had to deal with tasks like analyzing and solving configuration problems, or controlling the behaviour of a packet, being these too complex tasks for the limitations that they had. For this reason, both planes were separated, turning in a set of new network standards and solutions.

### 1.1.3 Architecture

In a SDN implementation, everything is based in the central element, the controller. The controller is the element that will communicate all the layers by exposing APIs (Application Programming Interface). To split the control and data planes, two kind of APIs are required: northbound APIs (communicates with application layer) and southbound APIs (that normally communicates with the network devices).

As figure 1.1 illustrates, applications communicate with the controller through the northbound APIs. In the same way, all the instructions from controller to network devices are routed through the southbound APIs. Furthermore, southbound APIs provide a layer of abstraction, making network device type's indifferent for the controller and applications.

- **Controller:** it is the core of the network. Placed in the middle layer, it is the framework in which the SDN abstractions can manifest. It provides a set of common APIs to the application layer (northbound APIs), while implements one or more network protocols for controlling network devices (southbound interface). SDN does not only support SDN-oriented networking protocols, in fact, it also supports traditional networking protocols like: Open Shortest Path First (OSPF), MultiProtocol Label Switching (MLPS) or Border Gateway Protocol (BGP).

The controller may also include a set of modules that allow him to carry out a set of basic networking tasks: inventory of the connected devices, statistics management and other functions. Providers can add new functionalities in the controller's core according to their needs, being this one of the key points of the SDN architecture.

- **Southbound APIs:** they make possible the communication between the controller and network elements. They allow making dynamic changes to the network elements in order to adapt them to the requirements at real time. Moreover, they also allow the independence between the underlying elements of the controller. This is because the controller only has to be aware of sending instructions, that adaptation APIs will translate and route to the appropriate network element.
- **Northbound APIs:** they are used to communicate the controller with applications that are running over the network. They facilitate innovation and enable efficient orchestration and automation of the network. Northbound APIs are the most critical APIs in the SDN environment, since they have to support a wide variety of applications.

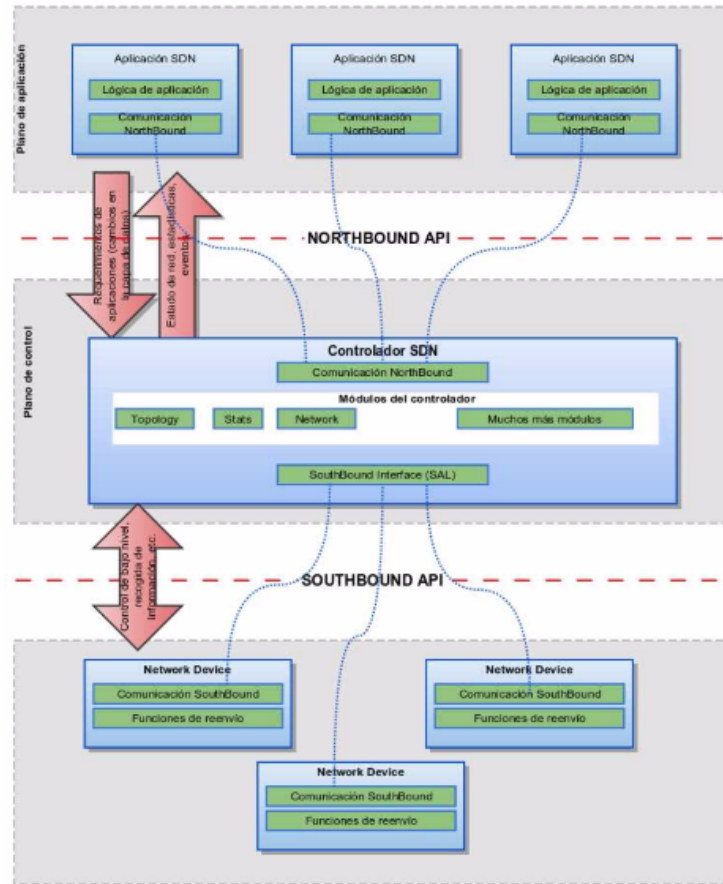


Figure 1.1: SDN architecture

## 1.2 Model-Driven Software Engineering

OpenDaylight leverages the Model Driven Software Engineering (MDSE), which is defined by the Object Management Group (OMG). MDSE describes a framework based on consistent relationships between (different) models, standardized mappings and patterns that enable model generation and, by extension, code/API generation from models [3].

The MDSE approach is meant to increase productivity by maximizing the compatibility between systems (via reuse of standardized models), simplifying the process of design (via models of recurring design patterns in the application domain), and promoting the communication between individuals and teams working on the system (via a standardization of the terminology and the best practices used in the application domain).

This generalization can overlay any specific modelling language. Although OMG focus their model driven adaptation solution on UML, YANG has emerged as the data modelling language for the networking domain.



## 1.3 Model-Driven Network management / Programmability

The model-driven approach is being increasingly used in the networking domain to describe the functionality of network services, policies and APIs. In OpenDaylight's framework, the protocols of choice are NETCONF and RESTCONF; and the modelling language is YANG [3].

- **NETCONF:** is an IETF network management protocol that defines configuration and operational conceptual data stores and a set of Create, Retrieve, Update, Delete (CRUD) operations that can be used to access to them. In addition to the CRUD operations, NETCONF also supports simple Remote Procedure Calls (RPCs) and notification operations. NETCONF uses XML, which is the base data encoding for configuration and operational data, as well as for its protocol messages.
- **YANG:** is a modelling language that allows to model configuration and state data in network devices. It also can be used to describe other network constructs, such as services, policies, protocols or subscribers. YANG is a data-oriented language that structures data as a tree, and can contain complex types. YANG supports constructs to model Remote Procedure Calls and notifications, which makes it suitable to use as an Interface Description Language (IDL) in a model-driven system.
- **RESTCONF:** is a REST-like protocol that provides a programmatic interface over HTTP for accessing data defined in YANG, using the data stores defined in NETCONF. Configuration and state data are exposed as resources that can be retrieved with the HTTP GET method, while resources representing configuration data can be modified with the HTTP DELETE, PATCH, POST and PUT methods. Data is encoded in either XML or JSON.



## Chapter 2

# OpenDaylight

### 2.1 Introduction

OpenDaylight (ODL) appeared with the objective to accelerate SDN adoption, and to create a solid base for the virtualization functions, collaborating with the biggest companies in the networking sector.

In April 2013, Linux Foundation announced OpenDaylight as an open project, with the objective of promoting innovation and creating an open and transparent perspective. Founder members were important companies such as Cisco, Citrix, HP, IBM, Juniper Networks, Microsoft, Red Hat and VMWare [4].

First release (Hydrogen) was launched in February 2014, which included an open code controller, virtualization capabilities and some protocol plugins. In November 2014, a new release was launched (Helium), which promoted OpenDaylight to a more suitable work environment based on Karaf and model-driven network management. The actual release is Lithium, launched at 29th June of 2015, which introduces clustering and defines plugins to support new network protocols. Also, improves the integration support with other technologies (e.g OpenStack Neutron) and a defines new OpenFlow plugin (now based on model-driven network management and supports OpenFlow versions 1.0/1.3). OpenDaylight release archives, containing release notes and downloads, can be found at [5].

Every change in the OpenDaylight project has to be voted by the technical steering committee. The committee is limited to one vote for each project member. Platinum members always have a vote independently of its collaboration in the project. However, not every member has a vote; for those that are not platinum, only the ones that are important or are directly cooperating with OpenDaylight will be considered to vote. By doing this, OpenDaylight ensures its independence from the different hardware companies [6].

As it can be seen in the figure 2.1, OpenDaylight platinum members are composed by some of the most important networking companies.



Figure 2.1: OpenDaylight Platinum members

## 2.2 Architecture

ODL is a highly available, extensible, scalable and multi-protocol controller infrastructure built for SDN deployments. It provides a model-driven service abstraction (service abstraction layer) that allows users to develop applications that easily work across a wide variety of hardware and southbound protocols.

Furthermore, it contains internal plugins that add services and functionalities to the network. For example, it have dynamic plugins that allow to gather statistics as well as to obtain the topology of the network.

Figure 2.2 shows the architecture framework of the latest OpenDaylight release, Lithium [7].

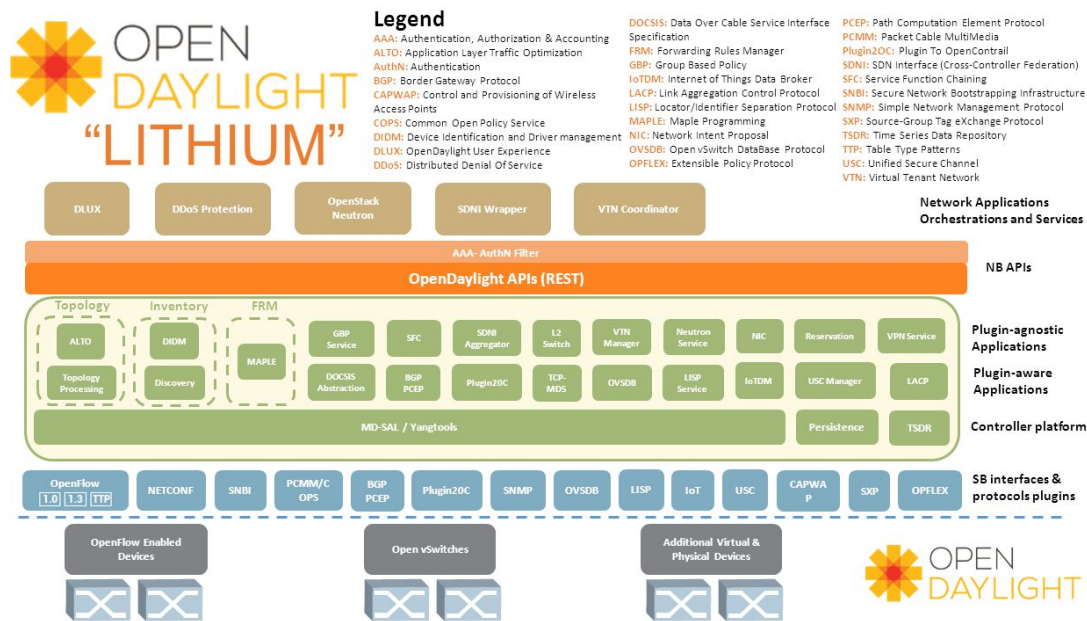


Figure 2.2: Lithium architecture framework

As an SDN implementation, OpenDaylight also implements the common three-layered architecture:

- **Network applications and orchestration:** in this layer we can find applications that compute network traffic engineering (control and monitor the controller) or solutions that use virtualization services. DLUX (ODL graphical user interface) and OpenStack Neutron are examples of network applications.
- **Controller:** is the central layer and where the SDN abstractions are manifested. The controller also contains implemented modules that allow network applications to obtain data and information about the state of the network, and multiple plugins to permit the communication with different network protocols (OpenFlow, BGP, NETCONF, OVSD).

Furthermore, it has different APIs that allow application development (DOM , REST or Java).

- **Network elements:** can be physical or virtual, they are programmable by the implemented protocols in the controller. Thanks to the service abstraction layer of the controller, OpenDaylight is compatible with most of network elements (independently of its providers).

## 2.3 Controller

ODL controller is strictly implemented in software and is contained within its own Java Virtual Machine (JVM). As such, it can be deployed on any hardware and operating system platform that supports Java [8]. OpenDaylight controller relies on the following technologies:

- **Maven:** is a project management tool that simplifies and automates dependencies between a project or different projects. This tooling will help developers to manage all the required plugins and dependencies of its applications, as well as to provide a project start-up by using its defined archetypes.
- **Java:** it is the programming language that is used to develop applications and features in the OpenDaylight's controller. Developing in Java provides a valuable compile-time safety, as well as an easy way to implement defined services.
- **Open Service Gateway Interface (OSGi):** is the back-end of OpenDaylight as it allows to dynamically load bundles and JAR packages (they compose the applications), and bind modules together for exchanging information.
- **Karaf:** it is an application container built on top of OSGi, which simplifies aspects of packaging and installing applications.
- **YANG:** it is the key-point of the model-driven behaviour in the controller. Developers will use YANG to model an application functionality, and to generate APIs from the defined models, which will be later used to provide its implementations. YANG supports modelling operational and configuration data, as well as RPC and notifications.

OpenDaylight also relies on the model-driven protocols, NETCONF and RESTCONF, which were introduced in the section 1.3. NETCONF is used to define the data stores and its supported interactions with applications (transactions, operations, etc.), while RESTCONF is used to handle its external access.

OpenDaylight provides the following model-driven subsystems as the foundation for Java applications:

- **Model-Driven Service Abstraction Layer (MD-SAL):** it defines messaging and storage functionality for data, notifications and RPCs modelled by application developers. MD-SAL uses YANG as the modelling language for service and data definitions.
- **Config subsystem:** it is an activation, dependency-injection and configuration framework. It handles initial configuration of the controller, new configuration commits, dependency management and run-time wiring. Developers will use the config subsystem to instantiate applications by exposing its initial configuration (its required dependencies and modules) to ensure that they are already active in the controller, so the application can be successfully instantiated.
- **MD-SAL clustering:** enables clustering support for core MD-SAL functionality and provides location-transparent access to YANG modelled data. As it is a new functionality in Lithium and still lacks of stable implementation, it will not be explained in this document.

## 2.4 Service Abstraction Layer

The initial bootstrap contribution to OpenDaylight offered a major innovation in the SDN framework: the Service Abstraction Layer (SAL), which separates southbound protocol plugins (SB) from northbound service/application plugins (NB).

SAL can be seen as a layer against which applications can be developed without assuming the underlying SDN protocols that it has to communicate. It adapts southbound plugin functions to higher-level application/service functions, which are provided by controller northbound APIs and available to external applications. The layered architecture allows the controller to support multiple southbound plugins, and an uniform set of services and APIs through a common set of northbound APIs.

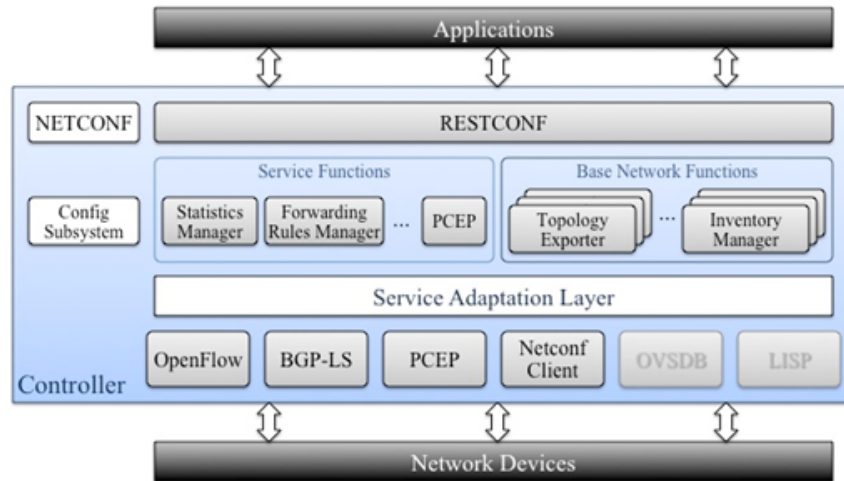


Figure 2.3: Service Abstraction Layer

However, in the original SAL (referenced as “API-Driven SAL”, AD-SAL) developers had to manually define which APIs were used by a specific plugin, and to hand-code the adaption functionality between southbound and northbound plugins. It quickly became apparent that hand-coding the SAL APIs and adaptations to support new plugin functionality would not scale in an initiative of OpenDaylight’s magnitude.

Basic AD-SAL features [9]:

- AD-SAL provides routing and optionally provides service adaptation if a northbound API is different from its corresponding southbound API. As it does not use a model-driven language (YANG), makes that data definitions and formats are not standardized. Thus, in order to make a northbound API accessible by multiple southbound plugins (each one with its own format), developers have to define a 1:1 mapping between the NB API and each of the different southbound plugins.
- Routing between consumers and providers is requested in abstraction APIs, where data adaptations are statically defined at compile-time.
- There is a dedicated REST API for each NB/SB plugin. This is due to the same problem stated before.
- It is stateless as there are no data stores to handle configuration and operational data.

### Evolution to Model-Driven Service Adaptation Layer

A new model-driven architecture was proposed and implemented for the SAL (referred to as “Model-Driven SAL”, MD-SAL). The architecture is built around the concepts, protocols and modelling language detailed in the section 1.3 [3].

One of the biggest improvements of MD-SAL is the definition of data stores, which are defined by NETCONF and modelled by YANG. They allow a new point of view of the controller plugins: now they can be either data/service providers or data/service consumers. A provider provides data/services to the data stores through its APIs. A consumer consumes services/data located in the data stores, provided by one or more providers.

Furthermore, using YANG as the modelling language allows the definition of common models and generic plugins that are available for any application. This is because now, every application or plugin will have YANG models that describe its functionality, and as a consequence, they can be easily interpreted by other application/plugins.

For example, now there is a common REST API for all applications, since they are all based on the same modelling-language and is only required to define a single API that translates the YANG format to a REST compatible one. This is definition of what is known as RESTCONF; a REST API that is auto-generated from the content of YANG models, but preserving REST interaction.

The previous example illustrates the new perspective of the MD-SAL development; now developers design plugins/applications by describing its functionality in YANG models and generating APIs from them. Generated APIs are basically composed by a set of Java classes that are automatically generated regarding the contents of the defined models. Then, developers are meant to expose their provider/consumer implementations of the generated APIs.

MD-SAL interprets and resolves plugin APIs when are loaded into the controller. However, the SAL does not contain any plugin-specific code or APIs, so it is therefore, a generic plumbing that can adapt itself to any plugins or services/applications loaded in the controller. This is possible thanks to the model-driven behaviour of MD-SAL, lead by the usage of the same modelling language among the different plugins/applications/services.

An example of a plugin is the OpenFlow (OF) plugin, which allows the interaction between the controller and OpenFlow based networks. It also provides a set of services to MD-SAL such as: topology, flow, stats, packet and error handling.

OpenDaylight's development environment includes tooling, which is centralized in the YANGTools project, that generates the code (codecs and Java APIs) from YANG models. The tooling preserves the YANG data type hierarchies, retains data tree hierarchy (providing normal Java compile-time type safety) and data addressing hierarchies.

From the infrastructure's point of view, there is no difference between a protocol plugin and application/service plugin. All plugin life-cycles are the same, each plugin is an OSGi bundle that contains models defining the plugin's APIs.

As all the plugins are the same to the controller infrastructure, the architecture can be drawn as in the figure 2.4.

Basic MD-SAL features [9]:

- Routing between consumers and provider is requested in abstraction APIs by using model definition.
- Provides a common REST API (RESTCONF) to access the data and functions defined in models.
- Data adaptations are provided by internal adaptation plugins (YANGTools project) and performed at run-time.
- Allows both the NB and SB plugins to use the same API generated from a model. One plugin becomes the API provider; the other becomes the consumer.
- MD-SAL can store data models defined by plugins, allowing provider and consumer plugins to exchange its data through the data stores.

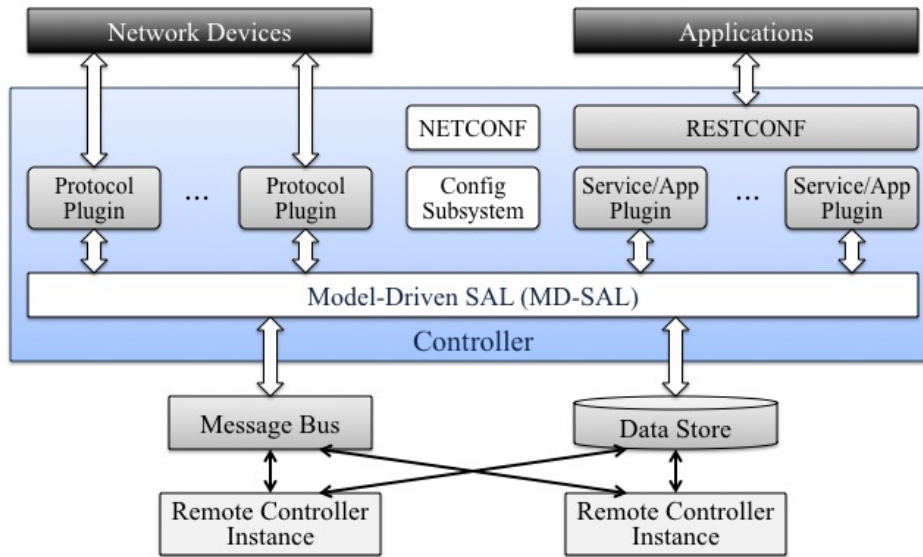


Figure 2.4: MD-SAL architecture

## 2.5 OpenDaylight projects

There are 42 projects in the OpenDaylight's latest release, Lithium. However, only those that provide a better understanding of the OpenDaylight's core will be detailed in this document.

Figure 2.5 illustrates the current projects in the OpenDaylight's latest release, Lithium.

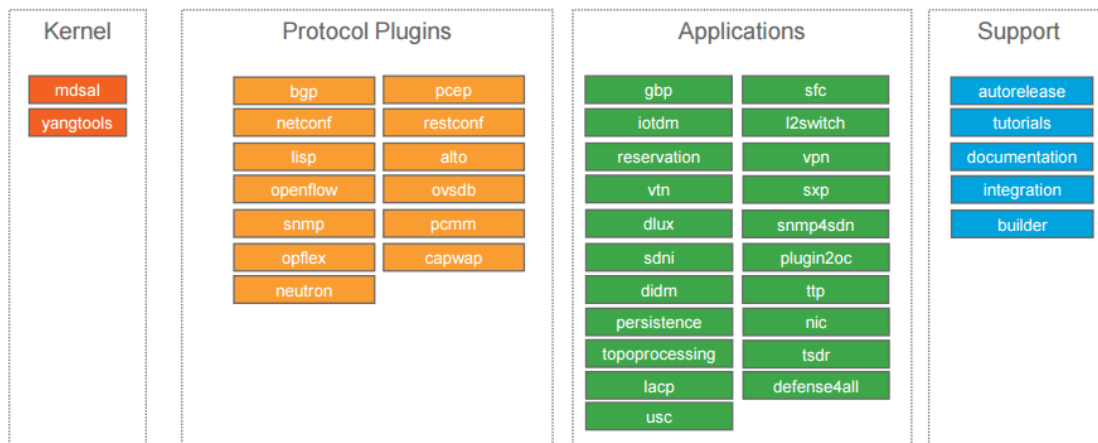


Figure 2.5: OpenDaylight Lithium projects

Dependencies among projects are modelled in a dependency tree, which implements an offset system that helps to understand and manage which projects are more important than others. A project with offset 0 does not directly depend on other projects, while projects with offset 1 depends on projects with offset 0, and so on [10].

Figure 2.6 illustrates the project dependency tree:



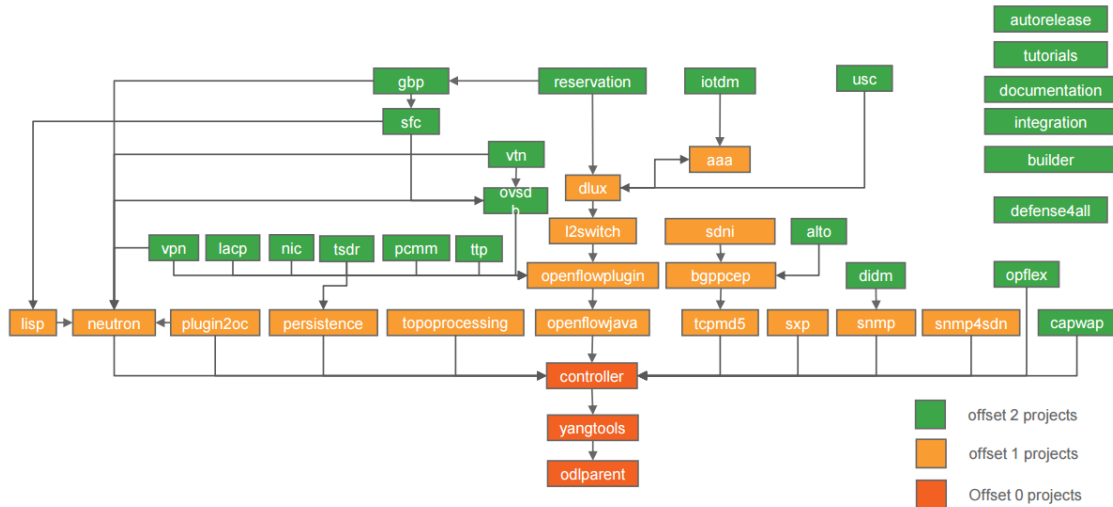


Figure 2.6: Project dependency tree

Projects that will be detailed or used in this document:

#### Offset 0:

- **ODLparent:** it is a dependency repository, being the root from which developers build. All the third-party dependencies along with its different versions are specified in the odlparent. Which also provides a versioning control as well as templates and utilities that can be used when developing an application.
- **YANGTools:** it basically parses YANG files. It is an indispensable project because it is the foundation of the way the different components communicate. YANGTools and MD-SAL projects (the content of MD-SAL project is being moved to others) are considered as the MD-SAL's core, as they provide all the plumbing and payload adaptation capabilities.
- **Controller:** is where the config subsystem (its basic definitions and interactions) and Karaf container reside.

#### Offset 1:

- **DLUX:** is an application designed to simplify and facilitate the application development and testing. It generates and renders a simple UI based on YANG models loaded into ODL.

#### Offset 2:

- **Tutorials:** it contains different examples and tutorials for various OpenDaylight projects.
- **Documentation:** aims to support OpenDaylight projects with improved documentation. It intends to improve the user content for different OpenDaylight projects and ensure a continued quality in its content.

To demonstrate OpenDaylight's multiple protocol supporting capabilities, relevant protocol plugins will be detailed [11]:

- **NETCONF plugin:** it mounts a device data tree into MD-SAL data tree. Moreover, it allows external applications to interact with the device by subscribing to its notifications and calling its RPCs.

Figure 2.7 shows an overview of the NETCONF plugin.

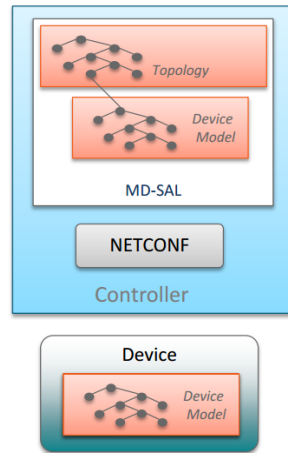


Figure 2.7: NETCONF plugin overview

- **SNMP plugin:** it automates the translation of SNMP (Simple Network Management Protocol) Management Information Bases (MIB) to YANG models. Also maps SNMP into the MD-SAL YANG modelled world.

Figure 2.8 shows an overview of the SNMP plugin.

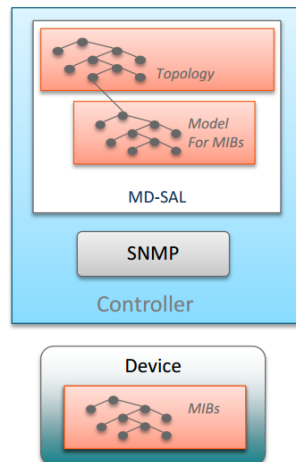


Figure 2.8: SNMP plugin overview

- **BGP plugin:** it contains a BGP (Border Gateway Protocol) listener, speaker and interprets BGP topology.

Figure 2.9 shows an overview of the BGP plugin.

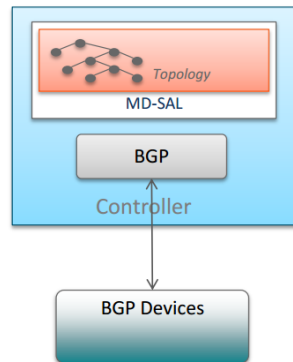


Figure 2.9: BGP plugin overview

- **PCEP plugin:** it programs traffic engineering paths using PCEP (Path Computation Element Protocol).

Figure 2.10 shows an overview of the PCEP plugin.

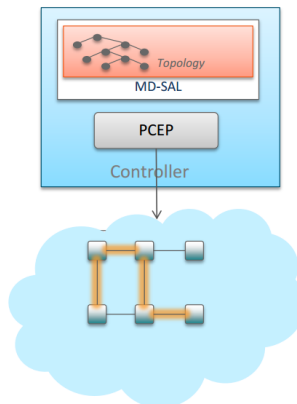


Figure 2.10: PCEP plugin overview

- **OpenFlow and OVSDb plugins:** OpenFlow (OF) plugin supports 1.0 and 1.3 versions. It provides an unified flow model, that allows to program flows on OF devices and can learn about extensions from third party code. OVSDb (Open Virtual Switch Data Base) is used to handle CRUD operations in the OVS bridges, switches and tunnels.

Figure 2.11 shows an overview of the OF/OVSDb plugin.

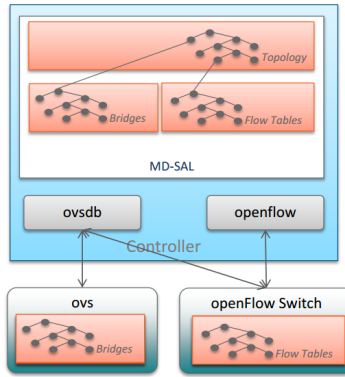


Figure 2.11: OFOVSD plugin overview

Moreover, there are different applications that provide functionality and support for other technologies. Relevant applications:

- **OpenStack Neutron:** OpenDaylight has many Neutron providers.

Figure 2.12 shows an overview of the OpenStack Neutron environment.

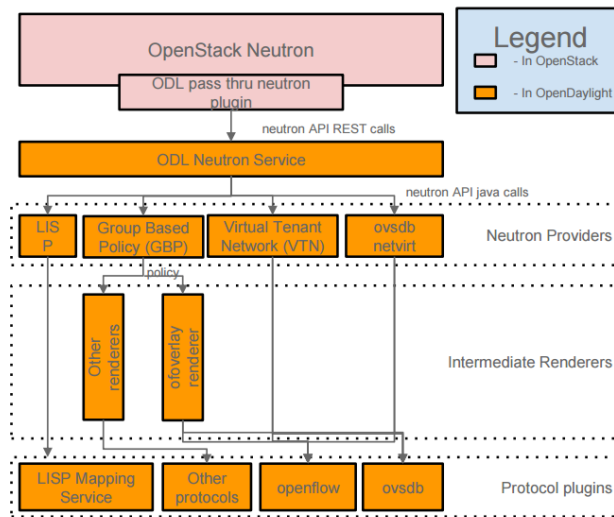


Figure 2.12: Neutron environment overview

- **Groupbased Policy (GBP):** it groups endpoints (EPs) into groups (EPGs), and applies policy (contracts) to the traffic between groups.

Figure 2.13 shows an overview of the GBP environment.

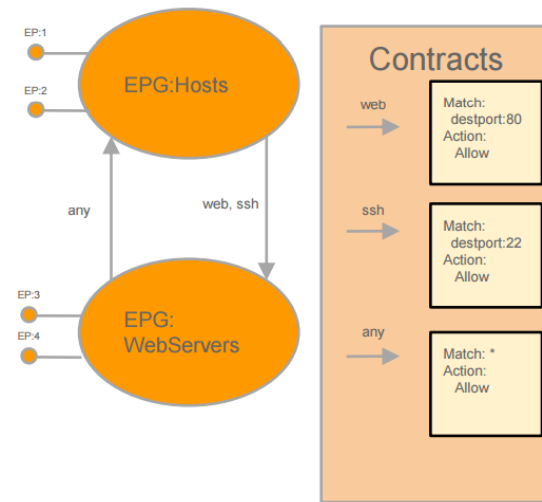


Figure 2.13: GBP environment overview

- **Service Function Chaining (SFC):** it defines an abstracted service chain, where service paths are rendered and programmed.

Figure 2.14 shows an overview of the Service Function Chaining environment.

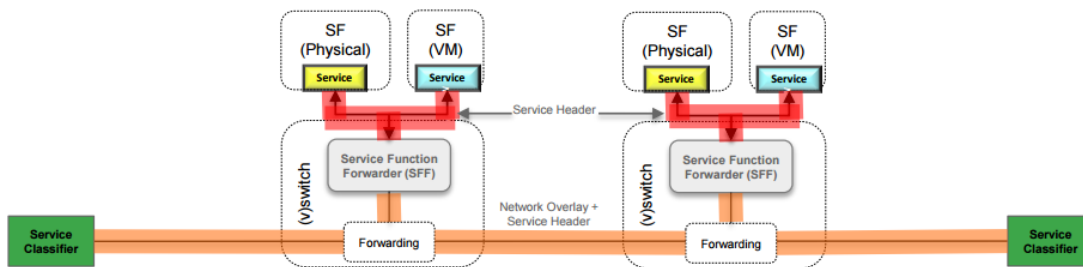


Figure 2.14: SFC environment overview

## 2.6 Configuration and installation

OpenDaylight Controller runs in a Java Virtual Machine. Being a Java application, it can potentially run on any operating system that supports Java. However, for best results using a recent Linux distribution is recommend.

The latest release (Lithium) can be found at OpenDaylight's download center [12].

The distribution has no features enabled by default. However, once the base installation is complete, additional features can be enabled simply as needed in your implementation. For compatibility reasons, you cannot simultaneously add all the features.

## 2.6.1 System requirements

- Java 7 JDK
- Modern multi-core processor and 8 GB of RAM for optimal performance.
- Highly recommended to use Linux operating system.

## 2.6.2 Downloading and installing the Java JDK

In Ubuntu:

```
> sudo apt-get install openjdk-7-jdk
```

In Fedora:

```
> sudo yum install java-1.7.0-openjdk
> sudo yum install java-1.7.0-openjdk-devel
```

## 2.6.3 Installing Git

OpenDaylight maintains all its code in Git repositories, allocated in [13]. To access them, a Git client is needed:

In Ubuntu:

```
> sudo apt-get install git-core
```

In Fedora:

```
> sudo yum install git
```

## 2.6.4 Running the Lithium distribution

OpenDaylight uses Apache Karaf to provide a lightweight container for feature modules. This allows the different features in ODL to be bundled into modules that can be loaded on-demand without reboots or instability. This capability makes it easy to run lean with only the features needed in your use case, and to enable additional features as they are needed.

To run the Lithium distribution, from the Linux prompt:

---

```
> unzip distribution-karaf-0.3.0-Lithium.zip //Unzip the zip file
> cd distribution-karaf-0.3.0-Lithium //Navigate to the directory
> ./bin/karaf //Run Karaf
```

---

## 2.6.5 Installing Karaf features

To install a feature use the following command:

```
> feature:install <feature-name>
```

The syntax for this command allows multiple features to be installed in one line, for example:

```
> feature:install <feature1-name> <feature2-name> ... <featureN-name>
```

To find a complete list of Karaf features, run the following command:

```
>feature:list
```

## 2.6.6 DLUX

Is an application designed to simplify and facilitate the application development and testing. It generates and renders a simple UI based on YANG models loaded into ODL.

Required features:

```
>feature:install odl-base-all odl-aaa-authn odl-restconf odl-adsal-northbound  
odl-mdsal-apidocs odl-l2switch-switch odl-dlux-all odl-openflowplugin-all
```

Checking if DLUX is available:

```
> http:list | grep dlux
```

It should return something similar like this:

```
282 | ResourceServlet | /dlux | Deployed | /
```

If it does not print anything, means that the features are not correctly installed, so try to install them again.

Accessing the DLUX UI:

```
http://localhost:8181/index.html
```

A login interface should appear, as shown in the figure 2.15.

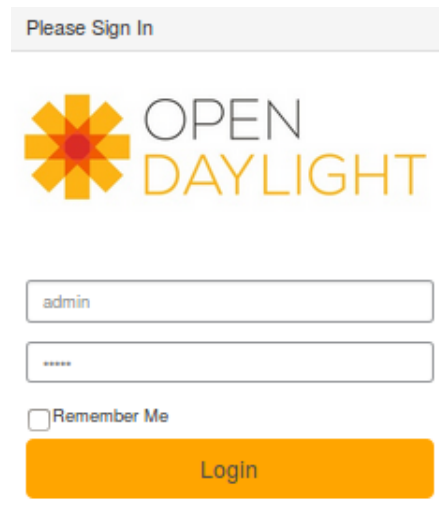
The image shows a web browser window displaying the DLUX login interface. At the top, there is a grey header bar with the text "Please Sign In". Below this is the OpenDaylight logo, which consists of a stylized orange flower-like icon to the left of the text "OPEN DAYLIGHT". Under the logo, there are two input fields: the first contains the text "admin" and the second contains a series of asterisks "\*\*\*\*\*". Below these fields is a checkbox labeled "Remember Me". At the bottom of the form is a large orange button with the text "Login" in white. The entire form is centered on a white background.

Figure 2.15: DLUX login interface

Username and password are admin/admin.





# Chapter 3

## MDSAL

### 3.1 Introduction

The MD-SAL is a message-bus inspired extensible middleware component that provides messaging and data storage functionality, based on the data and interface models defined by application developers [8].

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

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

Currently, MD-SAL exposes two different APIs:

- **Binding APIs:** Java-based APIs which extensively uses interfaces and classes generated from YANG models. They provide more compile-time safety as they use Java language to implement its content.
- **DOM (Document Object Model) APIs:** XML DOM inspired payload format APIs that interpret YANG models and provide functionality from any valid model. Using DOM format makes them more powerful and efficient, but provide less compile-time safety.

MD-SAL extends the YANGTools project, which is an infrastructure project aiming to develop necessary tooling and libraries to provide NETCONF and YANG supporting for Java projects and applications [14]. Some of its components:

- **YANG infrastructure parser:** which parses and processes YANG schemas.
- **Maven plugin:** that allows code generation based on the contents of the YANG schemas.
- **Mapping components:** that validate XML structures based on YANG schemas.
- **REST APIs:** based on YANG schemas.
- **Base YANG modules:** that provide common utilities and binding capabilities for the applications.

Model-driven nature of the MD-SAL allows for behind-the-scene APIs and payload type mediation and transformation to facilitate seamless communication between applications. This enables for other applications to provide connectors/expose different set of APIs and derive most of its functionality purely from models, which all its code can be used by other applications without requiring any modification.

## 3.2 MD-SAL format and API types:

### 3.2.1 DOM format and APIs

DOM is an application programming interface for XML and HTML documents. It defines an structured representation of a document and in which way applications can access to modify its structure, style and content [15].

DOM format is designed to be used with any programming language. In order to provide a precise, language-independent specification of DOM interfaces, it fulfills the requirements declared in the CORBA (Common Object Request Broker: Architecture and Specification) specification of the Object Management Group.

However, OpenDaylight uses XML DOM, which defines a standard way for accessing and manipulating XML documents. All XML documents can be accessed through XML DOM. It defines objects, properties and methods valid for all XML documents. In other words, is a standard for how to get, change, add or delete XML elements [16].

In DOM, everything in an XML document is modelled a node:

- The entire document is a document node.
- Every XML element is an element node.
- The text in the XML elements are text nodes.
- Every attribute is an attribute node.
- Comments are comment nodes.

XML DOM views an XML document as a tree-structure, named node-tree. All nodes can be accessed through the tree and its contents can be modified or deleted. Also new elements can be created and eliminated as well.

Figure 3.1 illustrates a node-tree with a set of nodes and the connections between them.

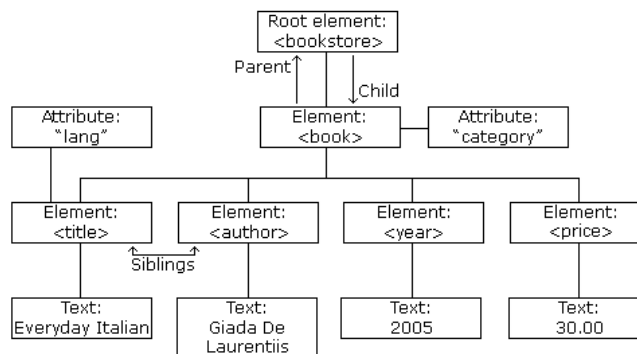


Figure 3.1: DOM node-tree representation

The nodes in a tree node have a hierarchical relationship to each other. The terms parent, child and sibling are used to describe relationships:

- In a node tree, the top node is called root.
- Every node, except the root, has exactly one parent node.
- A node can have any number of children.
- A leaf is a node with no children.
- Siblings are nodes with the same parent.

Figure 3.2 illustrates the different relationships between nodes.

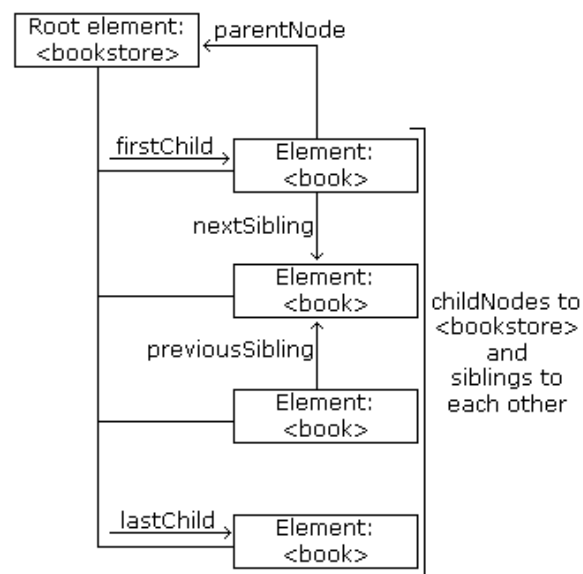


Figure 3.2: Relationships between DOM nodes

Properties are intended to obtain information about element nodes. Examples:

- `x.nodeName` – name of `x`
- `x.nodeValue` – value of `x`
- `x.parentNode` – parent node of `x`
- `x.childNodes` – child nodes of `x`
- `x.attributes` – attribute nodes of `x`

Methods are intended to describe interactions between different element nodes. Examples:

- `x.getElementsByTagName(name)` – gets all the elements with a specified tag.

- `x.appendChild(node)` – inserts a child node to `x`.
- `x.removeChild(node)` – removes a child node from `x`.

XML DOM is the data format in which the MD-SAL internally operates. In fact, data stores, which are considered as the core of the MD-SAL, use XML DOM data representation. This is mainly because:

- Data stores are based on the NETCONF protocol, which is an XML-based protocol that is used to define operational and configuration data stores and its supported operations. Moreover, data stores have to be accessed by RESTCONF, which uses XML or JSON payload types.
- YANG is the modelling language that is used to model the contents and the behaviour of data stores. YANG structures data in a similar way as XML DOM, it represents data as a tree that contains nodes and the relationships between them.

The previous statements illustrate how XML DOM perfectly fits on the technologies involved on the definition and modelling of data stores. For this reason, it is the chosen data format to operate with them.

This means that the data modelled in YANG schemas has to be translated, in some way, to an XML DOM format in order to populate data stores. How is this performed? The answer is that no translation is needed as YANG has an alternative XML-based representation syntax called YIN, which contains equivalent information using different notations. Mapping between YANG and YIN does not modify the information content of a model.

However, MD-SAL data stores are not exactly XML-based, since some terms and definitions had to be adapted to better fit Java class system and to support consumer use cases. This implies that the correspondence between YANG (specifically YIN) and the refined XML DOM, that is used in data stores, is not exactly equivalent. However, YANGTools project contains modules that refine the YANG XML behaviour and adapts it to the refined XML DOM in which data stores operate [17].

Some of the adapted terms and definitions are:

- **Handling of multiple module revisions:** introduces a layer of separation that isolates different data, RPC and notification implementations of the same module. By using module revision, multiple client versions can be supported without the risk of accidentally changing data whose semantics changed between revisions. Because multiple versions of the same module can coexist in the system, MD-SAL must support plugins responsible for translating data between different module revisions.
- **QNames:** they are used to reference particular elements or attributes within XML documents. A regular XML QName consists of a local element name and XML namespace. It is used to prevent name clashes between nodes with the same local name but from different schemas (XML namespaces).

In YANG context, local element points to a defined node, procedure or notification.

To support versioning, module revision was added in the QName definition: now it consists of XML namespace, YANG module revision and local name.

QName = (XMLNamespace, Revision, LocalName), where:

- **XMLNamespace:** namespace assigned to the YANG module that contains the defined element, type, procedure or notification.
- **Revision:** revision of the YANG module that describes the element.
- **LocalName:** YANG schema identifier given to the node in the YANG module.

- **RPCs:** in regular NETCONF / YANG use cases, RPCs are used to model functionality and APIs provided by NETCONF server and consumed by a NETCONF client. While, in MD-SAL context, RPCs are used to model functionality provided by different providers and consumed by different consumers. For this reason, Java Future mechanisms have to be added to RPC calls in order to provide lock prevention.

DOM APIs are also known as binding-independent APIs as they do not use the generated bindings from YANG models. They interpret any valid YANG model and can provide functionality to it by using General Purpose Models (GPM), which can be provided either in YANG (general purpose modelling language) or XML (general purpose markup language). Since they can interpret any YANG model, DOM APIs are mostly used for components, such as data stores, RESTCONF connector, NETCONF, etc.

RESTCONF connector is a DOM API that allows accessing and manipulating data stores via a REST interface. Requests can perform the typical CRUD operations as well as RPC and notifications, and can be either in XML or JSON format. It uses QNames to identify the specific nodes to perform operations.

### 3.2.2 Binding format and APIs

The binding model is a specification that describes how a YANG schema and the binding-independent data format are translated into generated bindings (statically-types Java interfaces, Data Transfer Objects (DTOs), builders and mappers) [18].

This translation is bidirectional. Binding-aware providers expose their implementations by using the generated Java APIs, which have to be translated into XML DOM format if they have to access and operate in the data stores. Binding-aware consumers usually receive data/information from the data stores, which operates in XML DOM format, so it has to be translated into Java format in order to be visible by binding-aware consumers.

DTOs are reusable classes that contain related data and no business logic. They are commonly used to transfer data as they reduce the amount of data to be sent across. DTOs encapsulate data using getters and setters, which may wrap primitive data types (integers, strings, etc.) or other DTOs. They can be used, for example, to encapsulate the input parameters of an RPC call.

The binding model has two subsystems:

- **Consumer and provider binding:** a subset of the binding model which is directly visible to consumers and providers; it uses binding at compile-time. This subset consists of generated Java interfaces and DTOs that allow an easier development of applications that are tailored to specific models.
  - **Data Transfer Objects:** represent instances of data nodes defined by a YANG schema and are used to store and transfer data. Data Transfer Objects are immutable and are represented as interfaces, which are generated in compile-time binding.
  - **DTO builders:** concrete classes that create specific DTOs. Builders are represented as interfaces, which are generated in compile-time binding.
  - **RPC interfaces:** they represent the programmatic API to invoke RPCs defined by providers. RPC interfaces describe all the available RPCs in a YANG model as methods to be implemented.
- **Binding infrastructure components:** infrastructure components are not directly visible to consumers and providers. They are used by the config subsystem, and are responsible for the implementation of binding functionality and generated Java interfaces. Infrastructure components are only available at run-time and are typically dynamically generated. Its usage is detailed in the section 4.2.5.

One of the main benefits of using Java as the programming language to implement the generated APIs is the creation of consistent DTOs everywhere that are [11]:

- **Immutable:** DTOs cannot change, so there is no need to worry about lock prevention as they cannot be modified by running threads.
- **Strongly typed:** it catches a large amount of common errors such as placing wrong things in the wrong place.
- **Consistent:** generated interfaces always follow the same convention.
- **Improvable:** when an improvement is done in a model, as DTOs are generated from it, it is automatically available to them.
- **Automated bindings:** to RESTCONF (XML and JSON), NETCONF, etc.
- **Run-time generable:** from models that dynamically discover at run-time.

The number and type of generated Java classes depend on the statements defined in a YANG schema. YANGTools provides the required tooling to interpret YANG schemas and generate binding APIs. YANG to Java mapping is detailed in the section 7.5.

Binding APIs are targeted to application and provider development. Applications rely on a set of specific models, from which Java-based APIs are generated, which developers (providers) are meant to implement.

### 3.2.3 RESTCONF interface

When we talk about REST we refer to any interface between systems that directly uses HTTP (HyperText Transfer Protocol) for the communication in any format.

In OpenDaylight framework:

- Data stores are created by the NETCONF protocol, which also defines its supported operations.
- Data structures and its interactions are modelled by using YANG modelling language.
- RESTCONF is a protocol implemented in the SAL that describes how to map a YANG specification for a REST interface, which is then used to access and manipulate data.

RESTCONF requests are detailed in the section 3.9.

## 3.3 Basic concepts

Basic concepts are building blocks that are used by applications, and from which MD-SAL uses to define messaging patterns, and to provide services and behaviour based on the developed-supplied YANG models [8]:

- **Data trees:** data is modelled and represented as a tree, with possibility to address any element/subtree.
  - **Operational data tree:** reports the state of the system, published by providers using the MD-SAL. Represents a feedback loop for applications to observe the status of the network/system.
  - **Configuration data tree:** shows the intended state of the system or network. It is populated by consumers, which express their intention.
- **Instance identifier:** unique identifier of a node/subtree in a YANG data tree, which provides unambiguous information. It is used to retrieve a node/subtree from conceptual data trees.

- **Notification:** asynchronous transient event (from perspective of provider) that can be consumed by consumers, which may act upon it.
- **RPC:** asynchronous request-reply message pair. Request is triggered by consumer and sent to the provider, which replies with a reply message.

### 3.4 MD-SAL design

Data handling functionality is separated into two distinct brokers: a binding-independent DOM broker, which is the core component of the MD-SAL, and interprets YANG models at run-time, and a binding-aware broker that exposes Java APIs for plugins that use binding-aware representation of data [3].

MD-SAL brokers, along with their supporting components, are shown in the figure 3.3.

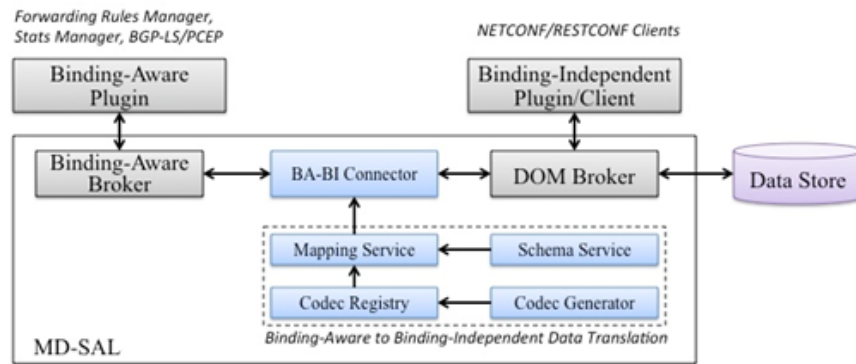


Figure 3.3: MD-SAL design

DOM broker uses YANG data APIs to describe data, and instance identifiers (QNames in binding-independent terminology) to describe paths to specific elements in the data stores. Basically, the DOM broker is the one that manipulates requests in the nodes of the data stores. The DOM broker relies on the presence of YANG schemas, which are interpreted at run-time for functionality-specific purposes, such as RPC routing, data store organization, and validation of paths.

The binding-aware broker relies on Java APIs, which are generated from YANG models, and on common properties of Java DTOs, which are enforced by code generation.

The binding-aware broker connects to the DOM broker through the BA-BI connector, so that binding-aware consumer/provider applications/plugins can communicate with their respective binding-independent counterparts. The BA-BI connector, together with the Mapping Service, the Schema Service, the Codec Registry and the Codec Generator implement dynamic late binding: the codecs translate YANG data representations between a binding-independent (DOM) format and DTOs, which are specific to Java bindings (auto-generated on demand).

OpenDaylight data storage is composed by data trees, which split into two logical data stores: operational and configuration. However, we have unified view of both and we can address to specific node by using instance identifiers.

Figure 3.4 shows how different data models can coexist in different subtrees under the same logical data stores: operational and configuration.

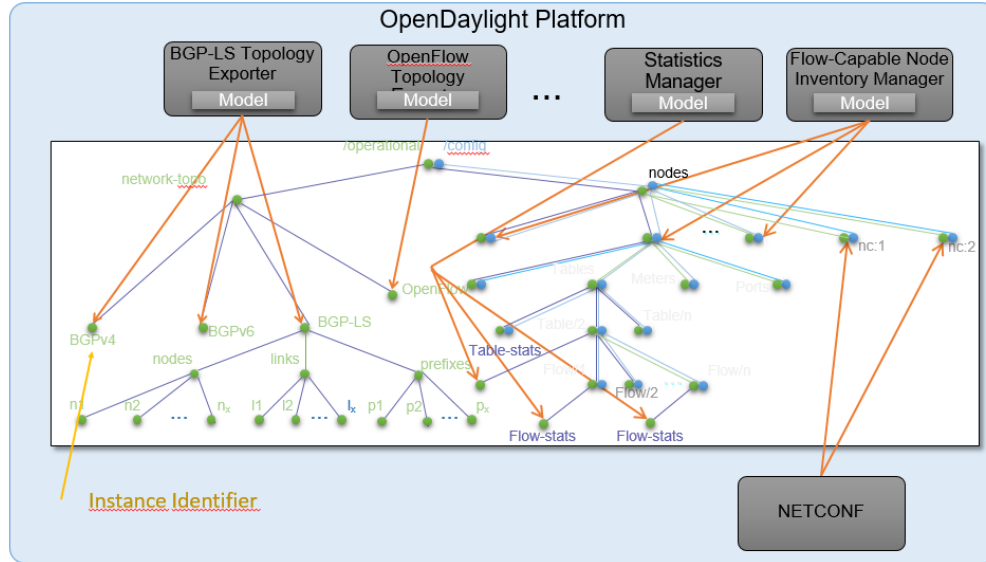


Figure 3.4: Structure of data stores

### 3.5 Messaging patterns

MD-SAL provides a set of messaging patterns, which are derived from the basic concepts. Messaging patterns use brokers, which are intended to transfer the YANG modelled data between applications to provide data-centric integration instead of API-centric integration [8].

Actually, OpenDaylight defines different brokers that manage specific messaging patterns, as shown in figure 3.5 [11].

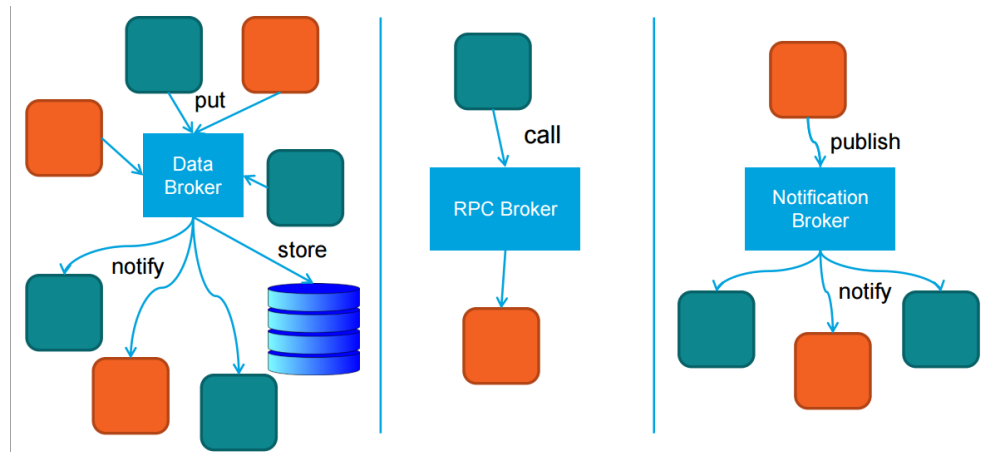


Figure 3.5: Different broker types

#### Transactional access to data tree:

- Transactional reads from a conceptual data tree, read-only transactions.
- Transactional modifications to a conceptual data tree, write transactions.



- Transaction chaining.

#### **Unicast communication:**

- **RPC:** unicast message between consumer and provider, consumer sends request message to provider, which asynchronously responds with reply message.

#### **Publish / Subscribe:**

- **Notifications:** multicast message which is send by provider and is delivered to its subscribers.
- **Data Change Events:** multicast asynchronous event, which is sent by the data broker if there is a change in a conceptual data tree, and is delivered to its subscribers.

## **3.6 MD-SAL data transactions**

MD-SAL data broker provides transactional access to conceptual data trees representing configuration and operational state [8].

Transactions provide an stable and isolated view from other currently running transactions. In other words, the state of running transaction and its underlying data tree is not affected by other concurrent transactions.

### **3.6.1 Transaction types**

- **Write-only:** provides modification capabilities, but does not provide read capabilities. Write-only transaction is allocated using `newWriteOnlyTransaction()`.
- **Read-write:** provides read and write capabilities. It is allocated using `newReadWriteTransaction()`.
- **Read-only:** provides stable read-only view based on current data tree. Read-only view is not affected by any subsequent write transactions. Read-only transaction is allocated using `newReadOnlyTransaction()`.

### **3.6.2 Write-only and read-write transactions**

They provide modification capabilities for the conceptual data trees.

Work-flow for data tree modifications:

1. Application allocates new transactions using `newWriteOnlyTransaction()` or `newReadWriteTransaction()`.
2. Application modifies data trees by using `put`, `merge` and/or `delete` operations.
3. Application finishes a transaction using `submit()`, which seals transaction and submits it to be processed.
4. Application observes the result of the transaction using either blocking or asynchronous calls.

The initial state of a write transaction is a stable snapshot of the data tree state captured when transaction was created, which is not affected by other concurrently running transactions. Modifications are local to the transaction; they only represent a proposal state change for a concrete data tree and are not visible to any concurrently running transactions until they are committed.

Flow diagram (using pseudocode):

1. Creating a new transaction: we create a local transaction field where we will propose state changes. Required code is detailed in the figure 3.6.

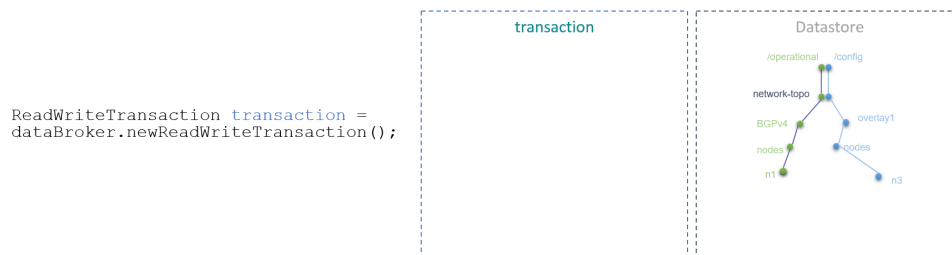


Figure 3.6: Creating a new transaction

2. Modifying data tree: we read n1 node by using its instance identifier, and we modify the configuration tree by adding a new node n2 and deleting the existing node n3. Required code is detailed in the figure 3.7.

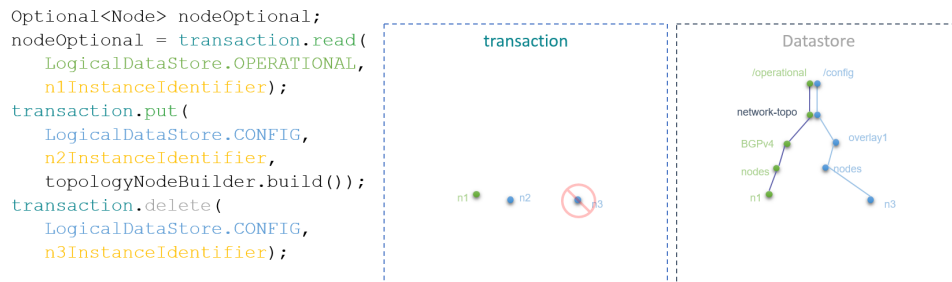


Figure 3.7: Modifying data tree

3. Finishing transaction and observing the result using CheckedFuture, if the transaction is successful, the data store is modified according to the defined transaction. Required code is detailed in the figure 3.8.

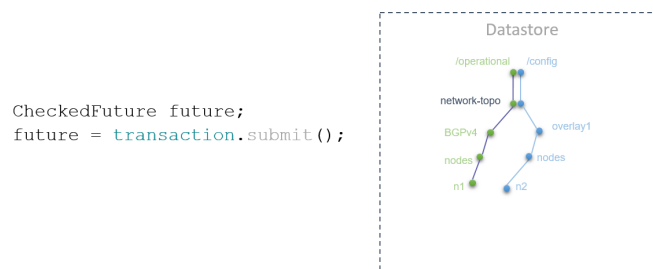


Figure 3.8: Finishing data transaction

## Transaction operations

Write-only and read-write transactions provide following methods to modify data trees:

- **Put:** stores a piece of data at specified path. This acts as an add/replace operation, which is to say that whole subtree will be replaced with the specified data.

```
<T> void put(LogicalDataStoreType store, InstanceIdentifier<T> path, T data);
```

Figure 3.9 illustrates how to create a put operation.

```
WriteOnlyTransaction transaction =  
dataBroker.newWriteOnlyTransaction();  
InstanceIdentifier<Node> path =  
    InstanceIdentifier.  
        .create(NetworkTopology.class)  
        .child(Topology.class,  
            new TopologyKey(  
                "overlay1"));  
transaction.put(  
    LogicalDataStore.CONFIG,  
    path,  
    topologyBuilder.build());  
CheckedFuture future;  
future = transaction.submit();
```

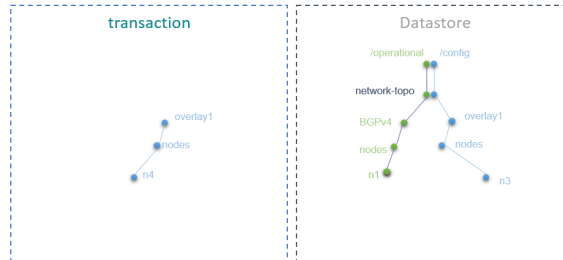


Figure 3.9: Creating a put transaction

Figure 3.10 shows how the n3 node is deleted after submitting the transaction.

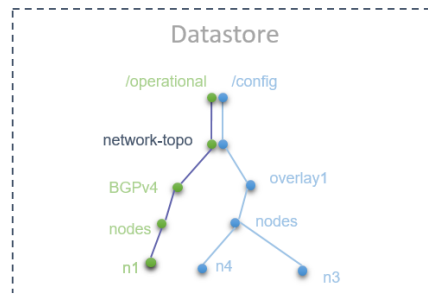


Figure 3.10: Put transaction result

- **Merge:** merges a piece of data with the existing data at specified path. Any pre-existing data which are not explicit overwritten will be preserved. This means that if you merge a container, its child subtrees will be merged too.

```
<T> void merge(LogicalDataStoreType store, InstanceIdentifier<T> path, T data);
```

Figure 3.11 illustrates how to create a merge operation.

```

WriteOnlyTransaction transaction =
dataBroker.newWriteOnlyTransaction();
InstanceIdentifier<Node> path =
    InstanceIdentifier
        .create(NetworkTopology.class)
        .child(Topology.class,
            new TopologyKey(
                "overlay1"));
transaction.merge(
    LogicalDataStore.CONFIG,
    path,
    topologyBuilder.build());
CheckedFuture future;
future = transaction.submit();

```

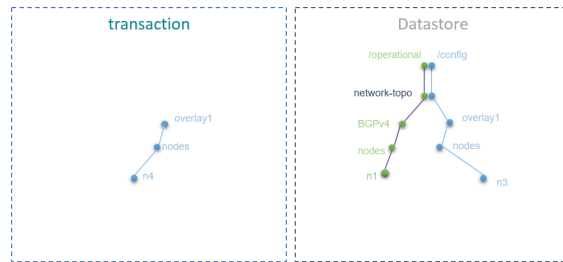


Figure 3.11: Creating a merge transaction

Figure 3.12 shows how the n3 node is preserved after submitting the transaction.

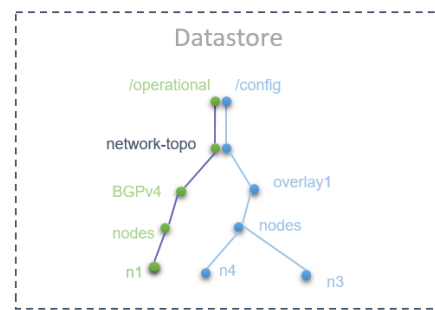


Figure 3.12: Merge transaction result

- **Delete:** removes a whole subtree from a specified path.

```
void delete(LogicalDataStoreType store, InstanceIdentifier<?> path);
```

## Submitting transactions

Transaction is submitted to be processed and committed using following method:

```
CheckedFuture<Void, TransactionCommitFailedException> submit();
```

Applications publish the proposed change by calling `submit()` on the transaction. This seals the transaction (preventing any further writes using this transaction) and submits it to be processed and applied to global conceptual data tree. The `submit()` method does not block, but rather returns `ListenableFuture`, which will complete successfully once processing transaction is finished and changes are applied to data tree. If data commit fails, the future will fail with `TransactionFailedException`.

Application may asynchronously listen on the state of the commit by using `ListenableFuture`. It is shown in the figure 3.13.

1. Submits `writeTx` and registers application provided by using `FutureCallback` on returned future.
2. Invoked when future completed successfully: transaction `writeTx` was successfully committed to data tree.
3. Invoked when future failed: commit of transaction `writeTx` failed. Supplied exception provides additional details and cause of failure.

```

Futures.addCallback( writeTx.submit(), new FutureCallback<Void>() { ❶
    public void onSuccess( Void result ) { ❷
        LOG.debug("Transaction committed successfully.");
    }

    public void onFailure( Throwable t ) { ❸
        LOG.error("Commit failed.",e);
    }
});

```

Figure 3.13: Checking commit with ListenableFuture

If application needs to block until commit, it may use `checkedGet()` to wait until it is finished. It is shown in the figure 3.14.

```

try {
    writeTx.submit().checkedGet(); ❶
} catch (TransactionCommitFailedException e) { ❷
    LOG.error("Commit failed.",e);
}

```

Figure 3.14: Blocking until commit

1. Submits `writeTx` and blocks until commit of `writeTx` is finished. If commit fails `TransactionCommitFailedException` will be thrown.
2. Catches `TransactionCommitFailedException` and logs it.

### Transaction local state and isolation

Read-write transactions maintain transactional-local state, which renders all modifications as if they happened, but only local to transaction.

Read operations on the same transaction returns data as if the previous modifications in the transaction were submitted.

Figure 3.15 illustrates the local state behaviour, assuming that initial state of data tree for `PATH` is `A`.

```

ReadWriteTransaction rwTx = broker.newReadWriteTransaction(); ❶

rwTx.read(OPERATIONAL, PATH).get(); ❷
rwTx.put(OPERATIONAL, PATH, B); ❸
rwTx.read(OPERATIONAL, PATH).get(); ❹
rwTx.put(OPERATIONAL, PATH, C); ❺
rwTx.read(OPERATIONAL, PATH).get(); ❻

```

Figure 3.15: Checking the local state behaviour

1. Allocates new `ReadWrite` transaction.
2. Read from `rwTx` will return value `A` for `PATH`.
3. Writes value `B` to `PATH` using `rwTx`.

4. Read now will return value B for PATH, since previous write occurred in same transaction.
5. Writes value C to PATH using `rwTx`.
6. Read now will return C for PATH, since previous write occurred in same transaction.

Running (not submitted) transactions are isolated from each other, and changes done in one transaction are not observable in other current running transaction

Figure 3.16 illustrates the isolation behaviour, assuming that initial state of data tree for PATH is A.

```

ReadOnlyTransaction txRead = broker.newReadOnlyTransaction(); ❶
ReadWriteTransaction txWrite = broker.newReadWriteTransaction(); ❷

txRead.read(OPERATIONAL, PATH).get(); ❸
txWrite.put(OPERATIONAL, PATH, B); ❹
txWrite.read(OPERATIONAL, PATH).get(); ❺
txWrite.submit().get(); ❻
txRead.read(OPERATIONAL, PATH).get(); ❼
txAfterCommit = broker.newReadOnlyTransaction(); ❽
txAfterCommit.read(OPERATIONAL, PATH).get(); ❾

```

Figure 3.16: Checking the isolation behaviour

1. Allocates read only transaction, which is based on data tree which contains value A for PATH.
2. Allocates another transaction, read-write, which is based on the same data tree.
3. Read from read-only transaction returns value A for PATH.
4. Data tree is updated using read-write transaction, PATH contains B. Change is not public and is only local to transaction.
5. Read from read-write transaction returns value B for PATH.
6. Submits changes in read-write transaction to be committed to data tree. Once commit will finish, changes will be published and PATH will be updated for value B. Previously allocated transactions are not aware of this change.
7. Read from previously allocated read-only transaction still returns value A for PATH, since it provides stable and isolated view.
8. Allocates new read-only transaction, which is based on data tree which contains value B for PATH.
9. Read from new read-only transaction return value B for PATH since read-write transaction was committed.

### Commit failure scenarios

A transaction commit may fail because of following reasons:

- **Optimistic Lock Failure:** another transaction finished earlier and modified the same node in a non-compatible way. The commit (and the returned failure) will fail with an `OptimisticLockFailedException`.

It is the responsibility of the caller to create a new transaction and submit the same modification again in order to update data tree.

Figure 3.17 shows an example of this failure scenario.

```

WriteTransaction txA = broker.newWriteTransaction();
WriteTransaction txB = broker.newWriteTransaction();

txA.put(CONFIGURATION, PATH, A);    ❶
txB.put(CONFIGURATION, PATH, B);    ❷

CheckedFuture<?,?> futureA = txA.submit(); ❸
CheckedFuture<?,?> futureB = txB.submit(); ❹

```

Figure 3.17: Optimistic Lock Failure scenario

1. Updates PATH to value A using txA.
  2. Updates PATH to value B using txB.
  3. Seals and submits txA. The commit will be processed asynchronously and data tree will be updated to contain value A for PATH. The returned `ListenableFuture` will complete successfully once state is applied to data tree.
  4. Seals and submits txB. Commit of txB will fail because previous transaction also modified PATH in a concurrent way. The state introduced by txB will not be applied. The returned `ListenableFuture` will fail with `OptimisticLockFailedException`, which indicates that concurrent transaction prevented the submitted transaction from being applied.
- **Data validation:** the data introduced by the transaction did not pass validation by commit handlers or data was incorrectly structured. The returned future will fail with a `DataValidationFailedException`. User should not retry to create new transaction with same data, as it will probably fail again.

## 3.7 MD-SAL RPC routing

Remote Procedure Calls (RPCs) are used to model any call implemented by a provider, which exposes functionality to consumers. In the MD-SAL terminology, the term RPC is used to define the input and output for a function that is to be provided by a provider, and adapted by the MD-SAL [8].

RPCs allows to asynchronously send messages and receive responses, without any knowledge of the provider of the implementation.

In the MD-SAL context, there are two RPC types:

- **Global:** one service instance (implementation) per controller container.
- **Routed:** one receiver per context, allowing multiple implementations per controller container.

There are different ways to define and implement RPCs depending on the used API types:

- **JAVA generated APIs:** for each YANG model there is a generated Service interface that defines methods for all the RPCs statements declared on it. Service interfaces and YANG to Java mapping are detailed in the section 7.5.

Providers expose their implementation of Service interface by registering it in its `ProviderContext`.

Consumers get the Service implementation. If the implementation uses a different API type, the MD-SAL automatically translates data in background.

- **DOM APIs:** RPCs are identified by QNames.

Figure 3.18 shows the RPC routing procedure between different data formats [19].

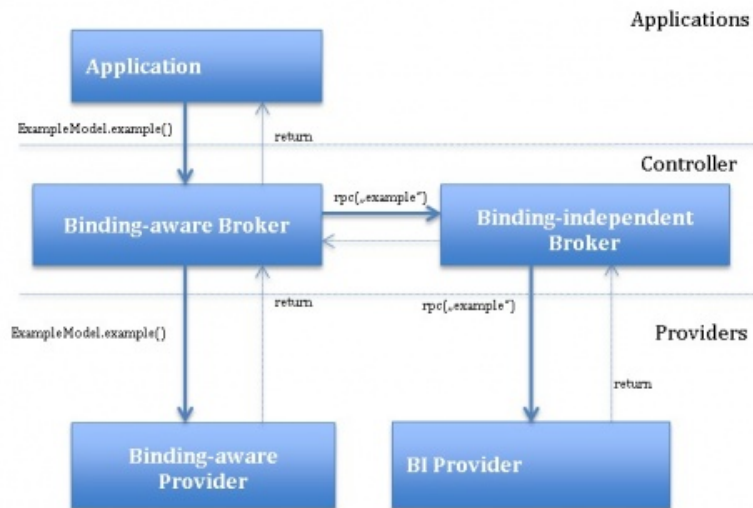


Figure 3.18: RPC routing between different data formats

1. The consumer (application) gets the generated proxy for the requested functionality.
2. The consumer constructs the RPC input.
3. The consumer invokes a method, which represents the RPC call, on the retrieved proxy.
4. The generated proxy processes the call:
  - If the RPC provider uses the same language-binding, its implementation is directly invoked.
  - If the provider is binding-independent:
    - I The call is generated in binding-independent format.
    - II The binding-independent call is invoked on the BI (binding-independent) broker.
    - III The BI broker routes the RPC and receives the binding-independent response.
    - IV The BI response is translated to the language-binding format in the binding-aware broker.

### 3.7.1 Sending a RPC message

Figure 3.19 illustrates how to call a RPC method and obtain its result (output).

1. The first step is to get an instance of the desired RPC service implementation in our `session`. The `getRpcService()` method, informs the MD-SAL that we are a consumer of the specified service. By doing this, we are able to call any RPC method that is defined on it. In this particular case, we obtain the RPC methods defined in the `HelloService` class.
2. RPCs use Java futures to asynchronously compute its tasks and handle its results. So the next step is to define an `RpcResult` future that will compute and handle the result of our RPC call. In this particular case, the result will be a `HelloWorldOutput` object.



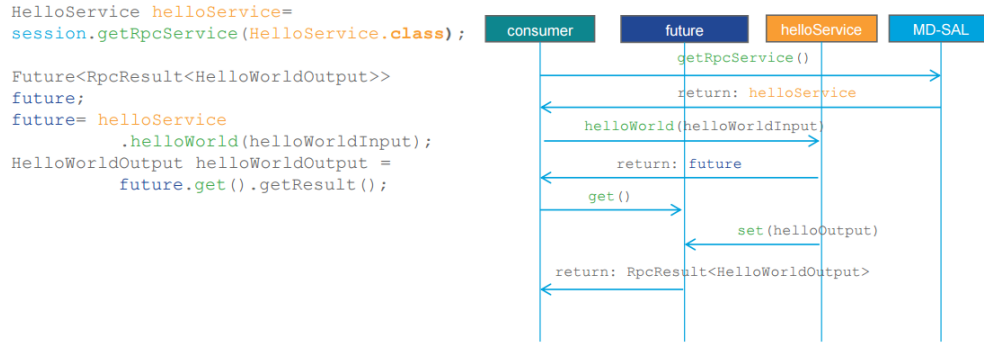


Figure 3.19: Sending a RPC message

3. To send a RPC message we have to call the desired RPC method and link it to our `RpcResult`. In this particular case, we call the `helloWorld()` method (defined in `HelloService`) and we pass `helloWorldInput` parameter to it. Result (`HelloWorldOutput`) is handled in our defined `RpcResult`, named `future`.
4. The last step is to obtain the output of our RPC call. This is done by calling the `get()` method in our `RpcResult`. Note that the `get()` method is blocking, which means that application will wait to finish it before continuing with its processing. In this case, the `get()` method returns the desired `HelloWorldOutput` object.

### 3.7.2 Global RPCs

In order to define global RPCs, YANG model authors only need to declare RPC statements, defined in the section 7.4.1.

#### Global RPC implementation

Figure 3.20 illustrates how to implement a global RPC method.

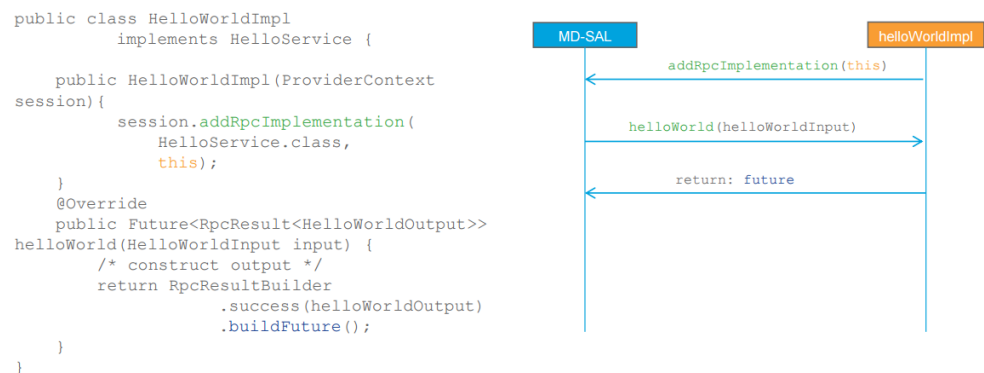


Figure 3.20: Defining a global RPC implementation

- First step is to implement the Service interface where our RPC methods are declared. In this particular case, the interface to be implemented is `HelloService`.

- We have to register our class as RPC implementation provider in the `ProviderContext`, by using the `addRpcImplementation()` method. By doing this, we inform to the MD-SAL that we are providing an implementation of the specified Service interface. Consumers, then, can obtain the implementation by calling the `getRpcService()` method.
- Now, we have to override the RPC methods by adding our custom implementations. This example is the simplest way to implement an RPC since we are manually building the `HelloWorldOutput` by using `RpcResultBuilder`.

### 3.7.3 Routed RPCs

MD-SAL provides a way to deliver RPCs to a particular implementation based on a context. When modelling the RPC in YANG, a part of the input will be used as context reference, which will specify which implementation is used.

MD-SAL does not dictate the name of the leaf which is used for RPC routing, but provides to developers the necessary functionality to define their context references in their RPC models.

MD-SAL routing behaviour is modelled using the following terminology and its application to YANG models:

- **Context type:** logical type of RPC routing. Context type is modelled as a YANG identity and is referenced in the model to provide scoping information.
- **Context instance:** conceptual location in the data tree, which represents the context in which the RPC could be executed. Context instance usually represents the logical point to which RPC execution is attached.
- **Context reference:** field of the RPC input payload which contains an instance identifier that references the context instance in which the RPC should be executed.

#### Modelling routed RPCs

NOTE: It is highly recommended to take a look at the YANG chapter 7, as specific YANG statements will be used in the modelling of routed RPCs.

In order to define routed RPCs, YANG model authors needs to declare (or reuse): a context type, a set of possible context instances and finally the RPCs, which will contain the context reference in which they will be routed.

Declaring a routing context type:

---

```
identity node-context {
    description "Identity used to mark node context";
}
```

---

This declares an identity named `node-context`, which is used as a marker for node-based routing and is used in other places to reference that routing type.

Defining context instances:

In order to define possible values of context instances for routed RPCs, we need to model that desired set of nodes and declare them as context instance.

This is achieved by using the YANG extension `context-instance` from the `yang-ext` model (declared in the MD-SAL project).

---

```
/** Base structure */
container nodes {
  list node {
    key "id";
    ext:context-instance "node-context";
    // other node-related fields would go here
  }
}
```

---

The statement `ext:context-instance "node-context"`; marks any element of the list node as a possible valid context instance in the `node-context` based routing.

#### Declaring a routed RPC:

To declare an RPC to be routed on the `node-context`, we need to add an input leaf to the RPC and mark it as context reference.

This is achieved by using the YANG extension `context-reference` from `yang-ext` model on the input leaf that will be used for RPC routing.

---

```
rpc example-routed-rpc {
  input {
    leaf node {
      ext:context-reference "node-context";
      type "instance-identifier";
    }
    // other input to the RPC would go here
  }
}
```

---

The statement `ext:context-reference "node-context"` marks the input node leaf as a context reference for the `node-context` routing. We also have to declare the node leaf as `instance-identifier` type, so it can refer to the declared context instances. The value of this leaf, will be used by the MD-SAL to select the specific RPC implementation that has registered using that particular context instance.

### **Implementing routed RPCs**

Now, providers will have to specify its instance identifier value for the context reference in which the RPC is declared. This is performed during the registration.

Figure 3.21 illustrates how to implement a routed RPC method.

In this example, providers specify their instance identifier (`iid1` and `iid2`) for the context reference, `MyContext`, during the registration. For example, the `HelloWorldImpl1()` implementation will be only executed if a consumer specifies the `iid1` instance identifier in the `helloWorld()` RPC input.

Routed RPCs are very useful and commonly used in OpenDaylight. For example, let's assume that we define a container named `ipversion` that contains the different IP versions as leaves: `ipv4` and `ipv6`. Routed RPCs can be used to define different implementations of an RPC in function of the chosen IP version. Consumers, when invoking the RPC, will have to pass, as an input parameter, the context reference describing their IP version.

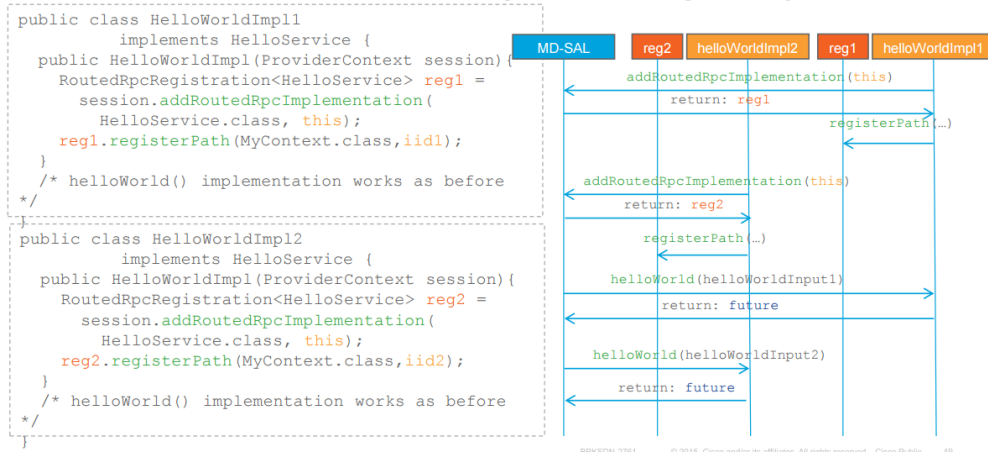


Figure 3.21: Defining a routed RPC implementation

From a user perspective there is no difference between routed and global RPCs. Routing information is just an additional leaf in the RPC input that has to be populated.

### 3.8 MD-SAL notifications routing

Notifications represent asynchronous events published by providers and delivered to subscribed listeners [8].

There are different ways to define and implement notifications depending on the used API types:

- **JAVA generated APIs:** for each YANG model there is a generated Listener interface and Data Transfer Objects that define the notifications statements declared on it. Listener interfaces and YANG to Java mapping are detailed in the section 7.5.

Providers publish notifications by invoking the `publish()` method defined on the `NotificationProviderService` class.

To receive notifications, consumers register their implementation of the Listener interface to the `NotificationProviderService`. If the notification publisher uses a different API type, MD-SAL automatically translates data in the background.

- **DOM APIs:** notifications are identified by QNames.

Figure 3.22 shows the notification routing procedure between different data formats [19]. Followed procedure:

1. Binding-aware broker invokes the method to all the binding-aware consumers that are subscribed to the notification.
2. Binding-aware translates the notification to the binding-independent format.
3. Notification in binding-independent format is forwarded to BI Broker, which forwards the notification to all BI registered listeners.

In order to define notifications, YANG model authors only need to declare notification statements. YANG notification declarations are detailed in the section 7.4.2.

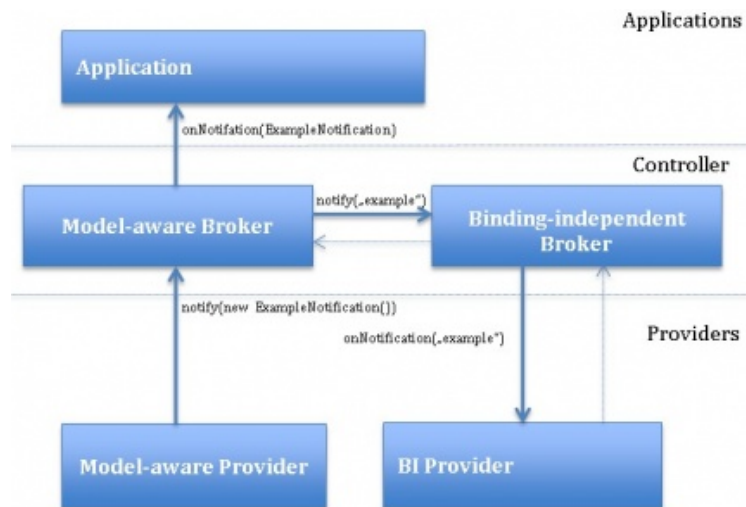


Figure 3.22: Notification routing between different data formats

### 3.8.1 Publishing a notification

Figure 3.23 illustrates how to publish a notification.



Figure 3.23: Publishing notifications

Notifications are published by calling the `putNotification()` method from the `NotificationPublishService`.

### 3.8.2 Subscribing for notifications

Figure 3.24 illustrates how to subscribe for receive notification.

In order to subscribe for receiving notifications, we have to implement `NotificationListener` and register our implementation in the `NotificationService` via `registerNotificationListener()` method. By implementing `NotificationListener`, we obtain the methods that will allow us to implement the processing of each of the defined notifications.

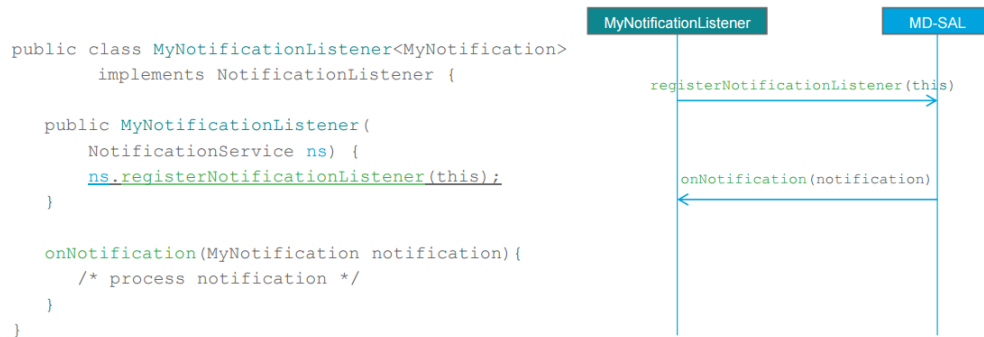


Figure 3.24: Subscribing for receiving notifications

## 3.9 RESTCONF requests

RESTCONF allows access to data stores in the controller. Request and response data can be either in XML or JSON formats, whose mapping from YANG models is well defined thanks to YANGTools project [8].

The media type of request data (input) has to be set via `Content-Type` field in the HTTP header.

The media type of response data (output) has to be set in the `Accept` field in the HTTP header.

Most of RESTCONF endpoints paths (URIs) use instance identifiers to point to the specific nodes to perform operations. Instance identifiers are formatted as `<identifier>` and have the following requirements to be met:

- Every URI must start with `<moduleName>:<nodeName>` where `<moduleName>` is name of the YANG module and `<nodeName>` is the name of a top level node in the module.
- If we have to point child nodes, it has to be specified in the following format:
  - We can use a single `<nodeName>` element, except in the case when a node with the same name is added via augmentation from external YANG model (Module A has node A1 with child X. Module B augments node A1 by adding node X). In this case, we also have to specify the module name: `<moduleName>:<nodeName>`.
  - `<moduleName>:<nodeName>:` is valid every time.
- When including a `<nodeName>` under another `<nodeName>` (for example, child of a container), they have to be separated by `/`.
- If the data node is a YANG list built-in type, its keys have to be defined behind the data node like this: `<nodeName>/<valueOfKey1>/<valueOfKey2>`.

### 3.9.1 Supported operations

- **OPTIONS:** returns the XML description of the resources with the required request and response media in Web Application Description Language (WADL).

URI link: `/restconf`

- **GET:** returns a data node from the config/operational data store. <identifier> points to the concrete data node whose data will be returned.

Config URI link: /restconf/config/<identifier>

Operational URI link: /restconf/operational/<identifier>

Figure 3.25 illustrates the GET procedure in action (config):

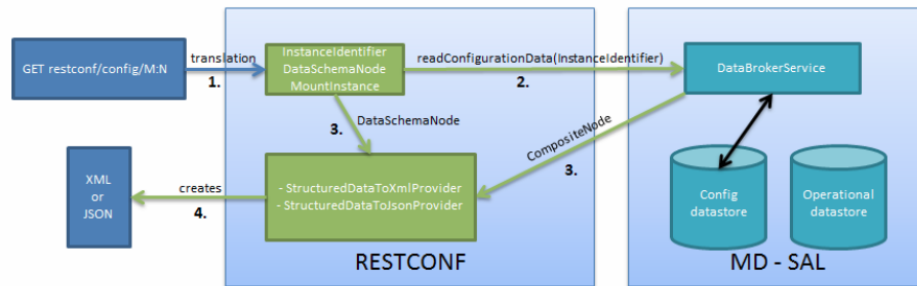


Figure 3.25: HTTP GET procedure in action

1. The request URI is translated into the InstanceIdentifier which points to the data node. During this translation, the DataSchemaNode that conforms to the data node is obtained.
2. RESTCONF asks for the value of the data node to the DataBrokerService based on the InstanceIdentifier.
3. DataBrokerService returns CompositeNode as data.
4. StructuredDataToXmlProvider or StructuredDataToJsonProvider is called based on the Accept field from the HTTP request. They transform the CompositeNode regarding DataSchemaNode to an XML or JSON document.
5. XML or JSON is returned as the answer on request from the client.

Figure 3.26 illustrates how to perform a HTTP GET operation to obtain operational data

```

http://localhost:8181/restconf/operational/toaster:toaster
{
  "toaster": {
    "toasterManufacturer": "Opendaylight",
    "toasterModelNumber": "Model 1 - Binding Aware",
    "toasterStatus": "up"
  }
}
  
```

Figure 3.26: HTTP GET request

- **PUT:** updates or creates a node in the config data store and returns the state about success. <identifier> points to the concrete data node which will be updated or created.

URI link: /restconf/config/<identifier>

Figure 3.27 illustrates the PUT procedure in action (config):

1. Input data is sent to JsonToCompositeNodeProvider or XmlToCompositeNodeProvider. The correct provider is selected based on the Content-Type field from the HTTP request. They transform input data to CompositeNode. However, this CompositeNode does not contain enough information for transactions.
2. The requested URI is translated into an InstanceIdentifier which points to the data node. DataSchemaNode conforming to the data node is obtained during this translation.

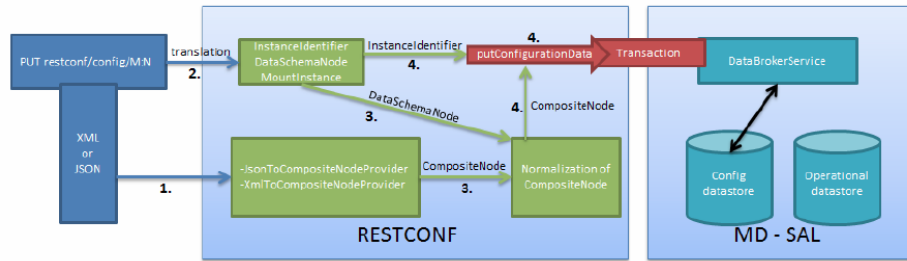


Figure 3.27: HTTP PUT procedure in action

3. CompositeNode can be normalized by adding additional information from the DataSchemaNode.
4. RESTCONF begins the transaction, and puts CompositeNode with the InstanceIdentifier into it. The response on the request from the client is the status code which depends on the result from the transaction.

Figure 3.28 illustrates how to perform an HTTP PUT operation to change the darknessFactor value.

```

http://localhost:8181/restconf/config/toaster:toaster
{
  "toaster": {
    "darknessFactor": "10"
  }
}

```

Figure 3.28: HTTP PUT request

- **POST:** there are different utilities from POST methods depending on the URI link.  
 URI link: /restconf/config/<identifier> Creates data if it does not exist in the config data store. <identifier> points to the concrete data node where data must be stored. The root element of data must have the namespace (if data is in XML) or module name (if data is in JSON).  
 URI link: /restconf/operations/<moduleName>:<rpcName> It is used to invoke RPCs. <moduleName> is the name of the module and <rpcName> is the name of RPC statement in this module. The root element of the data sent to RPC must have the name 'input'. The result can be the status code or the retrieved data having the root element as 'output'.

Figure 3.29 illustrates how to perform an HTTP POST operation to call the makeToast () RPC:

```

http://localhost:8181/restconf/operations/toaster:make-toast
{
  "make-toast": {
    "input": {
      "toasterDoneness": "10",
      "toasterToastType": "wheat-bread"
    }
  }
}

```

Figure 3.29: HTTP POST request

- **DELETE:** removes the data node in the config data store and returns the state about success. <identifier> points to a data node which must be removed.  
 URI link: /restconf/config/<identifier>



## Chapter 4

# Config subsystem

### 4.1 Overview

OpenDaylight's config subsystem provides two key features [8] [20]:

- An uniform way to express configuration, which can be read from configuration files updated at run-time.
- An uniform way to express requirements (dependencies) on other services.

The motivation of this system comes from the fact that almost all applications have to read some configuration information, which is usually done by means of configuration files (XML or properties files). When applications grow in complexity, is hard to manage configuration files without any formally-defined structure. For this reason, OpenDaylight includes a system that defines configuration structure and allows run-time configuration and automatic updating.

Moreover, most applications are dependent on other services. Dependencies can be hard to manage in complex projects, which may require multiple services. OpenDaylight applications, as are designed to run in a run-time environment, usually require multiple dependencies. Avoiding its directly embedding can be a key point to simplify its management. Expressing which other applications or services an application depends on, and whether if those dependencies are required or optional, is a critical part of assembling component-based systems like OpenDaylight.

OSGi's native dependency management can be used to solve some of these issues, but it have some drawbacks that are fixed by OpenDaylight's config subsystem:

- **Start-up sequence:** in OSGi, start-up is asynchronous, so it is hard to tell when the server is initialized, especially when you have to deal with dynamic bundle installation.
- **Dependency management:** with OSGi's dependency management, it is difficult to figure out when a dependency has fully loaded. Whereas config subsystem provides a relatively simple way to express such dependencies and to check if they are available.
- **Configuration:** OSGi framework provides configuration service and allows run-time module configuration. However, it still lacks a validation process, meaning that a new module configuration can negatively affect other modules.
- **Management interfaces:** config subsystem allows configuration management from NETCONF, RESTCONF and JMX (Java Management Extensions), which is a Java technology that supplies tools for managing and monitoring applications.

Config subsystem allows developers to:

- Expose the configuration and service dependencies of an application.
- Inject services and configuration to an application.
- Expose the run-time statistics of an application by using run-time beans (provided by JMX).
- Config subsystem handles all the required life-cycle management to instantiate applications.
- Package an application into one or more configurable modules (config modules will be defined in the following section).

It also enables operators and users to:

- Create new instances of applications at start-up or while the controller is running.
- Destroy and reconfigure running applications in a safe and transactional manner.
- Collect statistics from running applications.
- Invoke RPCs on module instances.

The config subsystem is activated by using OSGi activators and there is only one instance per controller JVM.

## 4.2 Configuration process

NOTE: sections detailing configuration process assume a basic knowledge of Maven and YANG, so it is highly recommended to previously read Maven (chapter 6) and YANG (chapter 7) chapters.

The following sections will detail the steps that are required to instantiate an OpenDaylight application, previously ensuring that the controller has everything prepared to handle its dependencies [20] [21].

### 4.2.1 Modules and services

An application interacts with the config subsystem by defining:

- **Modules:** config subsystem declares as a module to those YANG schemas that define the implementations of service/data interfaces by providing one or more services. Application service instantiation can be carried out by one or multiple config modules.

A config module:

- Defines a sort of configurations (and dependencies) required to run an instance of its defined services. The initial values for those are provided by a customized XML file that corresponds to the schema defined in the YANG file (will be explained in the section 4.2.8).
- It is responsible to instantiate and wire (injecting its required capabilities) the defined services.

Modules also allow to model read-only state data provided from an application (e.g. statistics) by defining run-time beans. A module can have zero or more run-time beans.

At run-time, a module is instantiated and inserted under the config modules/module hierarchy.

- **Services:** a service is a YANG model that describes data interfaces or certain services. Any module can implement or provide multiple services. Module configuration can be specified for each particular service instance. At run-time, a service is instantiated and inserted under the config services/service hierarchy.

## 4.2.2 Declaring an application aware of the config subsystem

The procedure that is followed to declare an application aware of the config subsystem (assuming that is ready to be included) is:

1. We have to write YANG models that describe the service/data interfaces (services) and its implementations (config modules) of our application.

- Each YANG model should contain the definition of one or more services and one or more config modules for those services.

Procedure to declare a config module:

- I Under the YANG model, we declare an identity that will be used to declare and reference our config module.

---

```
identity moduledeclaration {
    base config:module-type;
}
```

---

- II `base config:module-type` statement, is used to declare our identity as a config module. Note that it points to the external config YANG module.

- YANG models that are used in the config subsystem need to import *config.yang*, which contains the base definitions of the config subsystem (OpenDaylight's Controller project). Therefore, we need to add a dependency to *config-api* on the Maven's *pom.xml* file of our project.

In the YANG file:

---

```
import config { prefix config; revision-date 2013-04-05; }
```

---

In the project's *pom.xml* file:

---

```
<dependency>
  <groupId>org.opendaylight.controller</groupId>
  <artifactId>config-api</artifactId>
  <version>0.4.0-SNAPSHOT</version>
</dependency>
```

---

2. We have to configure the *yang-maven-plugin*, in the *pom.xml* file, to support JMX code generation, which will be used to manage configuration.

In the *yang-maven-plugin*, we add a code generator:

---

```
<generator>
  <codeGeneratorClass>org.opendaylight.controller.config.yangjmxgenerator.plugin.
  JMXGenerator</codeGeneratorClass>
  <outputBaseDir>${jmxGeneratorPath}</outputBaseDir>
```

---

```
<additionalConfiguration>
  <namespaceToPackage1>urn:opendaylight:params:xml:ns:yang:controller==org.opendaylight.
    controller.config.yang</namespaceToPackage1>
</additionalConfiguration>
</generator>
```

---

3. Different classes will be generated for each declared module:

- **ModuleFactory:** a singleton factory class, which is responsible of the instantiation of the module that declares, having one factory per defined module. For example, if our module is declared under `toaster` identity, it will generate a `ToasterModuleFactory` class.
- **<Module>:** JMX compatible wrapper class that wraps an instance of a whole application or its part (the services provided by the module) by using its `createInstance()` method, which users are expected to implement. In the previous example, a `ToasterModule` class would be generated.
- **Additional classes:** they are not meant for developers since they are re-generated with every build. Details in the following sections.

4. Implement the `createInstance()` method inside each module class to instantiate the application. It is normally implemented by using getters and registrations to retrieve the dependencies and configuration needed for the application instantiation.

## Providing state data

If we want our module to provide state data, we need to declare a configuration augment in our module's YANG schema, specifying the data that must be observed.

State data nodes are inserted under the `modules/module/state` hierarchy.

---

```
augment "/config:modules/config:module/config:state" {
  case data-provider-impl {
    when "/config:modules/config:module/config:type = 'data-provider-impl'";
    leaf packetssent {
      type uint32;
    }
  }
}
```

---

In this example, assuming that we have a module named `data-provider-impl`, we declare the `packetssent` leaf as state data to be observed.

The construction composed by 'augmentation', 'case' and 'when' statements indicates that we are implementing state data that is only valid for our module:

- **'augmentation' statement:** it indicates that the contents we define will be inserted under the `modules/module/state` hierarchy. In other words, it indicates that we are defining state data.
- **'case' and 'when' statements:** they indicate that the state data that we defined is only valid for our `data-provider-impl` module.

Now, different classes will be generated (its generation is performed in the same way defined before):

- **RootRuntimeBeanRegistrator:** every module that provides state data must be registered using this class.
- **RuntimeBean:** contains getters and setters (or JMX RPC methods) for the state data modelled in the YANG schema.

In the `createInstance()` method, we have to register our implementation of `RuntimeMXBean` using the injected instance of `RuntimeBeanRegistrator`:

---

```
public java.lang.AutoCloseable createInstance() {  
  
    ...  
  
    final DataProviderImplRegistration runtimeReg = getRootRuntimeBeanRegistratorWrapper().  
        register(provider);  
  
    ...  
}
```

---

### 4.2.3 Config subsystem models discovery

After the bundles with the application and config subsystem bindings are ready, they need to be included in the OpenDaylight distribution and picked up by the config subsystem at start-up.

Config subsystem detects 'config subsystem aware' bundles by:

1. Config-manager (and its activator) starts a bundle tracker and listens to 'Bundle is active' elements.
  - By listening on active, config subsystem ensures that all the dependencies (other bundles) of a bundle are already resolved.
  - If all the dependencies are resolved, all YANG model dependencies are resolved as well, since the models are part of those dependencies.
2. For every new bundle it looks for generated module factories.
  - A single text file is used to find the generated module factories:  
*META-INF/services/org.opendaylight.controller.config.spi.ModuleFactory.*
  - The file contains the fully qualified names of all the factories inside the bundle.
  - The file is added by the config subsystem code generator while generating its bindings.
3. Config-manager creates and stores a singleton instance of each module factory.
  - The instance will be later used to create one or more module instances.
  - Factory instances are registered in the OSGi service registry.
  - ConfigRegistry is responsible for managing factories and their instances.
  - ConfigRegistry itself is also exposed into OSGi.

## 4.2.4 Application instantiation

After the distribution is fully started and all the bundles are scanned, it is possible to instantiate the available modules using RESTCONF, NETCONF or JMX.

1. ConfigRegistry is the core class of config-manager and is responsible for transactional configuration management. The application instantiation is considered as a single configuration change.
2. Transaction from ConfigRegistry is started.
3. A new module instance is created in the transaction (using the stored instance of the module factory for that particular module).
4. Configuration attributes and dependencies are set for the new module instance. Note that they are only set and not injected, since they still have to be checked to ensure that they are not unhealthy for other bundles.
5. Transaction resolves and validates the dependencies and other configuration parameters.
  - Users can provide additional validation of configuration attributes by using the `customValidation()` method inside the generated module class.
  - Dependencies are checked for correct dependency type in the module and the dependency graph is searched for loops or missing dependencies (responsibility of DependencyResolver in the config-manager).
6. Transaction is committed.
  - All configuration attributes and dependencies are injected to the module instance.
  - Method `createInstance()` is called.
  - User code inside `createInstance()` method should instantiate the application and pass all the relevant attributes and dependencies to it.
  - Returned instance is registered into JMX, so it can be modified, and optionally into OSGi service registry.

As application instantiation is considered as a configuration change, it follows the following procedure (illustrated in the figure 4.1):

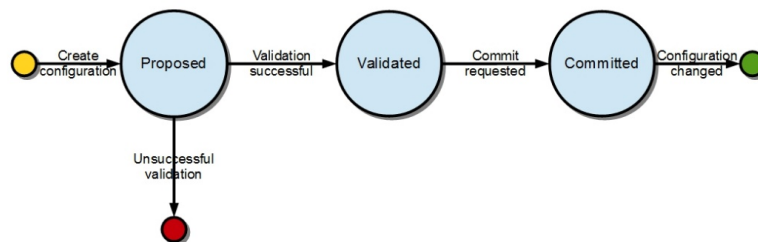


Figure 4.1: Configuration change process

- First, a proposed configuration is created, which target is to replace the old configuration.
- Then, the proposed configuration is validated. If it successfully passes the validation, the proposed configuration state will be changed to 'validated'.

- Finally, a validated configuration can be committed. The affected modules are reconfigured.

In fact, each configuration operation is wrapped in a transaction. Once a transaction is created, it can be configured, that is to say, a user can abort the transaction during this stage. After the transaction configuration is done, it is committed to validation stage. In this stage, the validation procedures are invoked. If one or more validations fail, the transaction can be reconfigured. Upon success, the second phase commit is invoked. If this commit is successful, the transaction enters last stage, committed. After that, the desired modules are reconfigured. If the second phase commit fails, it means that the transaction is unhealthy and is dropped. Transaction process is represented in the figure 4.2.

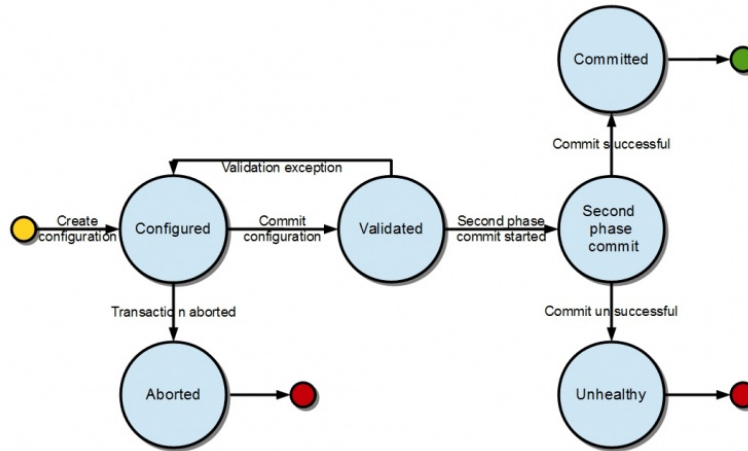


Figure 4.2: Configuration change as transactions

#### Validation:

To secure the consistency and safety of a new application and avoid conflicts, the configuration validation process is necessary. Usually, validation checks the input parameters of a new configuration and mostly verifies module-specific relationships. The validation procedure results in a decision indicating whether the proposed configuration is healthy or not.

#### Dependency management:

Since there can be dependencies between modules, a change of a module configuration can affect the state of other modules. Therefore, we need to verify whether dependencies on other modules can be resolved. The DependencyResolver acts similar to dependency injectors, basically builds a dependency tree.

### 4.2.5 Wiring a YANG schema to OpenDaylight

So far, we have defined how to instantiate an application that provides services, which are based on the generated service and data interfaces modelled from YANG schemas.

But, how OpenDaylight becomes aware of those YANG schemas? How are they deployed in the data stores? The config subsystem also provides an answer to these questions. When generating files from a YANG schema, a part of the DTOs and Java interfaces generated at compile-time, other run-time files are generated: `$YangModuleInfoImpl` and `$YangModelBindingProvider`, which are responsible to wire the YANG schemas to the MD-SAL.

1. Every YANG module is automatically copied to *META-INF/yang/\*.yang*, where *\** is the name of the YANG file.

2. The config subsystem contains a class, `ModuleInfoBundleTracker`, which scrapes the *META-INF/services/org.opendaylight.yangtools.binding.YangModelBindingProvider* resources from bundles on start-up and reads the class names defined in the file.
3. For each `YangModelBindingProvider` class specified, the MD-SAL creates an instance and calls the `getModuleInfo()` method to return a singleton `YangModuleInfoImpl` instance.
4. `YangModuleInfoImpl` has methods to obtain static configuration about the YANG module, e.g. name, revision, imports, etc.
5. It also contains a `getModuleSourceStream()` method that provides an input stream to the *META-INF/yang/\*.yang* file.
6. Once the MD-SAL knows about a YANG module and its definitions, it can wire it up to RESTCONF and other parts of the system.

## 4.2.6 Application reconfiguration

After a module is successfully instantiated, it can be reconfigured at any point. Reconfiguration process is very similar to instantiation:

1. Transaction from `ConfigRegistry` is started.
2. The configuration of a module (identified by its type and name) is changed.
3. A new module instance is created in the transaction.
4. Method `createInstance()` is not called yet.
5. Config subsystem decides whether we it can reuse an old instance of the module or not.
6. If the old instance can be reused, then the old instance of the module is reused and the new one is abandoned. The transaction is committed at this point.
7. If the old instance cannot be reused, then the old module is abandoned and itself and its wrapped instance (the services it manages) is closed. Then, the `createInstance()` method is called for the new module instance, and the process continues as described in the previous section.

## 4.2.7 Pushing initial configuration

In order to automate the start-up of infrastructural services, plugins and applications, a config-pusher mechanism was implemented, which is part of the config-persister component.

Initial configuration push consists of a few edit-config RPCs sent to the northbound NETCONF interface for the config subsystem. It does not differ from users sending edit-configs via NETCONF. Initial configuration files are described in XML format, as it is the native format of NETCONF RPCs and allows a trivial construction.

There are two ways of configuring the controller:

- **Using `config.ini`:** it is a property file that is used to pass configuration properties to OSGi bundles besides the config subsystem.
- **Using the config-persister:** it allows to push initial configuration for modules managed by the config subsystem.



## Using property file

The *config.ini* property file can be used to provide a set of properties for any OSGi bundle deployed to the controller. It is usually used to configure bundles that are not managed by the config subsystem. For this reason, it will not be explained in this document.

## Using the config-persister

Config-persister is a default service in the controller, and is automatically started by using the OSGi activator. Its purpose is to load the initial configuration for the config subsystem and store a snapshot for every new configuration state pushed by external clients. It retrieves the base configuration from the *config.ini* file, and tries to load the configuration for the config subsystem.

Config-persister as a NETCONF client, it pushes configuration to the config subsystem as a set of edit-config NETCONF RPCs followed by a commit RPC.

Config-persister life-cycle, illustrated in the figure 4.3, consists of following phases:

1. Config-persister service is started at start-up.
2. It retrieves the base configuration of adapters from the *config.ini* property file
3. Initializes the backing storage adapters.
4. Initializes the NETCONF client, and connects to the NETCONF endpoint of the config subsystem.
5. Loads the initial configuration snapshots from the latest storage adapter.
6. Sends the edit-config RPCs with the initial configuration snapshots.
7. Sends the commit RPC.
8. Listens for any change to the configuration and persists a snapshot.

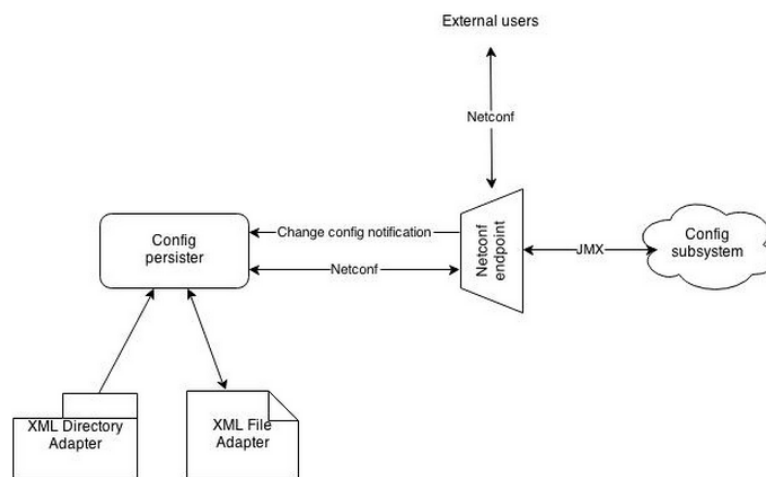


Figure 4.3: Config-persister life-cycle

The *config.ini* property file contains the configuration for the config-persister. It initializes two adapters: **XML directory adapter** and **XML file adapter**.

#### XML directory adapter:

It loads the initial configuration from the *etc/configuration/opendaylight/karaf* folder. These files will be pushed as the initial configuration for the config subsystem. Since this adapter is read only, it will not store any configuration snapshot during the controller life-cycle.

Lithium controller contains multiple configuration files that might depend on others; their resolution order can be ensured by the sorting the file names in ascending order. This means that every configuration file will have the visibility of the modules from all previous configuration files via aliases. You basically want to order the configuration files such that its dependencies are deployed first.

For this reason, the internal MD-SAL files, which most of the modules depend on, are named as: *00-netty.xml* and *01-md-sal.xml*.

- **Netty configuration file:** instantiates netty utilities, which will be used by the controller components that internally use netty. Netty is a client-server framework for the development of Java network applications such as protocols and clients.
- **MD-SAL configuration file:** snapshots different MD-SAL modules. After the controller is started, all these modules will be installed and configured. They can be further referenced as dependencies for new modules, reconfigured or even deleted. MD-SAL modules are considered to be the base configuration for controller. For this reason, they are automatically configured to simplify the process of controller's configuration. By doing this, you ensure that they are always available for use. Some of the defined modules are the binding-aware and DOM brokers, together with its capabilities.

Technically, any configuration could be deployed before than others, since the config subsystem will keep retrying if a dependency is not yet present. However, it is more efficient to explicitly define the ordering.

#### XML file adapter:

It stores snapshots of the current configuration after any change in the *controller.current.config.xml* file, located under the *etc/configuration/opendaylight* folder. It only keeps one snapshot backup, so every new change overwrites the previous one.

After every configuration change, a notification is sent to the config-persister with the current snapshot to store. Notifications are generated in the NETCONF interface, carried over JMX and sent to the config subsystem (config-persister). They are generated in the NETCONF interface as it keeps a snapshot of current configuration in XML format.

However, if users directly interact with the config subsystem without using NETCONF, the changed configuration will not be stored in the config-persister, as it will not be notified by NETCONF interface.

## 4.2.8 Customizing initial configuration

When we are developing an application, we usually require to push some modules from the start that our application depends on (MD-SAL brokers), or to define a customized configuration for specific components. These behaviour can be modelled using customized configuration files.

There are multiple ways to add custom initial configuration to the controller:

- Manually creating the configuration file and putting it in the initial configuration folder of the XML directory adapter.
- Reconfiguring the running controller. The XML file adapter will store the current snapshot and, on the next controller start-up, it will load the configuration containing the changes.

OpenDaylight projects, usually customize their initial configuration by creating its own configuration files. For this reason, it is the procedure that will be detailed in this section.

Configuration files are XML documents that are divided in two major sections:

- **Required-capabilities:** it defines the needed configuration for the NETCONF “hello” message (to establish connection) and describes the YANG capabilities that are required to push the configuration defined under the <configuration> tag. The config-persister will not push the configuration until the NETCONF endpoint for the config subsystem reports that all the needed capabilities are active. Every YANG model that is referenced within the configuration file must be referenced as capability in this list.
- **Configuration:** contains the configurations to be pushed by the config subsystem. It is wrapped in a <data> tag with the base NETCONF namespace. The whole <data> tag, with all its child elements, will be inserted into the edit-config RPCs. Here is where we define our service and implementation modules, together with its required dependencies.

For example, we create a *toaster-impl-config.xml* configuration file:

---

```
<snapshot>
  <required-capabilities>
    <capability>urn:opendaylight:params:xml:ns:yang:toaster:provider:impl?module=toaster-
    provider-impl&revision=2014-12-10</capability>
    <capability>urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding?module=opendaylight
    -md-sal-binding&revision=2013-10-28</capability>
  </required-capabilities>
  <configuration>
    <data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <modules xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
        <module>
          <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:toaster:provider:impl">
            prefix:toaster-provider-impl</type>
          <name>toaster-provider-impl</name>
          <binding-aware-broker>
            <type xmlns:binding="urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">
              binding:binding-broker-osgi-registry</type>
            <name>binding-osgi-broker</name>
          </binding-aware-broker>
        </module>
      </modules>
    </data>
  </configuration>
</snapshot>
```

---

We assume that our module, *toaster-provider-impl*, implements RPCs and notifications, so it depends on the binding-aware broker to provide its services.

Under the `<configuration>` element we specify our modules, with the configuration to be pushed and its required dependencies (`<module>` element), and the provided services (`<service>` element). In this case, we only need to specify our `toaster-provider-impl` module and its dependency `binding-osgi-broker`.

In the `<required-capabilities>` section, we specify the YANG models that are referenced in the `<configuration>` element, which are our `toaster-provider-impl.yang` and the `opendaylight-md-sal-binding.yang`, which contains the configuration for the binding-aware broker.

There might be other dependent modules, for example the YANGTools modules that provide code generation and data translation. However, most of the dependent modules are commonly inferred by imports in the YANG files that are exposed to the config subsystem, so there is no need to explicitly declare them in the configuration file. This is very useful in the dependency management as we avoid its direct embedding.

After our custom configuration file is done, we have to push it. In order to do this, we have to define it in the POM file of the config-pusher, which is located under the `controller/opendaylight/commons/opendaylight` folder.

---

```
<properties>
...

<config.toaster.configfile>toaster-provider-config.xml</config.toaster.configfile>

...

</properties>
```

---

## 4.3 Config subsystem APIs and SPIs definitions

This section describes the config subsystem APIs and SPIs (Serial Peripheral Interface).

### 4.3.1 SPIs

- **Module:** `org.opendaylight.controller.config.spi.Module`, represents one or more services that are to be configured. It holds configuration attributes, validates them, and creates instances of its defined services by using its `createInstance()` method.
- **ModuleFactory:** `org.opendaylight.controller.config.spi.ModuleFactory`, creates an instance of a concrete module by using its `createModule()` method, which returns a module identified by `InstanceName` and injects the dependencies to resolve it. A `ModuleFactory` instance needs to be exported into the OSGi service registry since it provides the metadata describing the services that can be published from it.

### 4.3.2 APIs

- **ConfigRegistry:** `org.opendaylight.controller.config.api.ConfigRegistry`, it provides functionality for creating and committing config transactions.
- **ConfigTransactionController:** `org.opendaylight.controller.config.api.ConfigTransactionController`, represents functionality provided by a configuration transaction (create and destroy a module, validate or abort transactions).

- **RuntimeBeanRegistrarAwareModule:** `org.opendaylight.controller.config.api.RuntimeBeanRegistrarAwareModule`, interface that has to be implemented by a module that makes use of run-time beans. Thus, that module will receive `RootRuntimeBeanRegistrar` before its `getInstance()` method is invoked.

### 4.3.3 Run-time APIs

- **RuntimeBean:** `org.opendaylight.controller.config.api.runtime.RuntimeBean`, it is a marker interface for all run-time beans. It contains the getter and setter methods of the defined state data.
- **RootRuntimeBeanRegistrar:** `org.opendaylight.controller.config.api.runtime.RootRuntimeBeanRegistrar`, it is the entry point for run-time bean functionality. Allows for registering a run-time bean, which subsequently allows hierarchical registrations.
- **HierarchicalRuntimeBeanRegistration:** `org.opendaylight.controller.config.api.runtime.HierarchicalRuntimeBeanRegistration`, it provides the hierarchical registration and unregistration, in the modules/module/state hierarchy, for a run-time bean.

### 4.3.4 JMX APIs

JMX APIs are purposed as a transition between the client APIs and the JMX platform.

- **ConfigTransactionControllerMXBean:** `org.opendaylight.controller.config.api.jmx.ConfigTransactionControllerMXBean`, it extends the `ConfigTransactionController`, using this interface as a proxy.
- **ConfigRegistryMXBean:** `org.opendaylight.controller.config.api.jmx.ConfigRegistryMXBean`, represents the entry point of configuration management. It extends `ConfigRegistry`, using this interface as a proxy.
- **ObjectNameUtil:** `org.opendaylight.controller.config.api.jmx.ObjectNameUtil`, provides the `ObjectName` creation. `ObjectName` is the pattern used in JMX to locate JMX beans. It consists of a domain and key properties, domain is defined as `org.opendaylight.controller`.



## Chapter 5

# OSGi and Karaf

### 5.1 Open Services Gateway Interface

#### 5.1.1 Overview

OSGi (Open Services Gateway interface) manifests a set of specifications, reference implementations, and test suites around a dynamic module system for Java [22].

It defines an architecture for modular application development that adds a framework for dynamic loading/unloading/starting/stopping of Java modules without bringing down the entire JVM platform. Figure 5.1 represents OSGi's overall architecture [23].

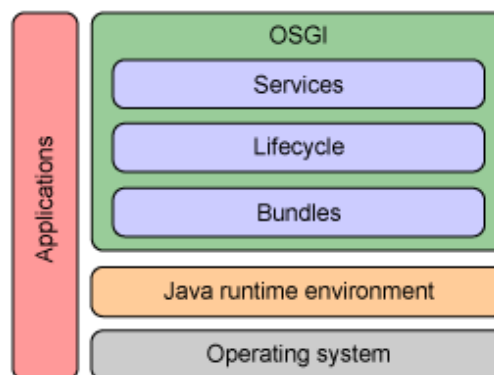


Figure 5.1: OSGi architecture

- **Bundles:** they are the unit of work in the OSGi environment, applications are defined as a collection of bundles.
- **Life-cycle:** framework that defines a sequence of steps that bundles go through when installed, started, updated, stopped and uninstalled.
- **Services:** module that facilitates dynamic interactions between different bundles.
- **Java run-time environment:** environment in which the OSGi architecture is deployed. In our case, the Open-Daylight controller.

OSGi describes an application as a collection of bundles, which are a group of Java classes and additional resources (e.g. metadata) detailed in a *MANIFEST.MF* file. Manifest files are used to self-describe bundles, declare their public APIs, define their run-time dependencies on other bundles, and hiding their internal implementations. In other words, bundles are implemented as JAR files, but with identity and dependency information. Figure 5.2 illustrates how bundles will be generated in our OpenDaylight applications. Maven will be detailed in the chapter 6.

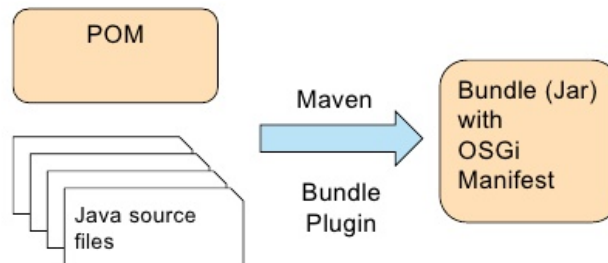


Figure 5.2: Bundle generation using Maven

The idea behind this, is to define a service platform where executing bundles are independent of each other, yet they can collaborate in well-defined ways. Bundle writers, also known as providers, create bundles and make them available for others to use. Application writers, or consumers, use these bundles.

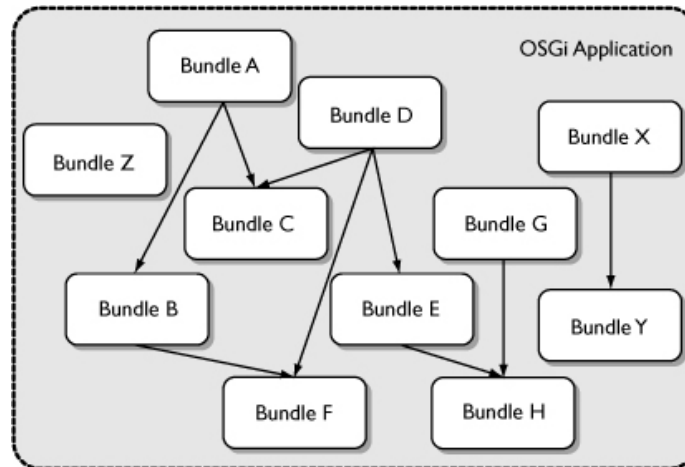


Figure 5.3: OSGi application

As shown in the figure 5.3, an OSGi application has no top and no bottom; it is simply a collection of bundles. There is also no main program; some bundles contribute code libraries, others start threads, communicate over the network, etc. While there are often dependencies between bundles, in many cases bundles are peers in a collaborative system.

By default, the JAR packages in a bundle are hidden from other bundles. Packages containing APIs must, by definition, be available to other bundles and so must be explicitly exported. This is done in the manifest file, where a bundle define what is to be exported to other bundles, and what needs to be imported from other bundles. Communication between bundles is detailed in the figure 5.4.

Summarizing, OSGi can be thought of as an extension to the Java programming language that allows package visibility and package dependency constraints to be specified at development time and enforced at run-time, which allow an easier development of applications.



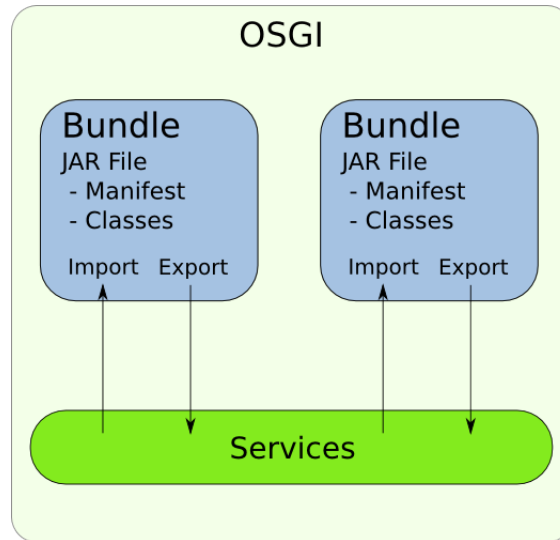


Figure 5.4: Communication between bundles

### 5.1.2 Service model

To facilitate the dynamic communication between bundles, OSGi defines a service model. A service is defined as an object that implements an specific contract and is registered with the OSGi service registry, so it can be available to other bundles. Bundles looking to use that service, only need to import the package defining the contract and discover the service implementation in the service registry.

Services are dynamic in nature, a bundle dynamically registers and unregisters services that it provides, and also can acquire and release the services that it consumes. Service interaction is illustrated in the figure 5.5

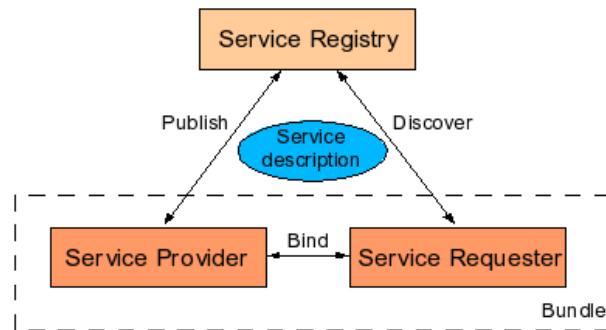


Figure 5.5: Service interaction

### 5.1.3 Deployment life-cycle

OSGi framework provides mechanisms to support continuous deployment activities. These deployment activities include installation, removal, update, starting (activation) and stopping of bundles. Once a bundle is installed in the platform, it can be activated if its dependencies are already activated.

Figure 5.6 shows the bundle life-cycle.

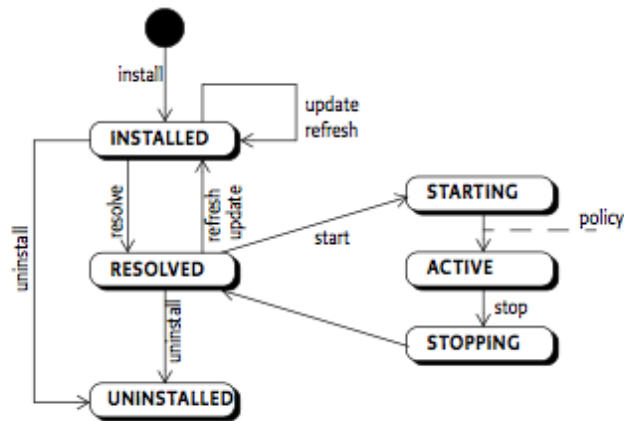


Figure 5.6: Deployment life-cycle

- **Installed:** OSGi run-time is aware that the bundle is there.
- **Resolved:** bundle dependencies are available, so it can be started.
- **Starting:** BundleActivator starts the bundle.
- **Active:** the bundle is running, so it can be accessed.
- **Stopping:** BundleActivator stops the bundle. The bundle returns to the Resolved phase.
- **Uninstalled:** The bundle has been removed from the OSGi run-time.

BundleListener is responsible for listening to life-cycle events, e.g. when an user calls to stop/run a bundle.

## 5.2 Karaf

### 5.2.1 Overview

Apache Karaf is a small OSGi based run-time which provides a lightweight container onto which various components and applications can be deployed [24].

In OSGi, a bundle can depend on other bundles, meaning that to deploy an OSGi application, most of the time, you have to firstly deploy a lot of other bundles required by the application. Karaf provides a simple and flexible way to provision applications, by introducing the 'feature' definition.

Short list of features supported by the Karaf:

- **Hot deployment:** Karaf supports hot deployment of OSGi bundles by monitoring JAR files. Each time a JAR is copied in Karaf folder, it will be installed inside the run-time OSGi framework. You can then update or delete it, and changes will be handled automatically.
- **Dynamic configuration:** Services are usually configured through the ConfigurationAdmin OSGi service. Such configuration can be defined in Karaf using property files. These configurations are monitored and changes on the properties files will be propagated to the services.

- **Logging System:** using a centralized logging back end supported by Log4J, Karaf supports a number of different APIs (JDK 1.4, JCL, SLF4J, Avalon, Tomcat, OSGi).
- **Provisioning:** provisioning of libraries or applications can be done through a number of different ways, by which they will be downloaded locally, installed and started.
- **Native OS integration:** Karaf can be integrated into your own Operating System as a service so that the life-cycle will be bound to your Operating System.
- **Extensible Shell console:** Karaf features a nice text console where you can manage the services, install new applications or libraries and manage their state. This shell is easily extensible by deploying new commands dynamically along with new features or applications.
- **Remote access:** use any SSH client to connect to Karaf and issue commands in the console
- **Security framework:** based on JAAS.
- **Managing instances:** Karaf provides simple commands for managing multiple instances. You can easily create, delete, start and stop instances of Karaf through the console.

Figure 5.7 shows the different Karaf modules.

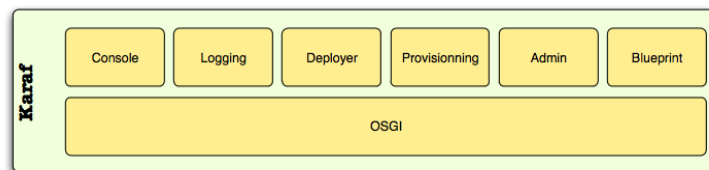


Figure 5.7: Karaf modules

## 5.2.2 Definition of a Karaf feature

A feature describes an application as a set of attributes, defined in the *features.xml* file, which contains:

- **Features identifier:** name, version, description.
- **OSGi bundles:** which contain the application itself.
- **Configuration files:** it is optional, it can expose a set of conditions that have to be available before deploying our feature.
- **Dependant features:** it is optional, describe other applications in which our application depends on.

Figure 5.8 illustrates the contents that *figures.xml* may have.

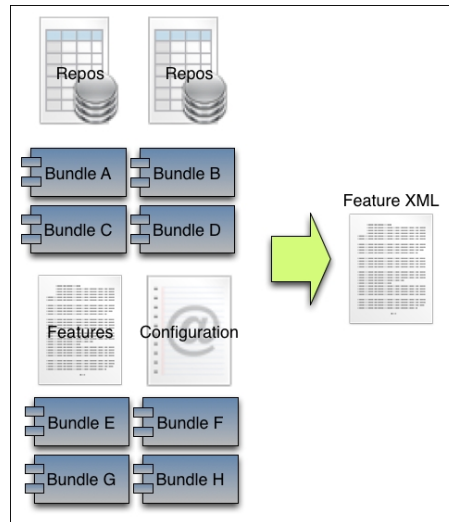


Figure 5.8: Elements to be declared in our Karaf feature

Structure of the *feature.xml* file:

---

```
<features name="odl-toaster-${project.version}" xmlns="http://karaf.apache.org/xmlns/features/
v1.2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://karaf
.apache.org/xmlns/features/v1.2.0 http://karaf.apache.org/xmlns/features/v1.2.0">

  //Repositories where dependant features are placed
  <repository>mvn:org.opendaylight.yangtools/features-yangtools/{VERSION}}/xml/features</repository>
  <repository>mvn:org.opendaylight.controller/features-mdsal/{VERSION}}/xml/features</repository>
  <repository>mvn:org.opendaylight.netconf/features-restconf/{VERSION}}/xml/features</repository>

  //Other features in which our project depends on
  <feature name='odl-toaster-api' version='${project.version}' description='OpenDaylight ::
  toaster :: API'>
    <feature version='${yangtools.version}'>odl-yangtools-common</feature>
    <feature version='${yangtools.version}'>odl-mdsal-binding-base</feature>
    <feature version='${controller.restconf.version}'>odl-restconf</feature>
    <bundle>mvn:org.opendaylight.toaster/toaster-api/{VERSION}</bundle>
  </feature>

  <feature name='odl-toaster-impl' version='${project.version}' description='OpenDaylight ::
  toaster :: Impl'>
    <feature version='${controller.mdsal.version}'>odl-mdsal-broker</feature>
    <feature version='${controller.restconf.version}'>odl-restconf</feature>
    <feature version='${project.version}'>odl-toaster-api</feature>
    <bundle>mvn:org.opendaylight.toaster/toaster-impl/{VERSION}</bundle>
    <configfile finalname="toaster-impl-config.xml">mvn:org.opendaylight.toaster/toaster-impl/
    {{VERSION}}/xml/config</configfile>
  </feature>
</features>
```

---

Our project is declared as a major feature named `odl-toaster`, which is composed by different repositories and other features (subprojects):

- **odl-toaster-api:** it declares three features that correspond to its dependencies: YANGTools commons and binding, and RESTCONF. Also, it declares a bundle that contains the toaster-api project.
- **odl-toaster-impl:** it declares three features that correspond to its dependencies: binding aware broker, RESTCONF and toaster API. Also, it declares a bundle that contains the toaster-impl project. Under the `<configfile>` statement, we specify that the bundle contains an initial configuration file.

### 5.2.3 Karaf shell console

Karaf features a nice text console where you can manage the services, install new applications or libraries and manage their state. This shell is easily extensible by deploying new commands dynamically along with new features or applications.

To access Karaf console in our OpenDaylight distribution (installation detailed in the section 2.6):

---

```
> unzip distribution-karaf-0.3.0-Lithium.zip    //Unzip the zip file
> cd distribution-karaf-0.3.0-Lithium          //Navigate to the directory
> ./bin/karaf                                  //Run Karaf
```

---

Figure 5.9 shows OpenDaylight Karaf console.

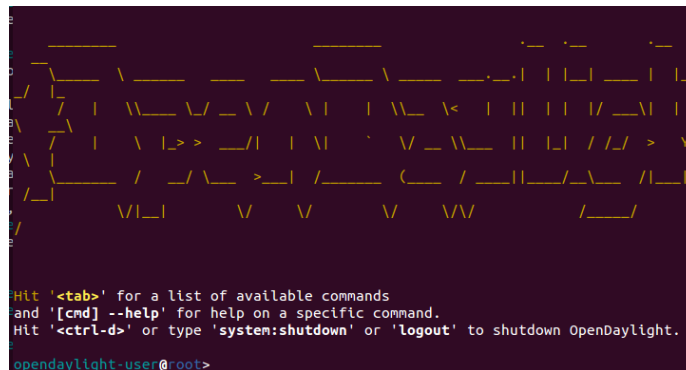


Figure 5.9: OpenDaylight Karaf console

To install a feature we can use the following command:

```
> feature:install <feature-name>
```

The syntax for this command allows multiple features to be installed in one line, for example:

```
> feature:install <feature1-name> <feature2-name> ... <featureN-name>
```

To find a complete list of Karaf features, run the following command:

```
> feature:list
```



# Chapter 6

## Maven

The aim of this section is to introduce Maven as the project management tool used by OpenDaylight. This chapter will focus on Maven basics and how it is used in ODL. Developer guides, as well as tutorials, can be found at Maven official website [25].

### 6.1 Introduction

Maven is an attempt to apply patterns to a project's build infrastructure in order to promote comprehension and productivity. It is a project management tool that provides a way to help with managing:

- Builds
- Documentation
- Reporting
- Dependencies
- Source Code Management (SCM)
- Releases
- Distribution

Maven is focused on the simplicity of both creation and management fields that covers all the build-oriented phases defined in the Application Lifecycle Management (ALM):

- Build management
- Testing
- Releasing
- Versioning
- Deployment

## 6.2 Installation and configuration

This section contains very detailed instructions for installing Maven 3 in Linux operating system, as well as its integration with Eclipse IDE [26].

### 6.2.1 Maven installation

The first step is to verify the current Java version, as Maven currently requires the usage of Java 7 or higher. In fact, Java 7 is also a pre-requisite for running ODL platform, so it should be already available.

We will use the following command to check the current Java version:

```
> java -version
```

We should receive an output similar than this:

---

```
> java -version
java version "1.7.0_71"
Java(TM) SE Runtime Environment (build 1.7.0_71-b14)
Java HotSpot(TM) 64-Bit Server VM (build 24.71-b01, mixed mode)
```

---

Once we have checked that our Java version is supported, we need to download Maven 3+ from the download section in Maven official website [27].

We have to download the Maven package (.tar or .zip) and choose an appropriate place for it, then we expand the archive there.

For example, if we expanded the archive into the directory /opt/apache-maven-3.2.5, we may want to create a symbolic link to make it easier to work with and to avoid the need to change any environment configuration when you upgrade to a newer version:

---

```
> cd /opt
> ln -s apache-maven-3.2.5 maven //creating a symbolic link
> export PATH=/opt/maven/bin:$PATH
```

---

Once Maven is installed, we need to do a couple of things to make it work correctly. We need to add its bin directory in the distribution (in this example, /opt/maven/bin) to our command path.

We will need to add PATH to a script that will run every time you login. To do this, we add the following line to *.bash\_login* or *.profile* files (located in the home directory):

```
export PATH=/opt/maven/bin:$PATH
```

Once we have added the PATH to our own environment, we will be able to run Maven from the command line.

We can check if the installation was successfully by running the following command:

```
> mvn -v
```

A similar output should be printed:



---

```
> mvn -v
Apache Maven 3.2.5 (12a6b3acb947671f09b81f49094c53f426d8cea1; 2014-12-14T09:29:23-08:00)
Maven home: /opt/apache-maven-3.2.5
Java version: 1.7.0_71, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.7.0_71.jdk/Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.8.5", arch: "x86_64", family: "mac"
```

---

## 6.2.2 Maven integration

This section will detail how to integrate Maven with Eclipse, which is used as the tool-set for application development.

We need first to download Eclipse from its official website [28]. Once it is downloaded and installed, we have to add the Maven software [29]. In Eclipse, under Help > Install New software section, we have to add the following path to install Maven:

```
http://download.eclipse.org/technology/m2e/releases
```

Figure 6.1 illustrates where to add the path and how to proceed.

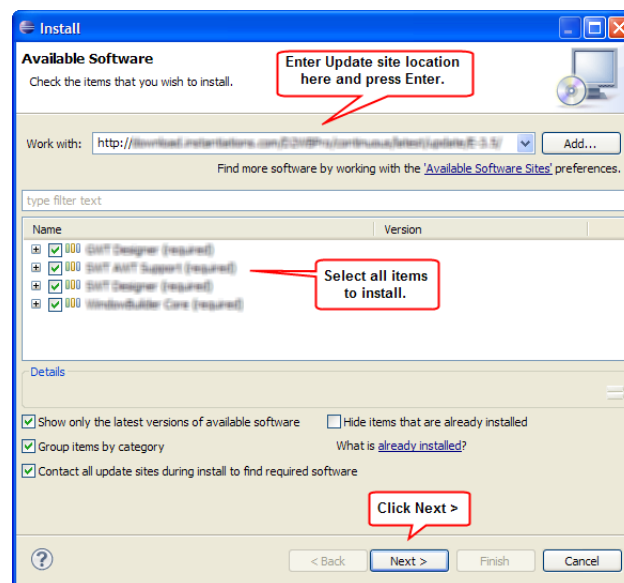


Figure 6.1: Maven and Eclipse integration

## 6.3 Introduction to the POM

Maven models builds by defining an XML file known as Project Object Model (POM), which is the fundamental unit of work. It always resides in the base directory of the project as a file named *pom.xml* [30].

The POM contains all the information about the project and its configuration details. It is used by Maven to build the project.

POM structure:

---

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<!-- The Basics -->
<groupId>...</groupId>
<artifactId>...</artifactId>
<version>...</version>
<packaging>...</packaging>
<dependencies>...</dependencies>
<parent>...</parent>
<dependencyManagement>...</dependencyManagement>
<modules>...</modules>
<properties>...</properties>

<!-- Build Settings -->
<build>...</build>
<reporting>...</reporting>

<!-- Additional Project Information -->
<name>...</name>
<description>...</description>
<url>...</url>
<inceptionYear>...</inceptionYear>
<licenses>...</licenses>
<organization>...</organization>
<developers>...</developers>
<contributors>...</contributors>

<!-- Environment Settings -->
<issueManagement>...</issueManagement>
<ciManagement>...</ciManagement>
<mailingLists>...</mailingLists>
<scm>...</scm>
<prerequisites>...</prerequisites>
<repositories>...</repositories>
<pluginRepositories>...</pluginRepositories>
<distributionManagement>...</distributionManagement>
<profiles>...</profiles>
</project>
```

---

Where:

- **Basics:** they contain all the required information about the project, as well as its required dependencies. All the POM files require the `<project>` element and three mandatory fields that will help to uniquely identify the project in a repository. They are known as Maven coordinates:
  - **groupId:** is the identifier of the project group. It is generally unique amongst an organization or project. For example, ODL uses the same groupId for the controller-related projects: `org.opendaylight.controller`.

- **artifactId:** is the identifier of the project, which generally corresponds to the project name. Along with the groupId, the artifactId defines the location of the artifact within a repository.
- **version:** is the version of the project. Along with the groupId and artifactId, is used to separate project versions from each other. For example: org.opendaylight.controller:toaster-parent:1.0 and org.opendaylight.controller:toaster-parent:1.1 are different versions of the same project, and can contain different implementations.

The notation that is used to uniquely identify a project is groupId:artifactId:version.

- **Build settings:** it is divided into the <build> and <reporting> elements. <build> elements handle the configuration and management of plugins that are used during the project's build phase. <reporting> elements are used in the site generation phase.
- **Additional project information:** although they are not mandatory, it includes elements that make developers life easier: licenses, contributors, organizations, etc. This information is normally used in the site generation, however, there might be certain plugins that use it.
- **Environment settings:** things like mailing list and Source Code Management (SCMs) are declared in this section. OpenDaylight uses SCMs to establish the connection to its versioning control system (OpenDaylight's Git repository) through Maven.

Maven define its default configuration in a file named Super POM, which defines common parameters like: default source and output directories, its basic plugins, repositories and the reporting directories that Maven will use while executing tasks.

Every POM extends the Super POM unless explicitly said, meaning that the configuration specified in the Super POM is inherited by the POMs that we create in our project.

## 6.4 Maven life-cycle:

When Maven starts building a project, it steps through a defined sequence of phases and executes goals that are registered in each phase. A goal represents a task that contributes to the building and management of a project. Goals may be bound to zero ore more build phases.

Figure 6.2 illustrates the typical Maven build life-cycle phases.

Phase	Handles	Description
Prepare-resources	Resource copying	Resource copying can be customized in this phase.
Compile	Compilation	Source code compilation is done in this phase.
Package	Packaging	This phase creates JAR/WAR package as mentioned in <packaging> POM.xml
Install	Installation	This phase installs the package in local/remote Maven repository.

Figure 6.2: Maven build life-cycle

There are always pre and post phases that can be used to register goals, which must run prior to or after a particular phases.

Maven has the following three standard life-cycles:

- **Clean:** defines the required phases to clean the output files of our project.
- **Default:** defines the required phases to build a project.
- **Site:** defines the required phases to publish a project.

### 6.4.1 Clean lifecycle

Clean lifecycle is used to clean the output files of our project.

It consists of the following phases:

- Pre-clean
- Clean
- Post-clean

Maven clean goal is bound to the clean phase in the clean life-cycle. This goal deletes the output of a build by clearing the specified build directory.

### 6.4.2 Default life-cycle

This is the primary Maven life-cycle and is used to build the application. Figure 6.3 shows the different phases of the default life-cycle. Some remarks on the default life-cycle:

- When a phase is called via Maven command, for example the install phase, the previous phases are also executed (compile, generate, test, package, etc.).
- Different Maven goals are bounded to different phases depending on the packaging type (JAR, WAR, etc.).

### 6.4.3 Site life-cycle

Site life-cycle is used to create documentation, reports and deploy our application in a site.

Phases:

- Pre-site
- Site
- Post-site
- Site-deploy

validate	validate the project is correct and all necessary information is available.
initialize	initialize build state, e.g. set properties or create directories.
generate-sources	generate any source code for inclusion in compilation.
process-sources	process the source code, for example to filter any values.
generate-resources	generate resources for inclusion in the package.
process-resources	copy and process the resources into the destination directory, ready for packaging.
compile	compile the source code of the project.
process-classes	post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
generate-test-sources	generate any test source code for inclusion in compilation.
process-test-sources	process the test source code, for example to filter any values.
generate-test-resources	create resources for testing.
process-test-resources	copy and process the resources into the test destination directory.
test-compile	compile the test source code into the test destination directory
process-test-classes	post-process the generated files from test compilation, for example to do bytecode enhancement on Java classes. For Maven 2.0.5 and above.
test	run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
prepare-package	perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package. (Maven 2.1 and above)
package	take the compiled code and package it in its distributable format, such as a JAR.
pre-integration-test	perform actions required before integration tests are executed. This may involve things such as setting up the required environment.
integration-test	process and deploy the package if necessary into an environment where integration tests can be run.
post-integration-test	perform actions required after integration tests have been executed. This may include cleaning up the environment.
verify	run any checks to verify the package is valid and meets quality criteria.
install	install the package into the local repository, for use as a dependency in other projects locally.
deploy	done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

Figure 6.3: Default life-cycle phases

## 6.5 Repositories

Maven gets most of its knowledge from the central Maven repository, which stores two kinds of information:

- **Archetype information:** it contains the archetype catalogs, which have information about the different project types that can be created, its full structure and its required capabilities. Archetypes will be described in the

section 6.8.

- **Dependency information:** it is a list of all the JAR files that a project requires. If a project depends on other JAR files, it also includes the dependencies of those JAR files, and so on.

Basically, a repository is a directory where all the project JARs, libraries, plugins or dependencies are stored and can be easily used by Maven.

There are three Maven repository types:

- **Local repository:** is a folder on the local machine that is created when a Maven command is executed for first time. It basically keeps all the project's dependencies. When a Maven build command is executed, Maven automatically downloads all the dependency JARs into the local repository. It helps avoid references to dependencies that are stored on remote machines every time a project is build.
- **Central repository:** provided by Maven community. It contains a large number of commonly used libraries. When Maven does not find a dependency in the local repository, it starts searching in the central repository. Maven repository is managed by the Maven community and it is not required to be configured (as it is specified in the Super POM). However, it requires Internet access to be used.
- **Remote repository:** sometimes, Maven does not find a dependency in the central repository, causing build process to fail. To prevent such situation, Maven provides the concept of remote repositories, which are developer's own repository containing the required libraries or project JARs. Maven will download dependencies from the remote repositories mentioned under the <repository> tag in the POM file.

## 6.6 Dependencies

Most projects depends on others to correctly build and run. Maven manages a list of the required dependencies by downloading and linking them to the compilation phase or to other goals that use them [25].

Dependencies are declared under the <dependency> statement in the project's POM file. They are uniquely identified by groupId, artifactId and version.

### 6.6.1 OpenDaylight common dependencies

OpenDaylight makes use of a large set of different dependencies, which depend on the imports used in the YANG models. However, there are some dependencies that are commonly used among the different ODL projects:

- **YANGTools:** these dependencies are a must when defining a YANG data model. They are provided by the YANGTools project, which are responsible of parsing the YANG schemas and defining its code generation as well as its data format abstraction. These dependencies will allow us to manage data models and its basic interactions: RPCs, notifications and transactions [31].
  - **yang-common:** contains a set of utilities that are common to every OpenDaylight project. It is composed by classes that handle the RPC results, as well as some YANG constants and the definition of QNames.
  - **yang-binding:** mainly contains all the utilities that are used in the mapping between YANG and Java. It is composed by classes that define RPCs (inputs, outputs, declaration), notifications, instance identifiers and other features related to the YANG to Java mapping.

---

```

<dependency>
  <groupId>org.opendaylight.yangtools</groupId>
  <artifactId>yang-binding</artifactId>
  <version>${yangtools.version}</version>
</dependency>

<dependency>
  <groupId>org.opendaylight.yangtools</groupId>
  <artifactId>yang-common</artifactId>
  <version>${yangtools.version}</version>
</dependency>

```

---

- **Config subsystem:** It contains all the base definitions of the config subsystem. In order to make a project aware of the config subsystem, *config.yang* model has to be imported in the project's YANG module. Thus, it is translated as a dependency to be added in its POM file [20].

---

```

<dependency>
  <groupId>org.opendaylight.controller</groupId>
  <artifactId>config-api</artifactId>
  <version>0.4.0-SNAPSHOT</version>
</dependency>

```

---

- **MD-SAL wiring:** this dependency is required for those projects that make use of the MD-SAL components. It inherits the definitions of brokers, notification service, RPC registry, etc. It is a very common dependency since most of the ODL projects use MD-SAL.

---

```

<dependency>
  <groupId>org.opendaylight.controller</groupId>
  <artifactId>sal-binding-config</artifactId>
  <version>1.3.0-SNAPSHOT</version>
</dependency>

```

---

- **OSGi:** it is an external dependency. This is required for those projects that intend to be deployed as OSGi bundles. It is required when a POM file sets its <packaging> statement as bundle. Contains activators, wiring, services, listeners, etc.

---

```

<dependency>
  <groupId>org.osgi</groupId>
  <artifactId>org.osgi.core</artifactId>
</dependency>

```

---

## 6.6.2 Dependency search sequence

When we execute a Maven build command, it starts looking for dependency libraries following this procedure [30]:

1. It searches for the dependency in the local repository. If not found, then it moves to step 2; if found, then it does the further processing.

2. It searches for the dependency in the central repository. If not found and remote repositories are mentioned, then it moves to step 4; if found, then it is downloaded to the local repository for future references.
3. If a remote repository has not been mentioned, Maven stops the processing and throws error 'Unable to find dependency'.
4. It searches for the dependency in remote repositories. If found, then it is downloaded to the local repository for future references; otherwise, Maven stops the processing and throws an error 'Unable to find dependency'.

## 6.7 Plugins

Maven is a plugin execution framework where every task is done by plugins. Maven plugins are generally used to:

- Create different output files (JAR, WAR, bundles, etc.).
- Compile code files.
- Testing code.
- Create project documentation and reports.

A plugin generally provides a set of goals, which can be executed by using the following syntax:

```
> mvn [plugin-name]:[goal-name]
```

For example, a project can be compiled with the Maven compiler plugin, using its compile goal, by running the following command:

```
> mvn compiler:compile
```

However, it is not needed to specify the plugin-name when we are using default plugins, as they are inherited by the Super POM. In the previous example, as the compiler plugin is the Maven default, we can execute the same command by running:

```
> mvn compile
```

Maven provides the following plugin types:

- **Build plugins:** they are executed during the build phase and are configured in the <build> element of the POM file.
- **Reporting plugins:** they are executed during the site generation phase and are configured in the <reporting> element in the POM file.

Some remarks about plugins:

- Plugins are specified under the <plugins> element in the *pom.xml* file
- While executing a task or goal, Maven looks for the POM file in the current directory and obtains the needed configuration, then Maven executes the corresponding task.
- Each plugin can have multiple goals.
- You can specify the phase where a plugin should start its processing under its <phase> element.



- You can configure tasks to be executed by binding them to the plugin goals.
- Maven handles everything else: it downloads the plugin if it is not available in the local repository and start its processing.

### 6.7.1 YANG interpreter plugin

The *yang-maven-plugin* is used to parse YANG schemas and generate outputs in the specified formats [31].

It is executed in the generate-sources phase, and it requires some features to be configured:

- **yangFilesRootDir:** specifies where are located the YANG files to interpret.
- **codeGeneratorClass:** specifies the format of the generated code.
- **outputBaseDir:** specifies where the output files are placed.

---

```
<plugin>
  <groupId>org.opendaylight.yangtools</groupId>
  <artifactId>yang-maven-plugin</artifactId>
  <version>${yangtools.version}</version>
  <dependencies>
    <dependency>
      <groupId>org.opendaylight.md.sal</groupId>
      <artifactId>maven-sal-api-gen-plugin</artifactId>
      <version>${yangtools.version}</version>
      <type>jar</type>
    </dependency>
  </dependencies>
  <executions>
    <execution>
      <goals>
        <goal>generate-sources</goal>
      </goals>
      <configuration>
        <yangFilesRootDir>src/main/yang</yangFilesRootDir>
        <codeGenerators>
          <generator>
            <codeGeneratorClass>org.opendaylight.yangtools.maven.sal.api.gen.plugin.
              CodeGeneratorImpl</codeGeneratorClass>
            <outputBaseDir>${salGeneratorPath}</outputBaseDir>
          </generator>
        </codeGenerators>
        <inspectDependencies>true</inspectDependencies>
      </configuration>
    </execution>
  </executions>
</plugin>
```

---

## 6.7.2 Maven bundle plugin

It is used to create an OSGI bundle from the contents of the compilation classpath, along with its resources and dependencies. This will allow our project to be later deployed in Karaf distribution.

---

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
    <instructions>
      <Export-Package>org.opendaylight.yang.gen.v1</Export-Package>
      <Import-Package>*</Import-Package>
    </instructions>
  </configuration>
</plugin>
```

---

- **Export-package:** is a list of the packages for a bundle to export. In OpenDaylight framework, we will normally use `org.opendaylight.yang.gen.v1`, which corresponds to the common nomenclature of all the Java classes generated from YANG models.
- **Import-package:** is a list of all the packages that are required by the bundle's contained packages. We will normally use `'*'`, which means that imports all the referred packages.

Usually, the `maven-bundle-plugin` is followed by the `builder-helper-maven-plugin`, which aim is to help the bundle plugin by accepting different goals. In OpenDaylight, we will be mostly using the `attach-artifact` goal, which is performed in the package phase, and allows to add additional files to deploy along with our bundles. This becomes really useful when we are using the config subsystem, since we can attach the configuration files together with our bundles.

---

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>attach-artifacts</id>
      <goals>
        <goal>attach-artifact</goal>
      </goals>
      <phase>package</phase>
      <configuration>
        <artifacts>
          <artifact>
            <file>${project.build.directory}/classes/toaster-provider-impl.xml</file>
            <type>xml</type>
            <classifier>config</classifier>
          </artifact>
        </artifacts>
      </configuration>
    </execution>
  </executions>
```

---

```
</executions>
</plugin>
```

---

## 6.8 Creating a project

Maven uses the archetype plugin to create projects. It allows to automatically create a project structure with the necessary starter files based on a specific type, which can be defined by the user. Plugin itself contains different prototypes for many Java-related projects (web applications, stand-alone applications, etc.) [30].

Depending on the project type, different structures and features are defined. For example, web applications use WAR files as output meanwhile stand-alone uses JAR.

All the default available archetypes, together with its capabilities, are listed on a catalog, which is obtained from the central repository. However, Maven allows to generate local catalogs to create your own archetypes.

Command to generate a project:

---

```
> mvn archetype:generate
+DarchetypeGroupId: specifies the groupId of the used archetype
(if not specified it takes as default: org.apache.maven.archetypes).

+DarchetypeArtifactId: specifies the name of the archetype to be used
(if not specified it takes as default: maven-archetype-quickstart).

+DarchetypeCatalog: file that contains the information about the used archetype
(if not specified it uses the internal catalog provided by the plugin itself).

+DarchetypeRepository: specifies the repository where the used archetype
is placed (if not specified it searches the archetype for in the repository
where the catalog comes from).
```

---

After running the command, Maven will ask for basic fields related to our project definition (groupId, artifactId, version, package, etc.) that have to be fulfilled.

With all this information, Maven handles all the build-related complexities.

### 6.8.1 OpenDaylight archetype

OpenDaylight has its own repository with different archetypes. There is a Karaf-based archetype that is commonly when developing OpenDaylight applications [32].

It is accessible by defining the following settings in the previous command:

---

```
> mvn archetype:generate

+DarchetypeGroupId=org.opendaylight.controller \
```

```
+DarchetypeArtifactId=opendaylight-startup-archetype \

+DarchetypeCatalog=http://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/ \

+DarchetypeRepository=+DarchetypeCatalog=http://nexus.opendaylight.org/content/repositories/
opendaylight.snapshot/archetype-catalog.xml
```

---

It will generate a project structure suitable for Karaf deployment and with different subproject folders to define different providers, features and data APIs.

For example, if we define the following parameters for our project:

---

```
Define value for property 'groupId': : org.opendaylight.hello

Define value for property 'artifactId': : hello

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

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

Define value for property 'classPrefix': Hello: :

Define value for property 'copyright': : Copyright (c) 2015 Yoyodyne, Inc.
```

---

The following structure will be autogenerated:

---

```
[Root directory]
  hello-api/
  artifacts/
  parent/
  features/
  hello-impl/
  distribution-karaf/
  pom.xml
```

---

Where:

- **hello-api:** is where the application's YANG data model is defined. It is named API because it is used by RESTCONF to define a set of REST APIs to view and manipulate the defined data.
- **artifacts:** is the folder where bundles are generated as.
- **features:** is used to deploy the application in the Karaf instance. It contains a feature descriptor, *features.xml*.
- **hello-impl:** is where the implementation of the defined data model is placed.
- **distribution-karaf:** it contains the instance in which the application will be deployed. Once it is compiled, it creates a distribution that can be executed to run a Karaf instance.

- **parent:** it only contains a single POM file, which is the parent of all the other POM files in the project. It acts similar to the Super POM file, it defines common properties that all POM files should inherit.

It also auto-generates a POM file in the root directory, known as aggregator file, that defines the project structure.

Relevant parts:

---

```
...  
  
<version>1.0.0-SNAPSHOT</version>  
<name>toaster-aggregator</name>  
<packaging>pom</packaging>  
<modelVersion>4.0.0</modelVersion>  
  
...  
  
<modules>  
  <module>hello-api</module>  
  <module>artifacts</module>  
  <module>features</module>  
  <module>hello-impl</module>  
  <module>distribution-karaf</module>  
  <module>parent</module>  
</modules>  
  
...
```

---

Note that the `<packaging>` element is marked as POM, meaning that this POM will not generate any output file. Subfolders, defined as modules in the aggregator POM, represent bundles and have their own POM files.



## Chapter 7

# YANG

The aim of this section is to introduce YANG as the modelling language used by OpenDaylight. This chapter will focus in the YANG basics and how it interacts with ODL. For a more detailed information, you can visit its RFC 6020 [33].

### 7.1 Introduction

YANG is a modelling language used to model data for the NETCONF protocol. A YANG module defines a hierarchy of data that can be used for NETCONF-based operations: including configuration and state data, Remote Procedure Calls (RPCs) and notifications.

YANG models the hierarchical organization of data as a tree in which each node has a name, and either a value or a set of child nodes. YANG provides clear and concise descriptions of nodes, as well as the interaction between those. Figure 7.1 illustrates YANG hierarchical organization.

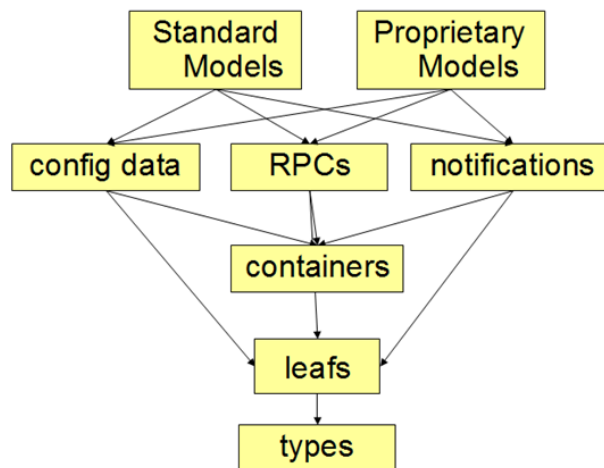


Figure 7.1: YANG hierarchical organization

Some of the advantages of using YANG as the NETCONF modelling language are:

- Human readable and easy to learn representation.
- Hierarchical data models.
- Reusable types and groupings (structured data).
- Extensibility through augmentation mechanisms.
- Supports operation definitions (RPCs and notifications).
- Data modularity through modules and submodules.
- Well-defined versioning rules.

For this reason, in OpenDaylight framework, YANG is used to model almost everything: from applications/plugins to the internal MD-SAL behaviour.

YANG management is handled by the YANGTools project, which contains modules that describe: code generation from YANG models, mapping between YANG and DOM/Java formats, modelling of data stores and its different operations (transactions, RPCs and notifications), provide RESTCONF accesss to data, etc.

## 7.2 Language overview

This section introduces the YANG basic constructs that will help in the understanding of the language specifics in later sections.

### 7.2.1 Modules and submodules

YANG supports data modularity by structuring data models into modules and submodules.

A module is the base unit of definition in YANG; it defines a single data model that can be a complete and cohesive model.

The hierarchy of a module can be augmented, allowing one module to add data nodes to the hierarchy defined in another module. For this reason, a module can also be an augmentation that adds or modifies nodes from another existing module.

Figure 7.2 illustrates the basic sections of a YANG module:

- **Header information:** it is used to uniquely identify our module. Contains general information about the module and its history (revisions).
- **Imports and includes:** specifies the dependencies between our module and other modules/submodules.
- **Type definitions:** defines the different data-types that will be used in our module's data modelling.
- **Configuration and operational data declarations:** is where configurational and operational data structures are defined.
- **Action and notification declarations:** is where RPC and notification statements are declared.



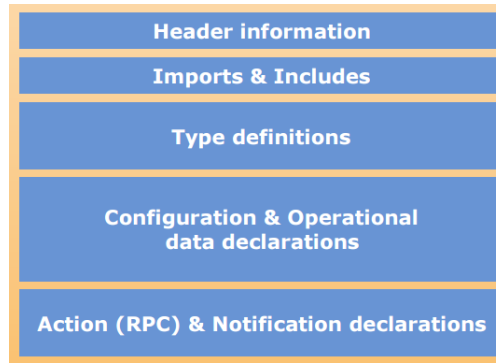


Figure 7.2: YANG module sections

A NETCONF server may implement multiple modules that describe the same data model, allowing different views of the same data. This is handled by module revision: two modules that describe the same data model with different revisions are independent.

A module may be divided into submodules, which are partial nodes that contribute definitions to a module, based on the needs of the module owner. The external view still remains of a single module, regardless the presence or size of its submodules. Each submodule may belong to only one module.

The 'include' statement allows a module or submodule to reference material defined in other submodules, while the 'import' statement allows to reference material defined in other modules.

Figure 7.3 shows the different relationships between modules and submodules, as well as the required YANG statements to model this behaviour.

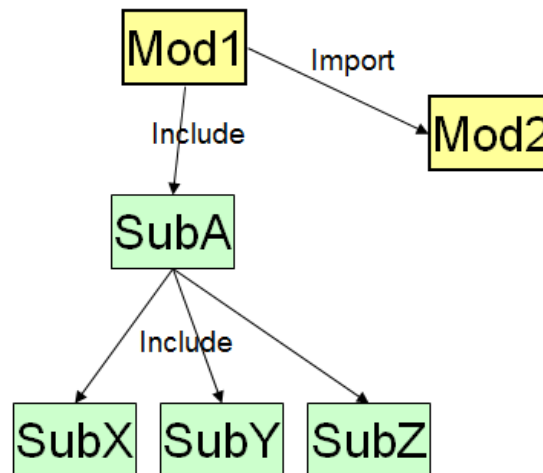


Figure 7.3: Relationships between modules and submodules

### Module example:

---

```
module acme-module {
  namespace "http://acme.example.com/module";
  prefix acme;
  import "yang-types" {
    prefix yang;
  }
  include "acme-system";
  organization "ACME Inc.";
  contact joe@acme.example.com;
  description "The module for entities implementing the ACME products";
  revision 2007-06-09 {
    description "Initial revision.";
  }

  ...
}
```

---

This example shows how the header information and imports/includes are defined. The 'prefix' statement defines a way to reference a module, which may be internal or external.

## 7.2.2 Data modelling basics

YANG defines four basic node types for modelling data:

- **Leaf:** one instance, is a node in the schema tree that has one value and no children.
- **Leaf-list:** multiple instances, is a sequence of leafs nodes with one value per leaf and no children.
- **Container:** one instance, is used to group related nodes (child nodes) in a subtree.
- **List:** multiple instances, holds related nodes. Each instance is identified by a key.

## 7.2.3 State and configurational data

YANG can model state data (operational), as well as configuration data, based on the 'config' statement. When a node is tagged with 'config false', its subhierarchy is flagged as state data, which can be reported using NETCONF <get> operations (read-only), but not <get-config> operations.

Operational data stores are used to show the running state view of the devices, network and services that an application might be looking at.

Configurational data stores are usually used to configure the devices in some way. These configurations are provided by users, being a way for the user to tell to the device how to behave.

Operational data example:

```
leaf toasterManufacturer {  
    type DisplayString;  
    config false;  
    mandatory true;  
    description "The name of the toaster's manufacturer. For instance, Microsoft Toaster.";  
}
```

In this example, `toasterManufacturer` is flagged as state data, so we can only access via `<get>` operations, meaning that it cannot be modified by a user.

## 7.2.4 Built-in types

YANG defines a set of built-in types, and also has a type mechanism through which additional types may be defined. Derived types can restrict their base type's set of valid values using mechanisms like range or pattern restrictions, which can be enforced by clients.

YANG built-in types are similar to those of many programming languages. Figure 7.4 details the type values that are available in YANG.

Name	Description
binary	Any binary data
bits	A set of bits or flags
boolean	"true" or "false"
decimal64	64-bit signed decimal number
empty	A leaf that does not have any value
enumeration	Enumerated strings
identityref	A reference to an abstract identity
instance-identifier	References a data tree node
int8	8-bit signed integer
int16	16-bit signed integer
int32	32-bit signed integer
int64	64-bit signed integer
leafref	A reference to a leaf instance
string	Human-readable string
uint8	8-bit unsigned integer
uint16	16-bit unsigned integer
uint32	32-bit unsigned integer
uint64	64-bit unsigned integer
union	Choice of member types

Figure 7.4: YANG built-in types

## 7.2.5 Derived types

YANG can define derived types from base types using the `'typedef'` statement. A base type can be either a built-in type or another derived type.

Example of a derived type:

---

```
typedef DisplayString {  
    type string {  
        length "0 .. 255";  
    }  
}
```

---

In this example, we have defined a derived type, `DisplayString`, from the built-in string type, which restricts the range of values from 0 to 255.

## 7.2.6 Grouping of reusable nodes

Groups of nodes can be assembled into reusable collections by using the 'grouping' statement. A grouping defines a set of nodes that are instantiated with the 'uses' statement. They can be refined (using the 'refine' statement) as they are used, allowing to override certain statements. Groupings are usually instantiated to refine or augment its nodes, allowing it to tailor its nodes to its particular needs.

Groupings can be defined in a module or submodule, and used in either that location or in another module/submodule that imports/defines it.

The 'grouping' statement is not a data definition statement and as such, it does not define any node in the data tree. It can be seen as a mechanism to structure data.

Grouping example:

---

```
import ietf-yang-types {prefix yang; revision-date "2010-09-24";}

grouping "mac-address-filter" {  
    leaf address {  
        mandatory true;  
        type yang:mac-address;  
    }  
    leaf mask {  
        type yang:mac-address;  
    }  
}  
}  
  
container LAN1 {  
    uses mac-address-filter {  
        refine "address" {  
            description "MAC address for LAN1";  
        }  
        refine "mask" {  
            description "Mask for LAN1";  
        }  
    }  
}
```

---

The type `yang:mac-address` refers to an external type named `mac-address` defined under the `yang` module, which is the prefix of the `ietf-yang-types` import.

The `LAN1` container refines the description of the `address` and `mask` leafs, which are defined in the `mac-address-filter` grouping.

## 7.2.7 Choices

YANG allows a data model to segregate incompatible nodes into distinct choices using the 'choice' and 'case' statements. The 'choice' statement contains a set of 'case' statements that define sets of schema nodes that cannot appear together. Each case may contain multiple nodes, but each node may appear in only one 'case' statement under a choice.

When an element from one case is created, all elements from all other cases are implicitly deleted.

The choice and case nodes appear only in the schema tree, not in the data tree or NETCONF messages.

Choice and case example:

---

```
import ietf-inet-types {prefix inet; revision-date "2010-09-24";}

choice layer-3-match {
  case "ipv4-match" {
    uses "ipv4-match-fields";
  }
  case "arp-match" {
    uses "arp-match-fields";
  }
}

grouping "ipv4-match-fields" {
  leaf ipv4-source {
    description "IPv4 source address.";
    type inet:ipv4-prefix;
  }
  leaf ipv4-destination {
    description "IPv4 destination address.";
    type inet:ipv4-prefix;
  }
}

grouping "arp-match-fields" {
  leaf arp-op {
    type uint16;
  }
  leaf arp-source-transport-address {
    description "ARP source IPv4 address.";
    type inet:ipv4-prefix;
  }
  leaf arp-target-transport-address {
    description "ARP target IPv4 address.";
    type inet:ipv4-prefix;
  }
}
```

```

container arp-source-hardware-address {
    description "ARP source hardware address.";
    presence "Match field is active and set";
    uses mac-address-filter;
}
container arp-target-hardware-address {
    description "ARP target hardware address.";
    presence "Match field is active and set";
    uses mac-address-filter;
}
}

```

---

In this example, we have a 'choice' statement for the `layer-3-match` choice. Depending on which case, different data trees will be created: the nodes defined in the `ip4-match-fields` or the ones defined in the `arp-match-fields` groupings. Note again that the types used in the different leafs are defined in the `inet` import.

## 7.2.8 Extending a data model (augmentations)

YANG allows a module to insert additional nodes into data models, including both the current module (and its sub-modules) or an external module.

The 'augment' statement defines the location in the data model hierarchy where new nodes are inserted, and the 'when' statement defines the conditions when the new nodes are valid.

Augmentation example:

---

```

grouping address-node-connector {
    list addresses {
        key "first-seen";
        leaf mac {
            type yang:mac-address;
            description "MAC address";
        }
        leaf ip {
            type inet:ip-address;
            description "IPv4 or IPv6 address";
        }
        leaf vlan {
            type uint16;
            description "VLAN id";
        }
    }
    augment "/inv:nodes/inv:node/inv:node-connector" {
        ext:augment-identifier "address-capable-node-connector";
        uses address-node-connector;
    }
}
}

```

---

This example defines a grouping `address-node-connector` that augmentates the inventory data model by populating it with its defined leafs.

If a module augments another module, the XML representation of the data will reflect the prefix of the augmenting module.

## 7.2.9 Identities

The 'identity' statement is used to define a new globally unique, abstract, and untyped identity. Its only purpose is to denote its name, semantics and existence. It takes as an argument an identifier that is the name of the hierarchy. It is followed by a block of substatements that hold detailed information about the identity.

The 'base' substatement is optional and takes as an argument a string that is the name of an existing identity, from which the identity that uses it is derived. If a 'prefix' is present on the base name, it refers to an identity defined in the module that was imported under that prefix.

Since submodules can not include the parent module, any identities in the module that need to be exposed to submodules must be defined in a submodule. Submodules can then include it to find the definition of the identity.

Identity example:

---

```
prefix toast;

identity toast-type {
  description "Base for all bread types supported by the toaster."

identity white-bread {
  base toast:toast-type;
  description "White bread.";
}

identity wheat-bread {
  base toast-type;
  description "Wheat bread.";
}
```

---

In this example, we define a base identity, `toast-type`, that is used by two other identities, `white-bread` and `wheat-bread`.

Note that we have used two ways of referencing the base: short and full path. As the base statement is defined in the same module, we can use the short path to reference it, `base toast-type`. However, we can also use the full path by using `base toast:toast-type`, that uses the `toaster` prefix in which our module is referenced.

## 7.3 Basic data modelling statements

### 7.3.1 Leaf nodes

A leaf node contains simple data like an integer or string. It has exactly one value of a particular type and no child nodes.

The 'leaf' statement is used to define a leaf node in the schema tree. It takes one argument, which is an identifier, followed by a block of substatements that hold detailed information about the leaf.

Figure 7.5 lists the supported leaf substatements:

substatement	cardinality
config	0..1
default	0..1
description	0..1
if-feature	0..n
mandatory	0..1
must	0..n
reference	0..1
status	0..1
type	1
units	0..1
when	0..1

Figure 7.5: Leaf substatements

- **'Mandatory' statement:** is optional, takes as an argument the string 'true' or 'false', and puts a constraint on valid data. If not specified, the default value is 'false'. If mandatory is 'true', means that the leaf must exist on the data tree.
- **'Type' statement:** must be present, takes as an argument the name of existing built-in or derived type.
- **'Default' statement:** is the value that the application uses if the leaf does not exist in the data tree. If the leaf is mandatory, the default value is the one specified in the 'default' statement. However, if the leaf is not mandatory, then its default value is its type's default value.

Leaf example:

```
leaf host-name {  
    type string;  
    description "Hostname for this system";  
}
```

NETCONF XML representation:

```
<host-name>my.example.com</host-name>
```

### 7.3.2 Leaf-list nodes

A leaf-list is a sequence of leaf nodes with exactly one value of a particular type per leaf.

While the 'leaf' statement is used to define a simple scalar variable of a particular type, the 'leaf-list' statement is used to define an array of a particular type. The 'leaf-list' statement takes one argument, which is an identifier, followed by a block of substatements that hold detailed information about the leaf-list.

Figure 7.6 lists the supported leaf-list substatements:



substatement	cardinality
config	0..1
description	0..1
if-feature	0..n
max-elements	0..1
min-elements	0..1
must	0..n
ordered-by	0..1
reference	0..1
status	0..1
type	1
units	0..1
when	0..1

Figure 7.6: Leaf-list substatements

- **'Min-elements' statement:** is optional, takes as an argument a positive integer, which puts a constraint on valid list entries.
- **'Max-elements' statement:** is optional, takes as an argument a positive integer, which puts a constraint on valid list entries.
- **'Ordered-by' statement:** defines whether the order of entries within a list are determined by the user (order defined by the user) or the system (unspecified order).

Leaf-list example:

---

```
leaf-list domain-search {
    type string;
    description "List of domain names to search";
}
```

---

NETCONF XML representation:

---

```
<domain-search>high.example.com</domain-search>
<domain-search>low.example.com</domain-search>
<domain-search>everywhere.example.com</domain-search>
```

---

### 7.3.3 Container nodes

A container node is used to group related nodes in a subtree. A container has only child nodes and no value. It may contain any number of child nodes of any type (including leaves, lists, containers and leaf-lists).

The 'container' statement is used to define an interior data tree node in the schema tree. It takes one argument, which is an identifier, followed by a block of substatements that hold the detailed information about the container.

YANG supports two container styles: those that exist only for organizing the hierarchy of data nodes, and those whose presence has an explicit meaning. The second style is called a presence container and is indicated using the 'presence' statement, which takes as an argument a string text indicating what the presence of the node means.

Figure 7.7 lists the supported container substatements:

substatement	cardinality
anyxml	0..n
choice	0..n
config	0..1
container	0..n
description	0..1
grouping	0..n
if-feature	0..n
leaf	0..n
leaf-list	0..n
list	0..n
must	0..n
presence	0..1
reference	0..1
status	0..1
typedef	0..n
uses	0..n
when	0..1

Figure 7.7: Container substatements

Container example:

```
container system {
  container login {
    leaf message {
      type string;
      description "Message given at start of login session";
    }
    leaf lastlogin {
      type string;
      description "Message that shows the last login";
    }
  }
  container information {
    leaf name{
      type string;
    }
  }
}
```

NETCONF XML representation:

```
<system>
  <login>
    <message>Good morning</message>
    <lastlogin>Your last login was Today at 8:32 AM</lastlogin>
  </login>
  <information>
    <name>YANG 2015</name>
```

```
</information>
</system>
```

---

Figure 7.8 represents the nodes that were previously defined in a tree representation. It simulates how these nodes would be inserted into the MD-SAL data stores.

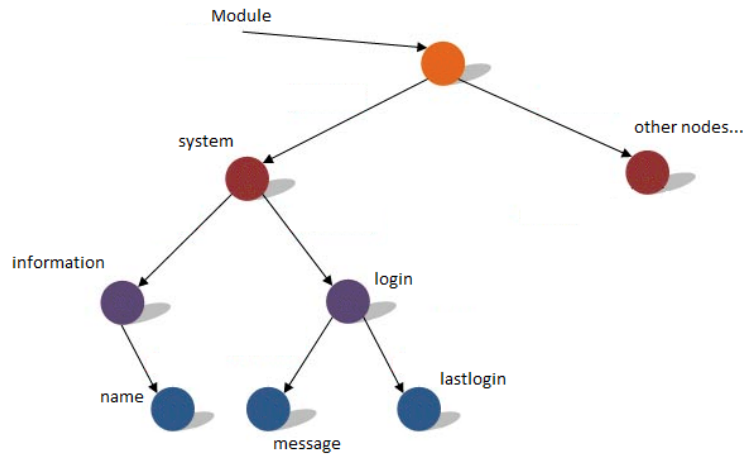


Figure 7.8: Container substatements

### 7.3.4 List nodes

A list defines a sequence of list entries. Each entry is like a structure or a recorded instance, and is uniquely identified by the values of its key leafs. A list can define multiple key leafs and may contain any number of childs of any type.

Figure 7.9 lists the supported list substatements:

substatement	cardinality
anyxml	0..n
choice	0..n
config	0..1
container	0..n
description	0..1
grouping	0..n
if-feature	0..n
leaf	0..n
leaf-list	0..n
list	0..n
must	0..n
presence	0..1
reference	0..1
status	0..1
typedef	0..n
uses	0..n
when	0..1

Figure 7.9: List substatements

List example:

---

```
list user {
    key "name";
    leaf name {
        type string;
    }
    leaf full-name {
        type string;
    }
    leaf class {
        type string;
    }
}
```

---

NETCONF XML implementation:

---

```
<user>
  <name>glocks</name>
  <full-name>Goldie Locks</full-name>
  <class>intruder</class>
</user>
<user>
  <name>snowey</name>
  <full-name>Snow White</full-name>
  <class>free-loader</class>
</user>
<user>
  <name>rzell</name>
  <full-name>Rapun Zell</full-name>
  <class>tower</class>
</user>
```

---

## 7.4 RPCs and notifications

### 7.4.1 RPC definitions

YANG allows the definition of NETCONF RPCs. The operation's name, input and output are modelled using YANG data definition statements. They are modelled using the 'rpc' statement.

- **'Input' statement:** is optional, it is used to define the input parameters of the RPC operation. If the leaf in the input tree has 'mandatory' statement, must be present in the NETCONF RPC invocation. If the 'config' statement is present for any node in the input tree, it is ignored. If a node has a 'when' statement that would evaluate false, then this node must not be present in the input tree.
- **'Output' statement:** is optional, it is used to define the output parameters of the RPC operation. If the leaf in the output tree has 'mandatory' statement, must be present in the NETCONF RPC invocation. If the 'config'

statement is present for any node in the output tree, it is ignored. If a node has a 'when' statement that would evaluate false, then this node must not be present in the output tree.

RPC example:

---

```
<user>
rpc activate-software-image {
  input {
    leaf image-name {
      type string;
    }
  }
  output {
    leaf status {
      type string;
    }
  }
}
```

---

## 7.4.2 Notification definitions

YANG allows the definition of NETCONF suitable notifications. YANG data definition statements are used to model the content of a notification.

The 'notification' statement is used to define a NETCONF notification. It takes one argument, which is an identifier, followed by a block of substatements that hold all the detailed information of the notification.

The 'notification' statement defines a notification node in the schema tree. If a leaf in the notification tree has a 'mandatory' statement with the value 'true', the leaf must be present in a NETCONF notification. If a 'config' statement is present for any node in the notification tree, it is ignored.

Notification example:

---

```
notification link-failure {
  description "A link failure has been detected";
  leaf if-name {
    type leafref {
      path "/interface/name";
    }
  }
  leaf if-admin-status {
    type admin-status;
  }
  leaf if-oper-status {
    type oper-status;
  }
}
```

---

## 7.5 YANG to Java mapping

This section details the creation of the Java DTOs and interfaces that are used to bind our application with the MD-SAL.

When generating Java files from a YANG schema, the type and number of the generated classes directly depends on the statements defined in the schema.

YANGTools project define all the mappings that will be declared in this section. And also defines a common set of mapping rules in order to standardize the code generation. These mapping rules apply constraints on the class, interface and package names and definitions [34].

### 7.5.1 Name mapping rules

#### Package name

The package name is composed by the following parts:

- **OpenDaylight prefix:** every package name starts with the prefix `org.opendaylight.yang.gen.v`
- **YANG version:** is specified in the 'yang-version' substatement of the YANG module.
- **Namespace:** equals to the value of the 'namespace' substatement of the YANG module. The following namespace characters are replaced with periods ( . ) : / : - @ \$ # ' \* + , ; =
- **Revision:** concatenation of the word 'rev' and the value of the 'revision' substatement of the YANG module, without leading zeros before month and day (e.g. rev2015821).

After the package name is generated, then it is checked if it contains any Java keywords or digits. If it does contain, then an underscore ( \_ ) is added before them.

List of Java keywords that are prefixed with underscore:

---

abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, double, do, else, enum, extends, false, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, true, try, void, volatile, while

---

#### Example:

---

```
module module {
  namespace "urn:2:case#module"
  prefix "sbd";
  organization "OpenDaylight";
  contact "http://www.whatever.com/";
  revision 2015-06-09{
  }
}
```

---

Package name will be: org.opendaylight.yang.gen.v1.urn:2:case#module.rev201569

After replacing digits and Java keywords: org.opendaylight.yang.gen.v1.urn.\_2.\_case.module.rev201569

In some cases, additional packages are also generated. It happens in the case where the superior YANG element contain specific subordinate YANG elements, listed in the figure 7.10.

Superior element	Subordinate element
list	list, container, choice
container	list, container, choice
choice	leaf, list, leaf-list, container, case
case	list, container, choice
rpc input rpc output	list, container, (choice isn't supported)
notification	list, container, (choice isn't supported)
augment	list, container, choice, case

Figure 7.10: Constructions that generate other packages

In those cases, superior elements also generate packages including its subordinate elements as Java classes. Names of the generated packages are composed by its superior element package name + its superior element name.

Example:

---

```
container cont {  
  container cont-inner {  
  }  
  list outter-list {  
    list list-in-list {  
    }  
  }  
}
```

---

Supposing that the container `cont` is a direct subelement of a module, it is also the superior element for its subordinate elements `cont-inner` and `outter-list`.

Java classes will be generated in the following structure:

- org.opendaylight.yang.gen.v1.urn.module.rev201569: package that contains the subordinate elements of the module.
  - Cont.java
- org.opendaylight.yang.gen.v1.urn.module.rev201569.con: package that contains the subordinate elements of the `cont` container element.
  - ContInner.java

- OutterList.java

Also, `outter-list` is the superior element of `list-in-list`, so another package will be created:

- `org.opendaylight.yang.gen.v1.urn.module.rev201569.cont.outter.list`: package that contains the subordinate elements of the `outter-list` list element.
- ListInList.java

## Class and interface names

Some YANG elements are mapped to Java classes and interfaces. YANG elements may contain names with various characters that are not permitted in Java class names. Characters like: `space` – `_` are deleted and subsequent letter is capitalized. Also, the first letter is capitalized.

For example: `example-name without _capitalization` is mapped to: `ExampleNameWithoutCapitalization`

## Getter and setter names

Some YANG elements are generated as getter or setter methods.

The process for a getter is:

- Getter method is named: `'get'` prefix + the name of the YANG element.
- Return type of the getter method is set to the element's `'type'` substatement value, converted to Java style.

The process for a setter is:

- Setter method is named: `'set'` prefix + the name of the YANG element.
- Input parameter name is set to the element's name, converted to Java parameter style.
- Input parameter type is set to the element's `'type'` substatement value, converted to Java style.
- Return parameter is set to void.

## 7.5.2 Code generation

### Module

Module is converted to Java as different classes in separate files. Names of the generated Java files are composed as follows: `<YANG_module_name><Sufix>`, where `<Sufix>` can be `Data`, `Service` or `Listener` [35].

- **Data interface:** extends `DataRoot` interface, which is defined in the `YANGTools` project and is used to identify our interface as the data root of our YANG module. It has a mapping similar to the container but only maps the top level children. In the example below, it can be seen how it maps the container but not its leafs.



- **Service interface:** is generated when RPCs are declared. It serves to describe the RPC contracts defined in a module. It implements `RpcService` from YANGTools project, which is a marker interface for tagging the YANG modules that make use of RPCs. Service interface declare methods for each of the RPCs declared in the YANG module.
- **Listener interface:** is generated when notifications are declared. It serves to describe the notification contracts defined in a module. It implements `NotificationListener` from YANGTools project, which is a marker interface for tagging the YANG modules that make use of notifications. Despite `NotificationListener` also extends Java `EventListener` interface, users should not implement this directly. Listener interface declare methods for each of the notifications declared in the YANG module.

YANG schema:

---

```

module toaster {
  revision "2015-08-20" {
  }

  container toaster {
    leaf toasterManufacturer {
      type DisplayString;
      config false;
      mandatory true;
    }
  }

  rpc make-toast {
    input {
      leaf toasterDoneness {
        type uint32 {
          range "1 .. 10";
        }
      }
      leaf toasterToastType {
        type identityref {
          base toast:toast-type;
        }
      }
    }
  }

  notification toasterOutOfBread {
  }
}

```

---

Generated Java interfaces:

- *ToasterData.java:*

---

```

public abstract interface ToasterData extends DataRoot {

```

```

...

public abstract Toaster getToaster();

...
}

```

---

- *ToasterService.java:*

```

public abstract interface ToasterService extends RpcService {

...

public abstract Future<RpcResult<Void>> makeToast (MakeToastInput paramMakeToastInput);

...
}

```

---

- *ToasterListener.java:*

```

public abstract interface ToasterListener extends NotificationListener {

...

public abstract void onToasterOutOfBread (ToasterOutOfBread paramToasterOutOfBread);

...
}

```

---

## Container

Container is mapped into two Java interfaces:

- **Container interface:** a DTO interface named as the YANG container that extends `ChildOf<P>` and `Augmentable<T>` interfaces, which are defined in the YANGTools project:
  - **ChildOf<P> interface:** marker interface that uniquely bounds generated Java interfaces to their parent container. Any nested Java interface generated from a YANG schema must implement this interface, where parameter `<P>` points to its superior element.
  - **Augmentable<T> interface:** indicates that this class can be augmented from objects of type `<T>` without introducing conflicts.

Container children are mapped as abstract getters of its defined types.

- **Builder interface:** named as the YANG container plus the 'Builder' suffix, which extends the `Builder<P>` class defined in YANGTools project. `Builder<P>` is an interface that contains a single method, `build()`, that returns an instance of the defined product `<P>`. Container's builder class implements the getters of the generated container interface and also implements the build method.

YANG schema:

---

```
container toaster {  
  leaf toasterManufacturer {  
    type DisplayString;  
    config false;  
    mandatory true;  
  }  
  
  leaf toasterModelNumber {  
    type DisplayString;  
    config false;  
    mandatory true;  
  }  
}
```

---

Generated Java classes:

- *Toaster.java*:

---

```
public abstract interface Toaster extends ChildOf<ToasterData>, Augmentable<Toaster> {  
  
    ...  
  
    public abstract DisplayString getToasterManufacturer();  
    public abstract DisplayString getToasterModelNumber();  
  
    ...  
}
```

---

- *ToasterBuilder.java*:

---

```
public class ToasterBuilder implements Builder<Toaster> {  
  
    ...  
  
}
```

---

## Leaf

Each leaf has to contain at least one 'type' substatement. The leaf is mapped as a getter method to the superior element with return type equal to the 'type' substatement value. The previous container example illustrates how a leaf is mapped at container level.

## Leaf-list

Each leaf-list has to contain one 'type' substatement. The leaf-list is mapped as a getter method to the superior element that returns a List of elements, whose type correspond to the 'type' substatement value defined in the 'leaf-list' statement.

YANG schema:

---

```
container cont {
  leaf-list leaflist {
    type typedef-union;
  }
}
```

---

Generated Java interfaces:

---

```
public abstract interface Cont extends ChildOf<ModuleNameData>, Augmentable<Cont> {

    ...

    public abstract List<TypedefUnion> getLeaflist();

    ...
}
```

---

## List

The 'list' statement is mapped to a DTO interface, which is similar to the container one. Moreover, it is mapped to the superior element as a getter method that returns a List of elements, whose type correspond to the generated DTO from the 'list' statement. The 'key' substatement is mapped to an independent Java class.

YANG schema:

---

```
container cont {
  list outter-list{
    key "name";
    leaf name {
      type string;
    }

    leaf-list leaf-list-in-list {
      type string;
    }

    list list-in-list {
      leaf-list inner-leaf-list {
        type int16;
      }
    }
  }
}
```

---

```

    }
}
}
}

```

---

Generated Java interfaces:

- *Cont.java:*

---

```

public abstract interface Cont extends ChildOf<ModuleName>, Augmentable<Cont> {

    ...

    public abstract List<OutterList> getOutterList();

    ...

}

```

---

- *OutterList.java:*

---

```

public abstract interface OutterList extends ChildOf<Cont>, Augmentable<OutterList> {

    ...

    public abstract List<String> getLeafListInList();
    public abstract List<ListInList> getListInList();
    OutterListKey getOutterListKey();

    ...

}

```

---

- *ListInList.java:*

---

```

public abstract interface ListInList extends ChildOf<OutterList>, Augmentable<ListInList> {

    ...

    public abstract List<Short> getInnerLeafList();

    ...

}

```

---

- *OutterListKey.java:*

---

```

public class OutterListKey {

    ...

}

```

---

```

private String Name;

public OutterListKey(String Name) {
    super();
    this.Name = Name;
}

public String getName() {
    return Name;
}

...
}

```

---

## Choice and case

Choice elements are mapped to a Java interface (marker interface) and also as getter methods in their superior element, with return type equal to the generated interface from the 'choice' statement.

The 'case' substatements are mapped to independent Java interfaces that extend the generated choice marker interface.

YANG schema:

```

container cont {
  choice choice-test {
    case case1 {
    }

    case case2 {
    }
  }
}

```

---

Generated Java classes:

- *Case1.java*:

```

public abstract interface Case1 extends ChildOf<Cont>, Augmentable<Case1>, ChoiceTest {
    ...
}

```

---

- *Case2.java*:

```

public abstract interface Case2 extends ChildOf<Cont>, Augmentable<Case2>, ChoiceTest {

```

---

```
...  
}
```

---

- *Cont.java*:

---

```
public abstract interface Cont extends ChildOf<ModuleName>, Augmentable<Cont> {  
  
    ...  
  
    public abstract ChoiceTest getChoiceTest();  
  
    ...  
}
```

---

- *ChoiceTest.java*:

---

```
public abstract interface ChoiceTest extends DataObject {  
  
    ...  
}
```

---

## Grouping and uses

A grouping is mapped to a Java interface. The 'use' substatement is mapped as an extension of the interface that is generated from the element that uses it. In other words, elements that are under the 'use' statement, in its generated interfaces, will have to extend the grouping interface.

YANG schema:

---

```
grouping group {  
}  
  
container cont {  
    uses group;  
}
```

---

Generated Java classes:

- *Cont.java*:

---

```
public abstract interface Cont extends ChildOf<ModuleName>, Augmentable<Cont>, Group {  
  
    ...  
}
```

---

- *Group.java*:

---

```
public abstract interface Group extends DataObject {

    ...

}
```

---

## Identities

The 'identity' statement is mapped as a Java abstract class:

- If it is a base identity, it generates a Java class named as its YANG 'identity' statement. This class extends the BaseIdentity interface, which is a marker interface that is defined in the YANGTools project.
- If it depends on a base identity ('base' substatement), it generates a class named as its YANG 'identity' statement that has to extend its base identity class.

YANG schema:

---

```
identity toast-type {

}

identity white-bread{
    base toast-type;
}
```

---

Generated Java classes:

- *ToastType.java*:

---

```
public abstract class ToastType extends BaseIdentity {

    ...

}
```

---

- *WhiteBread.java*:

---

```
public abstract class WhiteBread extends ToastType {

    ...

}
```

---



## RPCs

RPCs are mapped to Java as methods in the `ModuleService` class (it is shown in the module mapping example).

- 'Input' and 'output' substatements are mapped as independent DTO interfaces. If RPC inputs/outputs contain leafs, getters are declared in this interface. DTOs are named in the same way as the substatements are declared under the YANG statement.
- Builders are generated for each of the input/output DTO classes, which essentially are the same as container builders. They implement the declared getters and the `build()` method, which returns an instance of the specified product.

YANG schema:

---

```
rpc make-toast {
  input {
    leaf toasterDoneness {
      type uint32 {
        range "1 .. 10";
      }
    }
  }
  leaf toasterToastType {
    type identityref {
      base toast:toast-type;
    }
  }
}
```

---

Generated Java classes:

- *MakeToastInput.java*:

---

```
public abstract interface MakeToastInput extends DataObject, Augmentable<MakeToastInput> {

    ...

    public abstract Long getToasterDoneness();

    public abstract Class<? extends ToastType> getToasterToastType();

    ...
}
```

---

Note that the `getToasterTostType()` metod returns a class, this is because the `toasterToastType` leaf points to an identity (`type identityref`).

- *MakeToastInputBuilder.java*:

---

```
public class MakeToastInputBuilder implements Builder<MakeToastInput> {

    ...

}
```

---

NOTE: do not forget that the ModuleService interface is also generated, which contains the definitions of all the RPC methods.

## Augmentations

Augmentations are mapped as Java interfaces, which are named with the same name as the augmented interface but suffixed with a number that corresponds to the order number of augmenting interface.

The augmenting interface needs to extend Augmentation<T> interface, which is defined in the YANGTools project, and indicates that this class augments objects of type <T> without introducing conflicts.

YANG schema:

---

```
container cont {

    ...

}

...

augment "/cont" {

    ...

}
```

---

Generated Java classes:

- *Cont.java*:

---

```
public abstract interface Cont extends ChildOf<ModuleName>, Augmentable<Cont> {

    ...

}
```

---

- *Cont1.java*:

---

```
public abstract interface Cont1 extends ChildOf<ModuleName>, Augmentation<Cont> {

    ...

}
```

---

## Notifications

Notifications are mapped to Java as methods in the `ModuleListener` class (it is shown in the module mapping example).

Moreover, two Java interfaces are created for each notification, which are very similar to the container ones.

- A DTO interface named as the YANG 'notification' statement that also extends `Notification`, which is a class of YANGTools project that acts as a marker interface for the YANG notifications. If notifications contain leafs, getters are declared in this interface.
- Builder interfaces that have the same structure as container builders, which implement the getters declared and the `build()` method that returns an instance of the specified product.

YANG schema:

---

```
notification toasterRestocked {  
  leaf amountOfBread {  
    type uint32;  
  }  
}
```

---

Generated Java interfaces:

- *ToasterRestocked.java*:

---

```
public abstract interface ToasterRestocked extends DataObject, Augmentable<ToasterRestocked>  
, Notification {  
  
    ...  
  
    public abstract Long getAmountOfBread();  
  
    ...  
}
```

---

- *ToasterRestockedBuilder.java*:

---

```
public class ToasterRestockedBuilder implements Builder<ToasterRestocked> {  
  
    ...  
}
```

---

NOTE: do not forget that the `ModuleListener` interface is also generated, which contains all the definition of the notification methods.

## 7.6 OpenDaylight YANG models

Table 7.11 shows some of the YANG models that are commonly used in the OpenDaylight applications:

Model name	Description
	<b>OpenDaylight YANG extensions</b>
yang-ext.yang	YANG extensions for OpenDaylight.
	<b>Configuration subsystem</b>
	<u>Base types</u>
config.yang	Base YANG definitions for configuration subsystem.
rpc-context.yang	Base YANG definitions for RPC context reference used in the RPC routing.
	<u>MD-SAL modules</u>
opendaylight-md-sal-common.yang	Common definitions for MD-SAL
opendaylight-md-sal-dom.yang	Service definition for Binding Independent MD-SAL
opendaylight-md-sal-binding.yang	Service definition for Binding Aware MD-SAL
opendaylight-binding-broker-impl.yang	Service definition for Binding Aware MD-SAL
opendaylight-dom-broker-impl.yang	Service definition for Binding Independent MD-SAL
	<b>Services</b>
	<u>Inventory</u>
opendaylight-inventory.yang	The base (abstract) inventory model
opendaylight-inventory-config.yang	
	<u>Topology</u>
network-topology.yang	The base (abstract) network topology model
	<u>Model topology</u>
opendaylight-topology.yang	
opendaylight-topology-inventory.yang	
opendaylight-topology-view.yang	

Figure 7.11: YANG models that are commonly used in OpenDaylight

## Chapter 8

# Developing an OpenDaylight application

### 8.1 Introduction

Toaster project is a sample tutorial given by ODL to provide a better understanding of the MD-SAL infrastructure [36].

It is a suitable example as it provides the model of a programmable toaster, and shows how interactions are performed between different consumers (binding-aware and binding-independent) and a service provider that implements it.

Toaster application, as many other ODL applications, uses the following developing procedure, detailed in the figure 8.1:

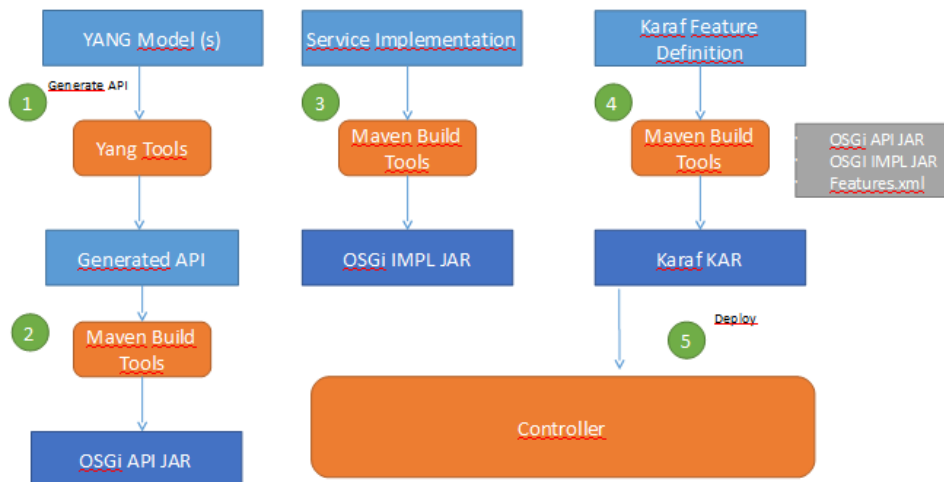


Figure 8.1: Application development process

1. We define the YANG data models for our application.
2. Thanks to YANGTools, Java APIs are generated from the defined models. Generated classes are mostly DTOs and service interfaces that describe the statements defined in the YANG models. Using Maven, they are packaged in a JAR file.

3. Then, we define the implementations of our models. We create Java classes that use and implement the DTOs and service interfaces generated from the YANG models. Using Maven, they are packaged in another JAR file.
4. If application uses Karaf, implementation and API packages are bundled together in a feature by using Maven.
5. Generated KAR (is the union of JAR files and *features.xml*) is deployed into the controller.

Toaster tutorial highlights in the following contexts:

- **YANG data models:** Toaster data model will be defined. Different YANG statements will be used, including configuration and operational data.
- **Binding-aware context:** generated Java APIs, binding-aware provider and consumer functions, and its interaction with the MD-SAL services.
- **Binding-independent context:** RESTCONF API and its interactions with the data stores.
- **Config subsystem:** plugin-level and system-level configurations.

Toaster sample will be explained step-by-step, starting with simple definitions that only enable access to operational data, and advancing to full-blown example that demonstrates many aspects of the MD-SAL: invoking RPCs via JMX and RESTCONF, accessing state data via JMX, notifications and a binding-aware consumer service.

Tutorial is divided in the following parts [37]:

- Part 1: toaster data model will be defined. A read-only implementation will be provided to retrieve operational data on the toaster.
- Part 2: RPCs will be defined and implemented. It will allow users to interact with the defined operations using RESTCONF interface, as well as see the changes in the status of the operational data tree.
- Part 3: illustrates how configuration data can be modified via RESTCONF and how the toaster can be a listener for those changes.
- Part 4: will provide statistical attributes, accessible by JMX implementations.
- Part 5: a binding-aware consumer for the toaster model will be defined. This provides a demonstration of how other business intelligence in the controller can access to data models and invoke RPCs for the purpose of providing additional business logic in the controller.
- Part 6: illustrates how to define notifications between the toaster provider and the binding-aware consumer.

### 8.1.1 Prerequisites and project structure

The required tools to develop and deploy the toaster project are:

- Java JDK 1.7+.
- Maven 3.1.1+.
- OpenDaylight controller (Helium or Lithium release).

Project's structure will be created by using the OpenDaylight start-up archetype defined in the section 6.8.

---

```
> mvn archetype:generate

+DarchetypeGroupId=org.opendaylight.controller \

+DarchetypeArtifactId=opendaylight-startup-archetype \

+DarchetypeCatalog=http://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/ \

+DarchetypeRepository=+DarchetypeCatalog=http://nexus.opendaylight.org/content/repositories/
opendaylight.snapshot/archetype-catalog.xml
```

---

Specifying the following parameters:

---

```
Define value for property 'groupId': : org.opendaylight.toaster

Define value for property 'artifactId': : toaster

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

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

Define value for property 'classPrefix':  Toaster :

Define value for property 'copyright': : Copyright (c) 2015 Yoyodyne, Inc.
```

---

Project structure should look like this:

---

```
[Root directory]
  toaster-api/
  artifacts/
  parent/
  features/
  toaster-impl/
  distribution-karaf/
  pom.xml
```

---

The *pom.xml* in the project's root acts like a bundle aggregator, which defines the parent project and declares all the modules presented in the structure above.

Important sections of the aggregator POM file [38]:

---

```
...

<version>1.0.0-SNAPSHOT</version>
<name>toaster-aggregator</name>
<packaging>pom</packaging>
```

```

<modelVersion>4.0.0</modelVersion>

...

<modules>
  <module>parent</module>
  <module>artifacts</module>
  <module>toaster-impl</module>
  <module>features</module>
  <module>toaster-api</module>
  <module>distribution-karaf</module>
</modules>

...
}

```

---

Note that the `<packaging>` is marked as POM. this means that it is only a declarative POM and no outputs are expected from him.

NOTE: as we will define a toaster consumer, we will manually add another folder named toaster-consumer and declare it in the aggregator.

## 8.2 Part 1: Defining an operational toaster

We will define a data model which maps to a toaster, and a service that provides operational data about the given toaster.

The toaster model and service will be made up of two YANG files, some new and modified Java classes, and a number of auto-generated Java files. It is important to remember that the MD-SAL provides the plumbing while the YANG data models provide the abstractions.

- *toaster.yang*: file that defines the northbound data model. Specifically, it defines the abstraction of a toaster that is visible to northbound clients (e.g. RESTCONF).
- *toaster-provider-impl.yang*: this file defines an implementation of the toaster model and the services it needs from the MD-SAL framework (e.g. binding-aware broker).

### 8.2.1 Defining a toaster data model

The *toaster.yang* file defines the northbound abstraction of the toaster data model, specifically its attributes, RPCs and notifications, which can be accessed by northbound clients. The *toaster.yang* file is located in the toaster-api project under `src/main/yang` folder.

---

```

//This file contains the Toaster YANG data definition.
module toaster {

    //The yang version - today only 1 version exists. If omitted defaults to 1.
    yang-version 1;

```



```

//a unique namespace for this toaster module, to uniquely identify it from other modules
that may have the same name.
namespace "http://netconfcentral.org/ns/toaster";

//a shorter prefix that represents the namespace for references used below
prefix toast;

//Defines the organization which defined / owns this .yang file.
organization "Netconf Central";

//provides a description of this .yang file.
description "YANG version of the TOASTER-MIB.";

//defines the dates of revisions for this yang file
revision "2009-11-20" {
    description "Toaster module in progress.";
}

//declares a base identity, in this case a base type for different types of toast.
identity toast-type {
    description "Base for all bread types supported by the toaster."
}

//the below identity section is used to define globally unique identities
identity white-bread {
    base toast:toast-type;          //logically extending the declared toast-type above.
    description "White bread.";    //free text description of this type.
}

identity wheat-bread {
    base toast-type;
    description "Wheat bread.";
}

//defines a new "Type" string type which limits the length
typedef DisplayString {
    type string {
        length "0 .. 255";
    }
}

// This definition is the top-level configuration "item" that defines a toaster.
// The "presence" flag connotes that there can only be one instance of a toaster which,
// if present, indicates the service is available.
container toaster {
    presence "Indicates the toaster service is available";
    description "Top-level container for all toaster database objects.";

    // Note in these three attributes that config = false. This indicates that they are
    // operational attributes.
    leaf toasterManufacturer {
        type DisplayString;
        config false;
    }
}

```

```

        mandatory true;
        description "The name of the toaster's manufacturer. For instance, Microsoft Toaster.";
    }

    leaf toasterModelNumber {
        type DisplayString;
        config false;
        mandatory true;
        description "The name of the toaster's model. For instance, Radiant Automatic.";
    }

    leaf toasterStatus {
        type enumeration {
            enum "up" {
                value 1;
                description "The toaster knob position is up. No toast is being made now.";
            }
            enum "down" {
                value 2;
                description "The toaster knob position is down. Toast is being made now.";
            }
        }
        config false;
        mandatory true;
    }
} // container toaster
} // module toaster

```

---

As it can be seen, we have defined a singleton `toaster` ('presence' statement) with different leaf's attributes (model, manufacturer, etc.) marked as operational data, and different toast types (using identities).

Most of the YANG statements are explained in the same code by using commentaries. However, a detailed information about the different YANG statements can be found in the section 7.1.

After we have defined our YANG data model, we have to compile it in order to generate the Java source files. To do this, we need to specify the `yang-maven-plugin` in the *pom.xml* of our *toaster-api* project.

---

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema
-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">

    <parent>
        <groupId>org.opendaylight.toaster</groupId>
        <artifactId>toaster-parent</artifactId>
        <version>0.1.0-SNAPSHOT</version>
        <relativePath>../parent</relativePath>
    </parent>

    <modelVersion>4.0.0</modelVersion>
    <artifactId>toaster-api</artifactId>
    <description>Yang Model/API for Toaster</description>
    <packaging>bundle</packaging>

```

```

<dependencies>
  <dependency>
    <groupId>org.opendaylight.yangtools</groupId>
    <artifactId>yang-binding</artifactId>
    <version>${yangtools.version}</version>
  </dependency>
  <dependency>
    <groupId>org.opendaylight.yangtools</groupId>
    <artifactId>yang-common</artifactId>
    <version>${yangtools.version}</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
    </plugin>
    <plugin>
      <groupId>org.opendaylight.yangtools</groupId>
      <artifactId>yang-maven-plugin</artifactId>
      <version>${yangtools.version}</version>
      <dependencies>
        <dependency>
          <groupId>org.opendaylight.mdsal</groupId>
          <artifactId>maven-sal-api-gen-plugin</artifactId>
          <version>${yangtools.version}</version>
          <type>jar</type>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
  <executions>
    <execution>
      <goals>
        <goal>generate-sources</goal>
      </goals>
      <configuration>
        <yangFilesRootDir>src/main/yang</yangFilesRootDir>
        <codeGenerators>
          <generator>
            <codeGeneratorClass>org.opendaylight.yangtools.maven.sal.api.gen.plugin.
              CodeGeneratorImpl</codeGeneratorClass>
            <outputBaseDir>${salGeneratorPath}</outputBaseDir>
          </generator>
        </codeGenerators>
        <inspectDependencies>true</inspectDependencies>
      </configuration>
    </execution>
  </executions>
</build>
</project>

```

We have defined two build plugins:

- **Maven-bundle-plugin:** allows to generate bundles from a defined model, which will be later deployed in the Karaf container. It is required since the POM `<packaging>` element is defined as bundle.
- **Yang-maven-plugin:** it defines a code generator, `CodeGeneratorImpl`, that generates Java files from the contents defined in a YANG schema.

Also, our toaster model requires dependencies the in YANGTools binding and common APIs, which contain the definitions of YANG to Java mapping and binding aware to binding independent data translation.

Plugins and dependencies are detailed in the Maven chapter 6.

Once we have defined our *pom.xml*, it is time to compile and run the project. To do this, we have to run the following command:

```
> mvn clean install
```

Following classes will be generated under the specified path:

- **Toaster:** an interface that represents the toaster container with methods to obtain its leaf node data.
- **ToasterData:** an interface that represents the top-level toaster module with one method `getToaster()` that returns a singleton `Toaster` instance.
- **WheatBread** and **WhiteBread:** abstract classes that represent the various toast types.
- **\$YangModelBindingProvider** and **\$YangModuleInfoImpl:** they are internally used by the MD-SAL to wire the toaster model for use. This is detailed in the config subsystem section 4.2.5.

Information about the mapping of YANG statements to Java classes can be found in the section 7.5.

## 8.2.2 Implementing the toaster operational data

We have defined a data model for the toaster, now we need to add an implementation to provide the operational data. We will create a class `ToasterImpl` under the `toaster-impl` project. On initialization, it writes the operational toaster data to the MD-SAL data store via the binding-aware broker interface, and deletes the data on close.

The reason this class is created in a separate project is that it allows the data model and its implementation to be provided by different bundles, thus allowing different bundles to define different implementations of the same data model.

---

```
public class ToasterImpl implements BindingAwareProvider, AutoCloseable {

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

    private ProviderContext providerContext;
    private DataBroker dataService;

    public static final InstanceIdentifier<Toaster> TOASTER_IID = InstanceIdentifier.
        builder(Toaster.class).build();
    private static final DisplayString TOASTER_MANUFACTURER = new DisplayString("Opendaylight");
```

```

private static final DisplayString TOASTER_MODEL_NUMBER = new DisplayString("Model 1 -
Binding Aware");

public ToasterImpl() {

}

/*****
 * AutoCloseable method
 *****/

/*
 * Called when MD-SAL closes the active session. Cleanup is performed,
 * i.e. all active registrations with MD-SAL are closed,
 */

@Override
public void close() throws Exception {

    // Delete toaster operational data from the MD-SAL data store
    WriteTransaction tx = dataService.newWriteOnlyTransaction();
    tx.delete(LogicalDatastoreType.OPERATIONAL, TOASTER_IID);

    Futures.addCallback( tx.submit(), new FutureCallback<Void>() {

        @Override
        public void onSuccess( final Void result ) {
            LOG.debug( "Delete Toaster commit result: {}", result );
        }

        @Override
        public void onFailure( final Throwable t ) {
            LOG.error( "Delete of Toaster failed", t );
        }

    });

    LOG.info("ToasterImpl: registrations closed");
}

/*****
 * BindingAwareProvider methods
 *****/

@Override
public void onSessionInitiated(ProviderContext session) {

    this.providerContext = session;
    this.dataService = session.getService(DataBroker.class);

    // Initialize operational data in MD-SAL data store
    initToasterOperational();

    LOG.info("onSessionInitiated: initialization done");
}

```

```

/*****
 * ToasterImpl private methods
 *****/

/*
 * Populates toaster's initial operational data into the MD-SAL operational
 * data store.
 * Note - we are simulating a device whose manufacture and model are fixed
 * (embedded) into the hardware. / This is why the manufacture and model
 * number are hardcoded
 */

private void initToasterOperational() {

    // Build the initial toaster operational data
    Toaster toaster = buildToaster(ToasterStatus.Up);

    // Put the toaster operational data into the MD-SAL data store
    WriteTransaction tx = dataService.newWriteOnlyTransaction();
    tx.put(LogicalDatastoreType.OPERATIONAL, TOASTER_IID, toaster);

    Futures.addCallback(tx.submit(), new FutureCallback<Void>() {

        @Override
        public void onSuccess(final Void result) {
            LOG.info("initToasterOperational: transaction succeeded");
        }

        @Override
        public void onFailure(final Throwable t) {
            LOG.error("initToasterOperational: transaction failed");
        }
    });

    LOG.info("initToasterOperational: operational status populated: {}", toaster);
}

private Toaster buildToaster( final ToasterStatus status ) {
    return new ToasterBuilder().setToasterManufacturer( TOASTER_MANUFACTURER )
                               .setToasterModelNumber( TOASTER_MODEL_NUMBER )
                               .setToasterStatus( status )
                               .build();
}

private void setToasterStatusUp( final Function<Boolean,Void> resultCallback ) {

    WriteTransaction tx = dataService.newWriteOnlyTransaction();
    tx.put( LogicalDatastoreType.OPERATIONAL, TOASTER_IID, buildToaster( ToasterStatus.Up ) );

    Futures.addCallback( tx.submit(), new FutureCallback<Void>() {

        @Override
        public void onSuccess( final Void result ) {

```

```

        notifyCallback( true );
    }

    @Override
    public void onFailure( final Throwable t ) {
        // We shouldn't get an OptimisticLockFailedException (or any ex) as no
        // other component should be updating the operational state.
        LOG.error( "Failed to update toaster status", t );
        notifyCallback( false );
    }

    void notifyCallback( final boolean result ) {

        if( resultCallback != null ) {
            resultCallback.apply( result );
        }
    }
});
}
}

```

---

#### Summary:

- In order to operate with the data stores, we have to get the binding aware broker service by using the `getSALService()` method.
- We have to initialize the operational data store.
- Methods are explained in the same code by using commentaries.

### 8.2.3 Wiring the ToasterImpl service

We have implemented the toaster provider service, now we have to get it instantiated and wired up with the MD-SAL. In order to do this, we are going to use the config subsystem.

We need first to describe our implementation service configuration and in which other services it depends on. In order to do this, we need to create a YANG file for our implementation and declare it as a module.

More details about how to make an application aware of the config subsystem can be found in the section 4.2.2.

---

```

module toaster-provider-impl {
    yang-version 1;
    namespace "urn:opendaylight:params:xml:ns:yang:toaster:provider:impl";
    prefix "toaster-provider-impl";

    import config { prefix config; revision-date 2013-04-05; }
    import opendaylight-md-sal-binding { prefix md-sal-binding; revision-date 2013-10-28;}

    description "Service definition for toaster project";

    revision "2014-12-10" {

```

```

    description "Initial revision";
  }

  identity toaster-provider-impl {
    base config:module-type;
    config:java-name-prefix ToasterImpl;
  }

  augment "/config:modules/config:module/config:configuration" {
    case toaster-provider-impl {
      when "/config:modules/config:module/config:type = 'toaster-provider-impl'";
      container binding-aware-broker {
        uses config:service-ref {
          refine type {
            mandatory true;
            config:required-identity md-sal-binding:binding-broker-osgi-registry;
          }
        }
      }
    }
  }
}

```

---

The augmentation of modules/module/configuration hierarchy choice-type node adds schema nodes specific to the toaster-provider-impl module (as indicated in the 'when' clause). This is where we define the needed configuration to initialize the toaster-provider-impl module; specifically, which external dependencies are needed.

As our ToasterImpl needs the binding-aware broker, we add a container that defines a dependency on the MD-SAL's binding-aware broker service. Syntactically, it defines a reference (of type `service-ref`) to the particular instance referred by the `md-sal-binding:binding-broker-osgi-registry` service identity, which is set at run-time by the MD-SAL.

To generate the Java source files that facilitate the service wiring, we need to define the *pom.xml* file for our toaster provider.

Under the toaster-impl project, the *pom.xml* looks like:

---

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.opendaylight.toaster</groupId>
    <artifactId>toaster-parent</artifactId>
    <relativePath>../parent</relativePath>
    <version>0.1.0-SNAPSHOT</version>
  </parent>

  <artifactId>toaster-impl</artifactId>
  <packaging>bundle</packaging>

```



```

<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>toaster-api</artifactId>
  </dependency>
  <dependency>
    <groupId>org.opendaylight.controller</groupId>
    <artifactId>config-api</artifactId>
    <version>0.4.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>org.opendaylight.controller</groupId>
    <artifactId>sal-binding-config</artifactId>
    <version>1.3.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>org.osgi</groupId>
    <artifactId>org.osgi.core</artifactId>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <configuration>
        <instructions>
          <Export-Package>org.opendaylight.yang.gen.v1</Export-Package>
          <Import-Package>*</Import-Package>
        </instructions>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.opendaylight.yangtools</groupId>
      <artifactId>yang-maven-plugin</artifactId>
      <dependencies>
        <dependency>
          <groupId>org.opendaylight.controller</groupId>
          <artifactId>yang-jmx-generator-plugin</artifactId>
          <version>${controller.config.version}</version>
        </dependency>
      </dependencies>
      <executions>
        <execution>
          <id>config</id>
          <goals>
            <goal>generate-sources</goal>
          </goals>
          <configuration>
            <codeGenerators>
              <generator>
                <codeGeneratorClass>org.opendaylight.controller.config.yangjmxgenerator.
                  plugin.JMXGenerator</codeGeneratorClass>

```

```

        <outputBaseDir>${jmxGeneratorPath}</outputBaseDir>
        <additionalConfiguration>
            <namespaceToPackage1>urn:opendaylight:params:xml:ns:yang:controller==
                org.opendaylight.controller.config.yang</namespaceToPackage1>
        </additionalConfiguration>
    </generator>
    <generator>
        <codeGeneratorClass>org.opendaylight.yangtools.maven.sal.api.gen.plugin.
            CodeGeneratorImpl</codeGeneratorClass>
        <outputBaseDir>${salGeneratorPath}</outputBaseDir>
    </generator>
</codeGenerators>
<inspectDependencies>true</inspectDependencies>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>

```

---

Required dependencies:

- **Toaster-api:** as our implementation uses data that is defined in this model.
- **Config-api:** it contains the basic definitions of the config subsystem. Every application that uses the config subsystem must declare this dependency.
- **Sal-binding-config:** it contains the definitions of most of the MD-SAL components. As we are using the binding-aware broker, we must declare this dependency.
- **Osgi:** it allows using OSGi's framework. It stems from the `maven-bundle-plugin`.

Declared plugins:

- **Maven-bundle-plugin:** it is used to create bundles from the defined files. Note that the `<Export-Package>` element is marked as `org.opendaylight.yang.gen.v1`, this means that all the Java auto-generated files will be exported (as they all have this nomenclature).
- **Yang-maven-plugin:** it is used to interpret YANG schemas and generate source files. As we are using the config subsystem, we must also add the JMX code generator.

Now, after re-generating the sources, two new Java classes will be generated:

- **ToasterImplModule:** concrete class whose `createInstance()` method provides a `ToasterImpl` instance.
- **ToasterImplModuleFactory:** concrete class that is internally instantiated by the MD-SAL, which creates `ToasterImplModule` instances.

## 8.2.4 Implementing the ToasterImplModule

ToasterImplModule class is mostly complete from the code generation. Only its `createInstance()` method needs to be implemented to instantiate and wire the ToasterImpl class.

---

```
public class ToasterImplModule extends org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.toaster.provider.impl.rev141210.AbstractToasterImplModule {

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

    public ToasterImplModule(org.opendaylight.controller.config.api.ModuleIdentifier identifier,
        org.opendaylight.controller.config.api.DependencyResolver dependencyResolver) {
        super(identifier, dependencyResolver);
    }

    public ToasterImplModule(org.opendaylight.controller.config.api.ModuleIdentifier identifier,
        org.opendaylight.controller.config.api.DependencyResolver dependencyResolver,
        org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.toaster.provider.impl.rev141210.ToasterImplModule oldModule, java.lang.AutoCloseable oldInstance) {
        super(identifier, dependencyResolver, oldModule, oldInstance);
    }

    @Override
    public void customValidation() {
        LOG.info("Performing custom validation");
        // add custom validation form module attributes here.
    }

    @Override
    public java.lang.AutoCloseable createInstance() {
        LOG.info("Creating a new Toaster instance");
        ToasterImpl provider = new ToasterImpl();
        String logMsg = "Provider: " + provider.toString();
        LOG.info(logMsg);
        getBindingAwareBrokerDependency().registerProvider(provider, null);

        return provider;
    }
}
```

---

The `createInstance()` method generates a ToasterImpl instance and manage its required dependencies. In this case, binding-aware broker dependency has already been injected by the MD-SAL and is available via the `getBindingAwareBrokerDependency()` method. The automatic injection is facilitated by the dependency augmentation that we had defined in the *toaster-provider-impl.yang* file.

The return type of `createInstance()` is `AutoCloseable`, so MD-SAL can inform our logic when it is time to shutdown.

## 8.2.5 Defining the initial XML configuration file

So far, we have defined the toaster data model (*toaster.yang*) and a provider implementation (*toaster-provider-impl.yang*). At this point, if bundles were deployed, the toaster model configuration (although we have not defined any config attributes yet) would be accessible via RESTCONF. However, the operational data provided by ToasterImpl service would not be accessible. This is because, we still need to tell the MD-SAL to 'deploy' our implementation, i.e. create an instance of ToasterImpl service, resolve its dependencies and advertise it for consumption.

To do this, we need to create an XML file that defines its initial configuration. This file will be pushed on start-up by the ConfigPusher.

Details about application instantiation can be found in the sections 4.2.4 and 4.2.3.

Under the toaster-impl project, in the src/main/resources folder, we create an XML file named *toaster-impl-config.xml* with the following contents:

---

```
<snapshot>
  <required-capabilities>
    <capability>urn:opendaylight:params:xml:ns:yang:toaster:provider:impl?module=toaster-provider-impl&revision=2014-12-10</capability>
    <capability>urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding?module=opendaylight-md-sal-binding&revision=2013-10-28</capability>
  </required-capabilities>
  <configuration>
    <data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <modules xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
        <module>
          <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:toaster:provider:impl">
            prefix:toaster-provider-impl</type>
          <name>toaster-impl</name>
          <binding-aware-broker>
            <type xmlns:binding="urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-broker-osgi-registry</type>
            <name>binding-osgi-broker</name>
          </binding-aware-broker>
        </module>
      </modules>
    </data>
  </configuration>
</snapshot>
```

---

Under the <configuration> element we specify the configuration to be pushed and its required dependencies. In this case, configuration is declared in our *toaster-provider-impl* module, and its required dependency (binding-aware broker) is declared in the *binding-osgi-broker* service. <Type> element refers to the path where those modules/services are declared.

Under the <required-capabilities> element we specify the YANG schemas where the configuration defined under the <configuration> element is defined. In this case, they are defined in the *toaster-provider-impl.yang* and the *opendaylight-md-sal-binding.yang* files.

A detailed explanation about initial XML configuration files can be found in the section 4.2.8.

To make the ConfigPusher aware of our configuration file, we have to define it in its POM file located in the controller

itself, under the controller/opendaylight/commons/opendaylight folder.

---

```
<properties>

...

<config.toaster.configfile>toaster-impl-config.xml</config.toaster.configfile>

...
</properties>
```

---

Finally, we have to attach our configuration file to the toaster implementation bundle. To do this, we will add another plugin in the *pom.xml*:

---

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>attach-artifacts</id>
      <goals>
        <goal>attach-artifact</goal>
      </goals>
      <phase>package</phase>
      <configuration>
        <artifacts>
          <artifact>
            <file>${project.build.directory}/classes/toaster-provider-impl.xml</file>
            <type>xml</type>
            <classifier>config</classifier>
          </artifact>
        </artifacts>
      </configuration>
    </execution>
  </executions>
</plugin>
```

---

The build-helper-maven-plugin is detailed in the section 6.7.2.

## 8.2.6 Getting the operational status of the toaster

In order to access to the operational status of our toaster, we have to deploy our bundles in the Karaf container. Thus, we can access data at run-time using, for example, RESTCONF.

This is done by declaring our project as a feature in the *feature.xml* file, located under the features folder:

```

<features name="odl-toaster-${project.version}" xmlns="http://karaf.apache.org/xmlns/features/
v1.2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://karaf.
apache.org/xmlns/features/v1.2.0 http://karaf.apache.org/xmlns/features/v1.2.0">

  <repository>mvn:org.opendaylight.yangtools/features-yangtools/{{VERSION}}/xml/features</repository>
  <repository>mvn:org.opendaylight.controller/features-mdsal/{{VERSION}}/xml/features</repository>
  <repository>mvn:org.opendaylight.netconf/features-restconf/{{VERSION}}/xml/features</repository>

  <feature name='odl-toaster-api' version='${project.version}' description='OpenDaylight ::
  toaster :: API'>
    <feature version='${yangtools.version}'>odl-yangtools-common</feature>
    <feature version='${yangtools.version}'>odl-mdsal-binding-base</feature>
    <feature version='${controller.restconf.version}'>odl-restconf</feature>
    <bundle>mvn:org.opendaylight.toaster/toaster-api/{{VERSION}}</bundle>
  </feature>

  <feature name='odl-toaster-impl' version='${project.version}' description='OpenDaylight ::
  toaster :: Impl'>
    <feature version='${controller.mdsal.version}'>odl-mdsal-broker</feature>
    <feature version='${controller.restconf.version}'>odl-restconf</feature>
    <feature version='${project.version}'>odl-toaster-api</feature>
    <bundle>mvn:org.opendaylight.toaster/toaster-impl/{{VERSION}}</bundle>
    <configfile finalname="toaster-impl-config.xml">mvn:org.opendaylight.toaster/toaster-
    impl/{{VERSION}}/xml/config</configfile>
  </feature>
</features>

```

---

Toaster project is declared as a major feature named `odl-toaster`, which is composed by different repositories and other features.

At the moment, we have to declare two features that correspond to the subprojects we have implemented: `odl-toaster-api` and `odl-toaster-impl`.

- **odl-toaster-api:** it declares three features that correspond to its dependencies: YANGTools commons and binding, and RESTCONF. Also, it declares a bundle that contains our toaster-api project.
- **odl-toaster-impl:** it declares three features that correspond to its dependencies: binding-aware broker, RESTCONF and the toaster API. Also, it declares a bundle that contains our toaster-impl project. Note that under the `<configfile>` statement, we specify that our bundle contains an initial configuration file.

More details on the Karaf container and OSGi bundles can be found in the chapter 5.

Once we have our features declared, we need to compile, run Karaf and install our features.

---

```

//compiling project (in the parent directory)
mvn clean install

//running Karaf
cd ../distribution-karaf/target/assembly/bin
./karaf

//installing features

```

```
feature:install odl-toaster-api
feature:install odl-toaster-impl

//check installed features
feature:list | grep toaster
```

---

We will perform a GET operation to get the operational status of the toaster. Figure 8.2 illustrates the GET request in payload format, while figure 8.3 shows the same operation by using YANGUI (DLUX).

```
http://localhost:8181/restconf/operational/toaster:toaster
{
  "toaster": {
    "toasterManufacturer": "Opendaylight",
    "toasterModelNumber": "Model 1 - Binding Aware",
    "toasterStatus": "up"
  }
}
```

Figure 8.2: HTTP GET request to retrieve the operational status

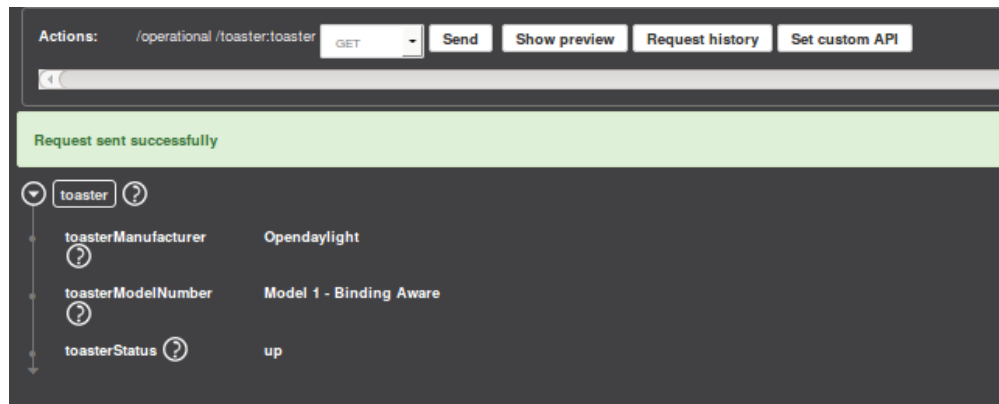


Figure 8.3: Using YANGUI to retrieve the operational status

## 8.3 Part 2: Enabling Remote Procedure Calls (RPCs)

This part will add some behaviour to our toaster by defining RPCs in the toaster YANG data model and writing its implementations.

### 8.3.1 Defining YANG RPC statements

We have to edit the existing *toaster.yang* file, where we will define two RPC methods: *make-toast* and *cancel-toast*.

---

```
rpc make-toast {
  description "Make some toast"
```

```

input {
  leaf toasterDoneness {
    type uint32 {
      range "1 .. 10";
    }
    default '5';
    description "This variable controls how well-done is the ensuing toast"
  }
  leaf toasterToastType {
    type identityref {
      base toast:toast-type; //Uses the defined identites as input
    }
    default 'wheat-bread';
    description "This variable informs the toaster of the type of material that is being toasted"
  }
}

} // rpc make-toast

rpc cancel-toast {
  description "Stops making toast, if any is being made."
} // rpc cancel-toast

```

---

Now, we have to compile and re-generate sources by running `mvn clean install`.

The following classes will be generated:

- **ToasterService:** an interface that defines the RPC methods corresponding to the YANG data model.
- **MakeToastInput:** an interface defining a DTO that provides the input parameters for the RPC call.
- **MakeToastInputBuilder:** a concrete class for creating MakeToastInput instances.

### 8.3.2 Implementing RPC methods

We have defined the data model interface for the RPC calls. Now, we have to provide its implementation. In order to do this, we are going to modify the `ToasterImpl` class to implement the new `ToasterService` interface and override its declared RPCS methods.

Only showing the modifications in the `ToasterImpl` class:

---

```

public class ToasterImpl implements ..., ToasterService {

    ...

    private RpcRegistration<ToasterService> rpcReg;

    ...

    private final ExecutorService executor;

    // The following holds the Future for the current make toast task.

```



```

// This is used to cancel the current toast.
private final AtomicReference<Future<?>> currentMakeToastTask = new AtomicReference<>();

public ToasterImpl() {
    executor = Executors.newFixedThreadPool(1);
}

/*****
 * AutoCloseable method
 *****/

@Override
public void close() throws Exception {

    // When we close this service we need to shutdown our executor!
    executor.shutdown();

    ...

    // Close active registrations
    rpcReg.close();

    ...
}

/*****
 * BindingAwareProvider methods
 *****/

@Override
public void onSessionInitiated(ProviderContext session) {

    ...

    // Register the RPC Service
    rpcReg = session.addRpcImplementation(ToasterService.class, this);

    ...
}

/*****
 * ToasterService methods
 *****/
/*
 * Restconf RPC call implemented from the ToasterService interface.
 * Cancels the current toast.
 */

@Override
public Future<RpcResult<Void>> cancelToast() {
    LOG.info("cancelToast");
    Future<?> current = currentMakeToastTask.getAndSet( null );
    if( current != null ) {
        current.cancel( true );
    }
}

```

```

    }

    // Always return success from the cancel toast call.
    return Futures.immediateFuture( RpcResultBuilder.<Void> success().build() );
}

/*
 * RestConf RPC call implemented from the ToasterService interface.
 * Attempts to make toast.
 */

@Override
public Future<RpcResult<Void>> makeToast(final MakeToastInput input) {
    LOG.info("makeToast: {}", input);

    final SettableFuture<RpcResult<Void>> futureResult = SettableFuture.create();

    checkStatusAndMakeToast( input, futureResult, 2 );
    LOG.info("makeToast returning...");
    return futureResult;
}

/*****
 * ToasterImpl private methods
 *****/

...

private RpcError makeToasterInUseError() {
    return RpcResultBuilder.newWarning( ErrorType.APPLICATION, "in-use", "Toaster is busy", null,
        null, null );
}

/*
 * Read the ToasterStatus and, if currently Up, try to write the status to
 * Down. If that succeeds, then we essentially have an exclusive lock and
 * can proceed to make toast.
 */

private void checkStatusAndMakeToast( final MakeToastInput input, final SettableFuture<RpcResult
<Void>> futureResult, final int tries ) {
    LOG.info( "checkStatusAndMakeToast" );

    final ReadWriteTransaction tx = dataService.newReadWriteTransaction();
    ListenableFuture<Optional<Toaster>> readFuture = tx.read( LogicalDatastoreType.OPERATIONAL,
        TOASTER_IID );

    final ListenableFuture<Void> commitFuture = Futures.transform( readFuture, new AsyncFunction
<Optional<Toaster>,Void>() {

        @Override
        public ListenableFuture<Void> apply(final Optional<Toaster> toasterData ) throws Exception {
            ToasterStatus toasterStatus = ToasterStatus.Up;
            if( toasterData.isPresent() ) {

```

```

        toasterStatus = toasterData.get().getToasterStatus();
    }

    LOG.debug( "Read toaster status: {}", toasterStatus );

    if( toasterStatus == ToasterStatus.Up ) {
        LOG.debug( "Setting Toaster status to Down" );

        // We are not currently making toast - try to update the status to Down
        // to indicate that we are going to make toast. This acts as a lock to prevent
        // concurrent toasting.
        tx.put( LogicalDatastoreType.OPERATIONAL, TOASTER_IID, buildToaster( ToasterStatus.Down ) );
        return tx.submit();
    }

    LOG.debug( "Oops - already making toast!" );

    // Return an error since we are already making toast. This will get
    // propagated to the commitFuture below which will interpret the null
    // TransactionStatus in the RpcResult as an error condition.
    return Futures.immediateFailedCheckedFuture( new TransactionCommitFailedException( "",
        makeToasterInUseError() ) );
}

});

Futures.addCallback( commitFuture, new FutureCallback<Void>() {

    @Override
    public void onSuccess( final Void result ) {
        // OK to make toast
        currentMakeToastTask.set( executor.submit( new MakeToastTask( input, futureResult ) ) );
    }

    @Override
    public void onFailure( final Throwable ex ) {
        if( ex instanceof OptimisticLockFailedException ) {
            // Another thread is likely trying to make toast simultaneously and updated the
            // status before us. Try reading the status again - if another make toast is
            // now in progress, we should get ToasterStatus.Down and fail.

            if( ( tries - 1 ) > 0 ) {
                LOG.debug( "Got OptimisticLockFailedException - trying again" );
                checkStatusAndMakeToast( input, futureResult, tries - 1 );
            }
            else {
                futureResult.set( RpcResultBuilder.<Void> failed().withError( ErrorType.APPLICATION,
                    ex.getMessage() ).build() );
            }
        }
        else {
            LOG.debug( "Failed to commit Toaster status", ex );
            // Probably already making toast.

```

```

        futureResult.set( RpcResultBuilder.<Void> failed().withRpcErrors( ((TransactionCommit
        FailedException)ex).getErrorList() ).build() );
    }
}

});

}

private class MakeToastTask implements Callable<Void> {

    final MakeToastInput toastRequest;
    final SettableFuture<RpcResult<Void>> futureResult;

    public MakeToastTask( final MakeToastInput toastRequest, final SettableFuture<RpcResult<Void>>
    futureResult ) {
        this.toastRequest = toastRequest;
        this.futureResult = futureResult;
    }

    @Override
    public Void call() {
        try {
            // make toast sleep based on ToasterDoneness
            Thread.sleep(toastRequest.getToasterDoneness());
        }

        catch( InterruptedException e ) {
            LOG.info( "Interrupted while making the toast" );
        }

        // Set the Toaster status back to up - this essentially releases the toasting lock.
        // We can not clear the current toast task nor set the Future result until the
        // update has been committed, so we pass a callback to be notified on completion.
        setToasterStatusUp( new Function<Boolean,Void>() {

            @Override
            public Void apply( final Boolean result ) {
                currentMakeToastTask.set( null );
                LOG.debug("Toast done");
                futureResult.set( RpcResultBuilder.<Void>success().build() );
                return null;
            }

        });

        return null;
    }
}
}

```

---

Summary:

- Now, `ToasterImpl` implements the `ToasterService` interface, from which it obtains the RPC methods.
- We have to notify to the MD-SAL binding-aware broker that our provider implements the RPC methods that are defined in the `ToasterService` interface. This is done by a registration in the `onSessionInitiated()` method. We also have to close this registration in the `close()` method.
- We have defined an executor, which will asynchronously execute the `MakeToastTask()` method. The reason to use executors is that they provide thread safety.
- Methods are explained in the same code by using commentaries.

Note that futures are commonly used while doing transactions and RPCs. Transaction and RPC messaging patterns are detailed in the MD-SAL chapter 3.

### 8.3.3 Invoking RPCs via RESTCONF

As the RPC wiring and registration are supported by the binding-aware broker, and we have already injected it, we do not have to add any other dependencies or configuration.

Thus, after re-generating the code, we will be able to invoke our defined RPCs.

To see the updated toaster status, we invoke the `make-toast` RPC (with a doneness of 10 to get the longest delay) and then we immediately invoke the GET request to retrieve the updated operational status. Figure 8.4 illustrates the POST request in payload format, while figure 8.5 shows the same operation by using YANGUI (DLUX). Figure 8.6 shows how the toaster status is now set to 'Down'.

```
http://localhost:8181/restconf/operations/toaster:make-toast
{
  "make-toast": {
    "input": {
      "toasterDoneness": "10",
      "toasterToastType": "wheat-bread"
    }
  }
}
```

Figure 8.4: HTTP POST request to invoke the make-toast RPC

The screenshot shows the YANGUI (DLUX) interface. At the top, the 'Actions' dropdown is set to '/operations /toaster:make-toast' with a 'POST' method selected. Buttons for 'Send', 'Show preview', 'Request history', and 'Set custom API' are visible. A green notification bar states 'Request sent successfully'. Below this, a tree view shows the 'make-toast' node expanded, with the 'input' node selected. Under 'input', there are two fields: 'toasterDoneness' with the value '10' and 'toasterToastType' with the value 'wheat-bread'.

Figure 8.5: Using YANGUI to invoke the make-toast RPC

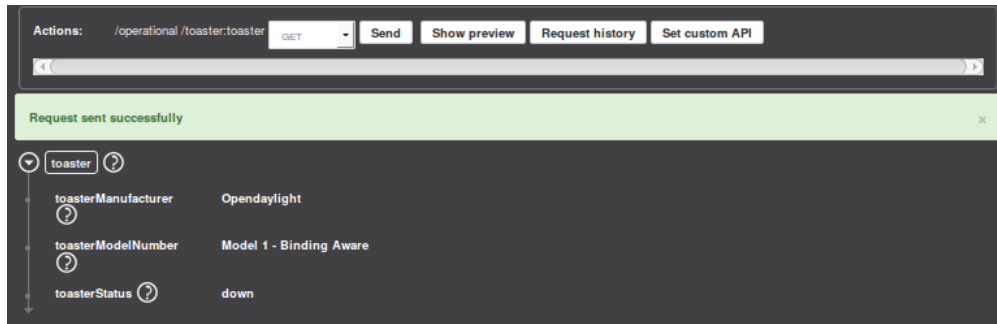


Figure 8.6: Updated operational status - ToasterStatus down

## 8.4 Part 3: Adding configuration data

In this part, we will show how to define and enable configuration attributes in our YANG toaster file.

We are going to define a new configuration attribute on the toaster that will allow the user to modify the number of seconds each level of doneness takes. More importantly, we will illustrate how our `ToasterImpl` can register for changes in the configuration data, as well as how the user can set up, update and delete that information.

### 8.4.1 Adding configuration data in the toaster data model

The first step is to add a new attribute, `darknessFactor`, to our toaster.

In the `toaster.yang` file:

---

```
container toaster {
    ...

    leaf darknessFactor {
        type uint32;
        config true;
        default 1000;
        description "The darkness factor";
    }

    ...
}
```

---

Note that the 'config' statement is now set to 'true'.

In order to make our `ToasterImpl` aware of the changes in the configuration data tree, we need to implement the `DataChangeListener` interface.

Things to be added to the `ToasterImpl`:

```

public class ToasterImpl implements ..., DataChangeListener {

    ...

    private ListenerRegistration<DataChangeListener> dcReg;

    // Thread safe holder for our darkness multiplier.
    private final AtomicLong darknessFactor = new AtomicLong( 1000 );

    ...

    /*****
     * AutoCloseable method
     *****/

    @Override
    public void close() throws Exception {

        ...

        // Close active registrations
        dcReg.close();

        ...
    }

    /*****
     * BindingAwareProvider methods
     *****/

    @Override
    public void onSessionInitiated(ProviderContext session) {

        ...

        // Register the DataChangeListener for Toaster's configuration subtree
        dataService.registerDataChangeListener( LogicalDatastoreType.CONFIGURATION, TOASTER_IID,
        this, DataChangeScope.SUBTREE );

        ...

        // Initialize configuration data in the MD-SAL data store
        initToasterConfiguration();

        ...
    }

    /*****
     * DataChangeListener methods
     *****/

    /*
     * Receives data change events on toaster's configuration subtree. Invoked
     * when data is written into the toaster's configuration subtree in the
     * MD-SAL data store.

```

```

*/

@Override
public void onDataChange( final AsyncDataChangeEvent<InstanceIdentifier<?>, DataObject> change ) {
    DataObject dataObject = change.getUpdatedSubtree();
    if( dataObject instanceof Toaster ) {
        Toaster toaster = (Toaster) dataObject;
        Long darkness = toaster.getDarknessFactor();
        if( darkness != null ) {
            darknessFactor.set( darkness );
        }
        LOG.info("onDataChanged - new Toaster config: {}", toaster);
    }
}

/*****
 * ToasterImpl private methods
 *****/
/*
 * Populates toaster's default config data into the MD-SAL configuration
 * data store.
 */

private void initToasterConfiguration() {
    // Build the default toaster config data
    Toaster toaster = new ToasterBuilder().setDarknessFactor(darknessFactor.get()).build();

    // Place default config data in data store tree
    WriteTransaction tx = dataService.newWriteOnlyTransaction();
    tx.put(LogicalDatastoreType.CONFIGURATION, TOASTER_IID, toaster);

    Futures.addCallback(tx.submit(), new FutureCallback<Void>() {
        @Override
        public void onSuccess(final Void result) {
            LOG.info("initToasterConfiguration: transaction succeeded");
        }

        @Override
        public void onFailure(final Throwable t) {
            LOG.error("initToasterConfiguration: transaction failed");
        }
    });

    LOG.info("initToasterConfiguration: default config populated: {}", toaster);
}

private class MakeToastTask implements Callable<Void> {

    ...

    @Override
    public Void call() {
        try {

```



```

        long darknessFactor = ToasterImpl.this.darknessFactor.get();
        // make toast sleep based on ToasterDoneness and darknessFactor
        Thread.sleep(darknessFactor * toastRequest.getToasterDoneness());
    }

    ...
}
}
}

```

---

Summary:

- Now ToasterImpl class implements the DataChangeListener interface, from which inherits the `onDataChanged()` method.
- We have to notify to the MD-SAL binding-aware broker that we are listening for changes in the configuration tree. This is done by a registration in the `onSessionInitiated()` method. We also have to close this registration in the `close()` method.
- Method implementations are explained in the same code by using commentaries.
- Toaster's configuration data tree also has to be initialized.
- Now, the delay (sleep) on the make toast task also depends on the `darknessFactor` value.

## 8.4.2 Changing the configuration data

As the listener registration is provided by the binding-aware broker, and we have already injected it, we do not have to add any other dependencies or configuration.

Thus, after re-generating the code, we will be able to change the configuration data and listen for the changes.

We will perform a PUT operation to modify the `darknessFactor` value. Figure 8.7 illustrates the PUT request in payload format, while figure 8.8 shows the same operation by using YANGUI (DLUX).

```

http://localhost:8181/restconf/config/toaster:toaster
{
  "toaster": {
    "darknessFactor": "10"
  }
}

```

Figure 8.7: HTTP PUT request to modify the `darknessFactor` value

An HTTP response with a code of 200 should be received.

If we perform a GET operation to the same URL, we should see the `darknessFactor` value updated. We will use YANGUI to validate it, as shown in the figure 8.9.

Moreover, if we call the `make-toast` RPC, we should notice that it takes more time to finish.

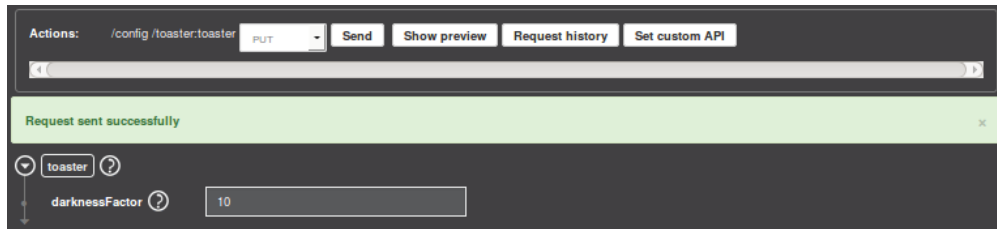


Figure 8.8: Using YANGUI to modify the darknessFactor value

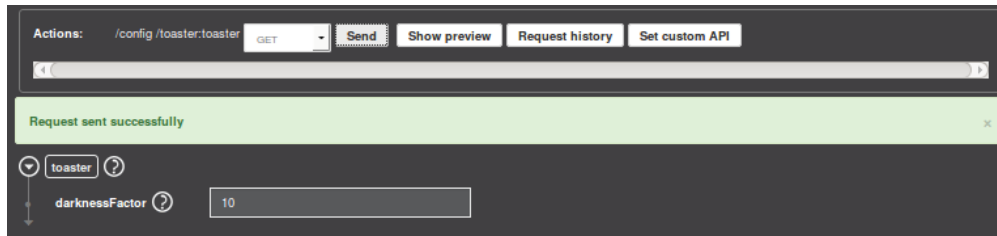


Figure 8.9: Updated darknessFactor value

## 8.5 Part 4: Adding state data (JMX access)

For internal statistics purpose and troubleshooting, we would like to keep track of how many pieces of toast the toaster has made over time. To do this, we need to declare an attribute, `toasts-made`, that increments whenever we successfully call the `make-toast` RPC. In addition, we would like a mechanism to clear the `toasts-made` count.

To accomplish this, the MD-SAL provides the capability to define internal state data and RPC calls that are only accessible via JMX.

### 8.5.1 Defining state data models

We will define `toasts-made` as statistical data on the toaster provider implementation, as it is where the `make-toast` happens. In addition, we will define a routed RPC, `clear-toasts-made`, that will set the `toasts-made` count to 0.

In the *toaster-provider-impl.yang*:

---

```

module toaster-provider-impl {

    ...

    import rpc-context { prefix rpcx; revision-date 2013-06-17; }

    ...

    // adding jmx/jconsole access to internal toaster state
    augment "/config:modules/config:module/config:state" {
        case toaster-provider-impl {
            when "/config:modules/config:module/config:type = 'toaster-provider-impl'";
            leaf toasts-made {

```

```

        type uint32;
    }
    rpcx:rpc-context-instance "clear-toasts-made-rpc";
}

identity clear-toasts-made-rpc;

rpc clear-toasts-made {
    description "JMX call to clear the toasts-made counter.";
    input {
        uses rpcx:rpc-context-ref {
            refine context-instance {
                rpcx:rpc-context-instance clear-toasts-made-rpc;
            }
        }
    }
}
}

```

---

State data nodes are inserted under the modules/module/state hierarchy.

The construction composed by 'augmentation', 'case' and 'when' statements indicates that we are implementing state data that is only valid for our module.

To declare the routing RPC:

- We declare an identity, `clear-toasts-made-rpc`, to define the RPC context.
- Then, we mark the `toasts-made` leaf as a possible context instance of the `clear-toasts-made-rpc` context.
- Finally, we have to declare the RPC statement. As it is a routed RPC, we have to use its input as context reference, which is used to declare the `toasts-made` leaf as the context-instance for that RPC.

In other words, we are specifying that `clear-toasts-made-rpc` RPC method will be only executed by our module on the `toasts-made` leaf, located in the state data hierarchy.

Details on how to declare, model and implement routed RPCs can be found in the section 3.7.3. This example provides an alternative way to model routed RPCs, but they are both essentially the same.

Run-time bean generation is detailed in the section 4.2.2.

Now, we re-generate the source files. Three additional classes are generated:

- **ToasterImplRuntimeMXBean:** JMX base interface that defines the `getToastsMade()` method, to provide access to the `toasts-made` attribute, and the `clear-toasts-made` RPC method.
- **ToasterImplRuntimeRegistration:** concrete class that wraps a `ToasterImplRuntimeMXBean` registration.
- **ToasterImplRuntimeRegistrator:** concrete class that registers a `ToasterImplRuntimeMXBean` implementation to the MD-SAL.

## 8.5.2 Implementing the state data model

Now that we have defined our state data model and its behaviour, we need to provide an implementation in our `ToasterImpl`.

Only showing modified parts:

---

```
public class ToasterImpl implements ..., ToasterImplRuntimeMXBean {

    ...

    private final AtomicLong toastsMade = new AtomicLong(0);

    ...

    /*****
    * ToasterImpl private methods
    *****/

    private class MakeToastTask implements Callable<Void> {

        ...

        @Override
        public Void call() {
            ...
            LOG.debug("Toast done");
            toastsMade.incrementAndGet();

            ...
        }
    }

    /*****
    * Run-time bean methods
    *****/
    /*
    * JMX RPC call implemented from the ToasterImplRuntimeMXBean interface.
    */

    @Override
    public void clearToastsMade() {
        LOG.info("clearToastsMade");
        toastsMade.set(0);
    }

    /*
    * Accessor method implemented from the ToasterImplRuntimeMXBean interface.
    */

    @Override
    public Long getToastsMade() {
        return toastsMade.get();
    }
}
```

```
}  
}
```

---

Summary:

- Now `ToasterImpl` interface implements the `ToasterImplRuntimeMXBean` interface, from which `ToasterImpl` inherits the state data methods.
- Method implementations are detailed in the same code by using commentaries.
- In the `MakeToastTask()` method, the `toastsMade` attribute is incremented after doing a toast.

### 8.5.3 Registering the `ToasterImplRuntimeMXBean` service

We need to register the `ToasterImpl` as a provider of the `ToasterImplRuntimeMXBean` service. The registration is done in the `ToasterImplModule`, via the `ToasterImplRuntimeRegistrator` returned by the `getRootRuntimeBeanRegistratorWrapper()` method.

---

```
public java.lang.AutoCloseable createInstance() {  
  
    ...  
  
    final ToasterImplRuntimeRegistration runtimeReg = getRootRuntimeBeanRegistratorWrapper().  
        register(provider);  
  
    ...  
}
```

---

There is no need to add any configuration and dependencies since run-time beans are defined in the `config-api`, which is already declared as a dependency in our `pom.xml` file.

### 8.5.4 Accessing state data via JMX

To access state data via JMX, we will use `JConsole`, located in the `bin` directory of our Java folder.

To locate our Java folder we can run the following command:

```
> readlink -f $(which java)
```

As an example, in our case it returns:

```
/usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java
```

Once we have located the Java path, we move to its `bin` directory and execute `Jconsole`.

---

```
//bin directory of our Java folder.  
> /usr/lib/jvm/java-7-openjdk-amd64/bin
```

```
//execute Jconsole  
> ./jconsole
```

Once it is executed, we have to connect to our Karaf running instance. JConsole will automatically identify it. Figure 8.10 illustrates how is this performed.

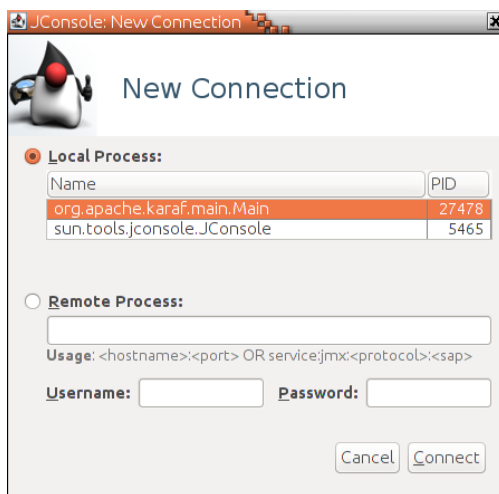


Figure 8.10: Connecting to the Karaf running instance

Then, we go back to MBean section, expand `opendaylight.org.controller > RuntimeBean > toaster-provider-impl > attributes`. As shown in the figure 8.11, we are able to see the current `toastsMade` value.

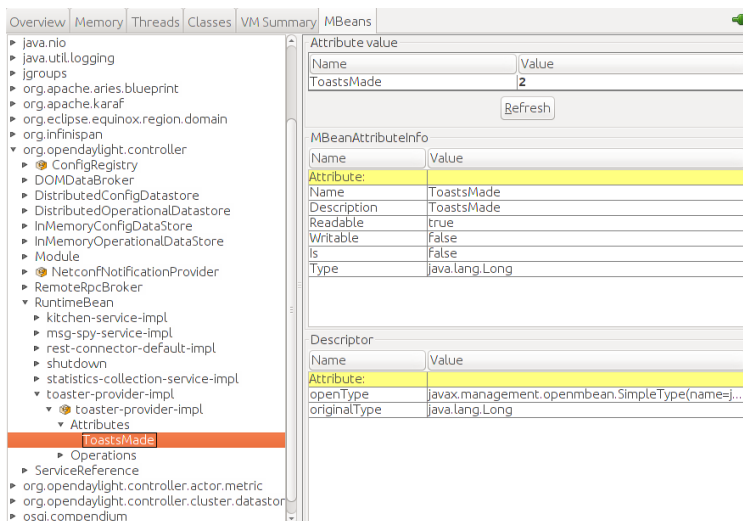


Figure 8.11: Tracking the `toastsMade` value

If we invoke the `make-toast` RPC and refresh the counter, we will see how it has been incremented.

Now, if we go to `opendaylight.org.controller > RuntimeBean > toaster-provider-impl > operations`, we will be able to invoke `clear-toasts-made` RPC, illustrated in the figure 8.12.

Figure 8.13 shows the `toastsMade` value after invoking the `clear-toasts-made` RPC.

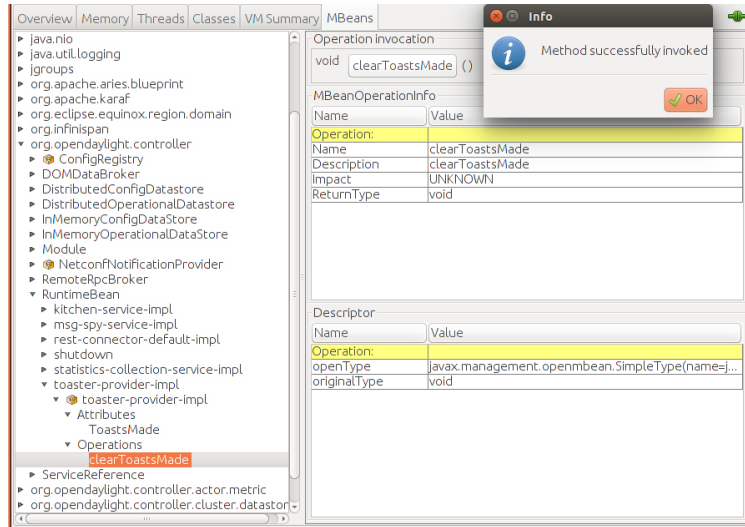


Figure 8.12: Invoking the clearToastsMade RPC

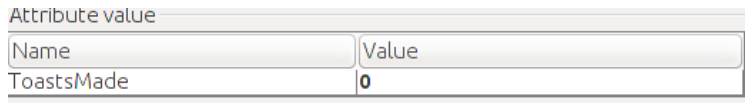


Figure 8.13: Updated toastsMade value

## 8.6 Part 5: Adding a ToasterService consumer

We have seen how RESTCONF can be used to access the ToasterService RPC methods. In this section, we will show how to access the ToasterService programmatically from within the controller.

Under the toaster-consumer project, we will create a new service called KitchenService that provides a method to make breakfast. This service will access the ToasterService to provide a toast for our breakfast.

KitchenService defines a higher-level service for making a full breakfast. This nicely demonstrates service chaining, where a consumer of one or more services is also a provider of another service. This example will only call into the toast service but one can see that it could be extended to other services.

### 8.6.1 Defining the KitchenService interface

We will hand-code the KitchenService data model and interface instead of defining it in YANG. In a true service you would likely want to define it in YANG to get benefit of the auto-generated classes and the out-of-box functionality that MD-SAL provides.

We define an enumeration of egg types to be later used in our Kitchen Service:

```
//EggsType.java
public enum EggsType {
    SCRAMBLED,
    OVER_EASY,
```

```
    POACHED
}
```

---

KitchenService has a single RPC method `makeBreakfast()` that uses as input: a concrete egg type, a toast and a toast doneness value.

---

```
//KitchenService.java
public interface KitchenService {
    Future<RpcResult<Void>> makeBreakfast( EggsType eggs, Class<? extends ToastType> toast, int
    toastDoneness );
}
```

---

## 8.6.2 Defining the KitchenServiceImpl implementation

We create a class, `KitchenServiceImpl`, to implement the `KitchenService` interface and access to the `ToasterService` to make the toast.

---

```
public class KitchenServiceImpl implements KitchenService {

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

    private final ToasterService toaster;

    public KitchenServiceImpl(ToasterService toaster) {
        this.toaster = toaster;
    }

    @Override
    public Future<RpcResult<Void>> makeBreakfast( EggsType eggs, Class<? extends ToastType>
    toast, int toastDoneness ) {

        ListenableFuture<RpcResult<Void>> makeToastFuture = makeToast( toastType, toastDoneness);
        ListenableFuture<RpcResult<Void>> makeEggsFuture = makeEggs( eggsType );

        // Combine the 2 ListenableFutures into 1 containing a list of RpcResults.
        ListenableFuture<List<RpcResult<Void>>> combinedFutures = Futures.allAsList( ImmutableList
        .of( makeToastFuture, makeEggsFuture ) );

        // Then transform the RpcResults into 1.
        return Futures.transform( combinedFutures, new AsyncFunction<List<RpcResult<Void>>,
        RpcResult<Void>>() {

            @Override
            public ListenableFuture<RpcResult<Void>> apply( List<RpcResult<Void>> results ) throws Exception {

                boolean atLeastOneSucceeded = false;
                Builder<RpcError> errorList = ImmutableList.builder();
                for( RpcResult<Void> result: results ) {
```



```

        if( result.isSuccessful() ) {
            atLeastOneSucceeded = true;
        }

        if( result.getErrors() != null ) {
            errorList.addAll( result.getErrors() );
        }

    }

    return Futures.immediateFuture( combinedFutures.<Void> getRpcResult( atLeastOneSucceeded,
        errorList.build() ) );
    });
}

private ListenableFuture<RpcResult<Void>> makeEggs( EggsType eggsType ) {
    // Returns a successful result.
    return combinedFutures.<Void> getRpcResult( true, Collections.<RpcError>emptyList() );
}

private ListenableFuture<RpcResult<Void>> makeToast( Class<? extends ToastType> toastType, int
toastDoneness ) {
    // Access the ToasterService to make the toast.
    MakeToastInput toastInput = new MakeToastInputBuilder().setToasterDoneness( (long) toastDoneness )
                                                                .setToasterToastType( toastType )
                                                                .build();

    return toaster.makeToast( toastInput );
}
}

```

---

### 8.6.3 Wiring the KitchenServiceImpl

Similar as we did in the toaster provider, we have to describe the kitchen service implementation in a YANG schema, so we can use the config subsystem to instantiate our KitchenServiceImpl and wire it with the MD-SAL.

*kitchen-service-impl.yang:*

---

```

module kitchen-service-impl {

    yang-version 1;
    namespace "urn:opendaylight:params:xml:ns:yang:kitchen:service:impl";
    prefix "kitchen-service-impl";

    import config { prefix config; revision-date 2013-04-05; }
    import rpc-context { prefix rpcx; revision-date 2013-06-17; }
    import opendaylight-md-sal-binding { prefix mdsal; revision-date 2013-10-28; }

    description "This module contains the base YANG definitions for the KitchenService
    implementation.";
}

```

```

revision "2014-01-31" {
    description "Initial revision.";
}

// This is the definition of the KitchenService interface identity.
identity kitchen-service {
    base config:service-type;
    config:java-class "org.opendaylight.controller.sample.kitchen.api.KitchenService";
}

// This is the definition of KitchenService implementation module identity.
identity kitchen-service-impl {
    base config:module-type;
    config:provided-service kitchen-service;
    config:java-name-prefix KitchenService;
}

augment "/config:modules/config:module/config:configuration" {
    case kitchen-service-impl {
        when "/config:modules/config:module/config:type = 'kitchen-service-impl'";
        container binding-aware-broker {
            uses config:service-ref {
                refine type {
                    mandatory true;
                    config:required-identity md-sal-binding:binding-broker-osgi-registry;
                }
            }
        }
    }
}

```

---

This is similar to the *toaster-provider-impl.yang* except that we have to define the `KitchenService` as service to be implemented by our `kitchen-service-impl` module. This is because we did not use a YANG schema to declare our services, and as a consequence, we did not take advantage of the auto-generated service interface and its automatic injection.

To declare our `KitchenService` as a service, we have to mark our `kitchen-service` identity as a config service using the `config:service-type` statement, and refer it to our `KitchenService` class by using the `config:java-class` statement.

Moreover, we have to declare the `KitchenServiceImpl` as a module. This is done in the `kitchen-service-impl` identity in a similar way as we did with the service declaration. However, we also have to declare the `KitchenService` as a provided service, using the `config:provided-service kitchen-service` statement.

The `kitchen-service` identity will be used by the config subsystem to advertise its declared service instances, provided by the `kitchen-service-impl` module, as an OSGi service. Since we did not define a kitchen YANG data model and advertise the `KitchenServiceImpl` with the MD-SAL RPC registry, the only way for other bundles to access the `KitchenService` is to obtain it via OSGi. Typically you would not need to advertise a service with OSGi unless a bundle that is not MD-SAL aware needs to access it. However, it demonstrates that is possible to do so.

Moreover, in the *pom.xml* of our `toaster-consumer` project, we have to specify similar plugins and dependencies as in the provider case. The only thing that has to be changed is that the `maven-bundle-plugin` `<Export-Package>` element will have to manually add our `KitchenService` and `KitchenServiceImpl` files, as they were not auto-generated

so they do not follow the nomenclature.

After running `mvn clean install`, several sources will be generated. We only need to modify the `createInstance()` method of the `KitchenServiceImplModule` to instantiate the `KitchenServiceImpl` and wire it:

---

```
@Override
public java.lang.AutoCloseable createInstance() {
    LOG.info("Creating a new KitchenServiceImpl instance");
    ToasterService toasterService = getBindingAwareBrokerDependency().getRpcService
        (ToasterService.class);
    KitchenServiceImpl consumer = new KitchenServiceImpl(toasterService);
    String logMsg = "Consumer: " + consumer.toString();
    LOG.info(logMsg);
    return consumer;
}
```

---

Note that as we advertised the `ToasterService` in the MD-SAL RPC registry, the `KitchenServiceImpl` can obtain it via the binding-aware broker.

## 8.6.4 Defining the initial XML configuration file

We need to create a configuration file for the toaster-consumer project, in a similar way as we did in the provider case.

*kitchen-service-config.yang*:

---

```
<snapshot>
  <required-capabilities>
    <capability>urn:opendaylight:params:xml:ns:yang:kitchen:service:impl?module=kitchen-service-impl
      &revision=2014-01-31</capability>
    <capability>urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding?module=opendaylight-md
      -sal-binding&revision=2013-10-28</capability>
  </required-capabilities>
  <configuration>
    <data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <modules xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
        <module>
          <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:kitchen:service:impl">prefix:kitchen
            -service-impl</type>
          <name>kitchen-service-impl</name>
          <binding-aware-broker>
            <type xmlns:binding="urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">
              binding:binding-broker-osgi-registry</type>
            <name>binding-osgi-broker</name>
          </binding-aware-broker>
        </module>
      </modules>
      <services xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
        <service>
          <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:kitchen:service:impl">prefix:
            kitchen-service</type>
```

```

    <instance>
      <name>kitchen-service</name>
      <provider>/modules/module[type='kitchen-service-impl'][name='kitchen-service-impl']
    </provider>
    </instance>
  </service>
</services>
</data>
</configuration>
</snapshot>

```

---

In this case, under the `<modules>` element, we declare our `kitchen-service-impl` module, and its required dependency (binding-aware broker), defined in the `binding-osgi-broker` service. Under the `<service>` element, we specify our `kitchen-service` as a provided service. The `<Type>` element refers to the path of where those modules/services are declared.

Under the `<required-capabilities>` element we specify the YANG schemas where the contents defined under the `<configuration>` element are defined. In this case, they are defined in the `kitchen-service-impl.yang` and `opendaylight-md-sal-binding.yang` files.

Now, we have to make the ConfigPusher aware of the new configuration file in the same way as we did with our provider.

Finally, we need to define the configuration file in the `build-helper-maven-plugin` of the `toaster-consumer`, as it can be attached with the generated bundles.

### 8.6.5 Adding a JMX RPC

At this point, if we deploy the `KitchenService` we will not be able to access it via `RESTCONF` as we did not define a YANG data model for it. However, we can use JMX to manipulate the `KitchenService`.

The MD-SAL also supports RPC calls via JMX. To do this, we only need to define a routed RPC in the YANG module and tie it to the `config:state` via augmentation. This is very similar as we did earlier for the `clear-toasts-made` RPC in the toaster provider.

We add the routed RPC declaration in the `kitchen-service-impl.yang`:

---

```

augment "/config:modules/config:module/config:state" {
  case kitchen-service-impl {
    when "/config:modules/config:module/config:type = 'kitchen-service-impl'";
    rpcx:rpc-context-instance "make-scrambled-with-wheat-rpc";
  }
}

identity make-scrambled-with-wheat-rpc;

rpc make-scrambled-with-wheat {
  description "Shortcut JMX call to make breakfast with scrambled eggs and wheat toast for testing.";
  input {
    uses rpcx:rpc-context-ref {
      refine context-instance {
        rpcx:rpc-context-instance make-scrambled-with-wheat-rpc;
      }
    }
  }
}

```

```

    }
  }
}
output {
  leaf result {
    type boolean;
  }
}
}
}

```

---

We have defined a routed RPC that will be executed under the state data hierarchy (as the whole state hierarchy is used as context-instance). By doing this, we are enabling JMX RPC calls that are specific to our module ('case' and 'when' statements).

After re-regenerating the source files, the run-time beans interfaces will be generated as in the provider case.

In our `KitchenServiceImpl` class, we need to implement the generated interface `KitchenServiceRuntimeMXBean`, which defines the `makeScrambledWithWheat()` method.

Only showing the implementation:

---

```

public class KitchenServiceImpl implements ..., KitchenServiceRuntimeMXBean {

    ...

    @Override
    public Boolean makeScrambledWithWheat() {
        try {
            // This call has to block since we must return a result to the JMX client.
            RpcResult<Void> result = makeBreakfast( EggsType.SCRAMBLED, WheatBread.class, 2 ).get();
            if( result.isSuccessful() ) {
                LOG.info( "makeBreakfast succeeded" );
            }
            else {
                LOG.warn( "makeBreakfast failed: " + result.getErrors() );
            }
            return result.isSuccessful();
        }

        catch( InterruptedException | ExecutionException e ) {
            LOG.warn( "An error occurred while making breakfast: " + e );
        }

        return Boolean.FALSE;
    }

    ...
}

```

---

It basically calls to the `makeBreakfast()` method and parses its result.

We also need to modify the `createInstance()` in the `KitchenServiceImplModule` to register the `KitchenService` with the `RuntimeMXBean` registration.

Modified part of the `createInstance()` method:

```
@Override
public java.lang.AutoCloseable createInstance() {

    ...

    final KitchenServiceRuntimeRegistration runtimeReg = getRootRuntimeBeanRegistratoWrapper().
        register( kitchenService );
}
```

Finally, we need to specify the consumer project as a feature to be deployed in Karaf.

```
<feature name='odl-toaster-consumer' version='${project.version}' description='OpenDaylight
:: toaster :: Consumer'>
    <feature version='${controller.mdsal.version}'>odl-mdsal-broker</feature>
    <feature version='${controller.restconf.version}'>odl-restconf</feature>
    <feature version='${project.version}'>odl-toaster-api</feature>
    <bundle>mvn:org.opendaylight.toaster/toaster-consumer/{{VERSION}}</bundle>
    <configfile finalname="kitchen-service-config.xml">mvn:org.opendaylight.toaster/toaster
-consumer/{{VERSION}}/xml/config</configfile>
</feature>
```

## 8.6.6 Invoking RPCs via JMX

We will go to JConsole and navigate to `opendaylight.org.controller > RuntimeBean > kitchen-service-impl > operations`. As shown in the figure 8.14, we will be able to call the `make-scrambled-with-wheat` RPC.

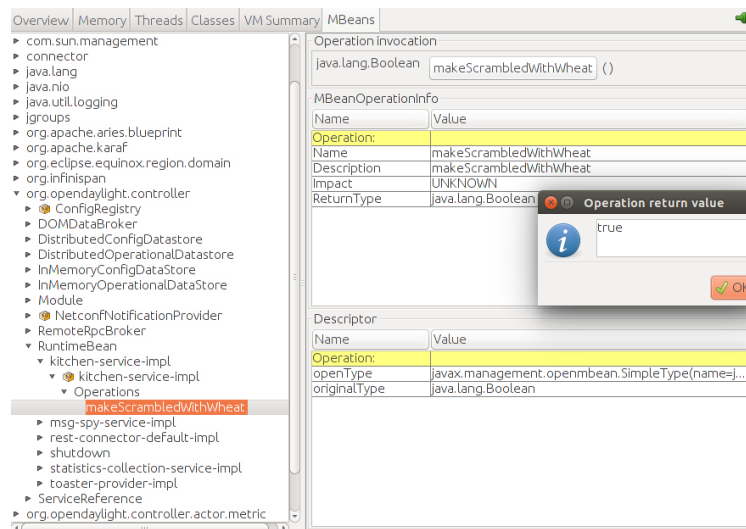
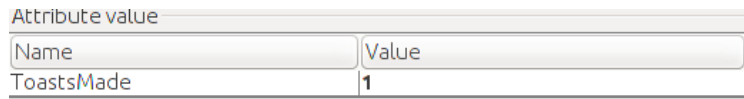


Figure 8.14: Invoking the `makeScrambledWithWheat` RPC

We go back to `opendaylight.org.controller > RuntimeBean > toaster-provider-impl > attributes`. Figure 8.15 illustrates how the `toastsMade` count has incremented (its initial value was 0, as the last thing we did was to invoke `clear-toasts-made` RPC).



Attribute value	
Name	Value
ToastsMade	1

Figure 8.15: Updated `toastsMade` value

## 8.7 Part 6: Defining notifications

This part will make use of the MD-SAL's unsolicited notification service to allow our `ToasterImpl` to send notifications when a significant events occur. Notifications can be consumed by registered listener implementations or by external NETCONF clients.

In our context, a toaster can only make toast if it has a supply of bread. Currently, our `ToasterImpl` has an infinite supply of bread which is not very realistic in the real world.

We will modify our `ToasterImpl` to have a finite stock of bread. So, when it is called to make a toast, if it is out of bread, a `toasterOutOfBread` notification will be sent.

We will also add an RPC call, `restock-toaster`, that can be used to set the amount of bread in stock. In addition, it will also send a `toasterRestocked` notification.

The `KitchenService` will register for both notifications and act accordingly when received.

### 8.7.1 Defining YANG notification statements

In the `toaster.yang` file:

```
module toaster {  
  
    ...  
  
    rpc restock-toaster {  
        description "Restocks the toaster with the specified amount of bread .";  
        input {  
            leaf amountOfBreadToStock {  
                type uint32;  
                description "Indicates the amount of bread to re-stock";  
            }  
        }  
    }  
  
    notification toasterOutOfBread {  
        description "Indicates that the toaster has run out bread.";  
    }  
  
    notification toasterRestocked {
```

```

    description "Indicates that the toaster has been restocked.";
    leaf amountOfBread {
        type uint32;
        description "Indicates the amount of bread that was re-stocked";
    }
}
}
}

```

---

After running `mvn clean install`, several new classes will be generated:

- **ToasterOutOfBread:** an interface defining a DTO for the `toasterOutOfBread` notification.
- **ToasterOutOfBreadBuilder:** a concrete class for creating `ToasterOutOfBread` instances.
- **ToasterRestocked:** an interface defining a DTO for the `toasterRestocked` notification.
- **ToasterRestockedBuilder:** a concrete class for creating `ToasterRestocked` instances.
- **ToasterListener:** interface generated from the `Toaster` module. It contains methods that correspond to the notification statements that are specified in the YANG model.

## 8.7.2 Publishing notifications

We will add code to our `ToasterImpl` to implement `restock-toaster` RPC and send the notifications.

Modified part of the `ToasterImpl`:

---

```

public class ToasterImpl implements ...{

    ...

    private NotificationProviderService notificationService;

    ...

    private final AtomicLong amountOfBreadInStock = new AtomicLong( 100 );

    ...

    /*****
    * BindingAwareProvider methods
    *****/

    @Override
    public void onSessionInitiated(ProviderContext session) {

        ...

        this.notificationService = session.getSALService(NotificationProviderService.class);

        ...
    }
}

```



```

}

/*****
 * ToasterService methods
 *****/
/*
 * RestConf RPC call implemented from the ToasterService interface.
 * Restocks the bread for the toaster
 */

@Override
public Future<RpcResult<java.lang.Void>> restockToaster(final RestockToasterInput input) {
    LOG.info( "restockToaster: {}", input );
    amountOfBreadInStock.set( input.getAmountOfBreadToStock() );

    if( amountOfBreadInStock.get() > 0 ) {
        ToasterRestocked reStockedNotification = new ToasterRestockedBuilder().setAmountOf
            Bread( input.getAmountOfBreadToStock() ).build();
        notificationService.publish( reStockedNotification );
    }

    return Futures.immediateFuture( RpcResultBuilder.<Void> success().build() );
}

/*****
 * ToasterImpl private methods
 *****/

...

private RpcError makeToasterOutOfBreadError() {
    return RpcResultBuilder.newError( ErrorType.APPLICATION, "resource-denied", "Toaster
        is out of bread", "out-of-stock", null, null );
}

private boolean outOfBread() {
    return amountOfBreadInStock.get() == 0;
}

private void checkStatusAndMakeToast( final MakeToastInput input, final SettableFuture<Rpc
    Result<Void>> futureResult, final int tries ) {

    ...

    if( toasterStatus == ToasterStatus.Up ) {
        if( outOfBread() ) {
            LOG.debug( "Toaster is out of bread" );
            return Futures.immediateFailedCheckedFuture( new TransactionCommitFailedException( "",
                makeToasterOutOfBreadError() ) );
        }

        LOG.debug( "Setting Toaster status to Down" );

        ...

```

```

}

private class MakeToastTask implements Callable<Void> {

    ...

    @Override
    public Void call() {

        ...

        toastsMade.incrementAndGet();
        amountOfBreadInStock.getAndDecrement();

        if( outOfBread() ) {
            LOG.info( "Toaster is out of bread!" );
            notificationService.publish( new ToasterOutOfBreadBuilder().build() );
        }

        ...
    }

    ...
}

...
}

```

---

Summary:

- We have to obtain the notification service via the binding-aware broker, so we can publish our notifications. This is done in the `onSessionInitiated()` method.
- `restockToaster()` method restocks the `amountOfBreadInStock` attribute and publishes a notification advertising it.
- `amountOfBreadInStock` is decreased after doing a toast in the `MakeToastTask()` method, if it gets to 0, then it publishes a notification advertising it.

### 8.7.3 Wiring our `ToasterImpl` for notifications

As the notification service is provided by the binding-aware broker, and we have already injected it, we do not have to add any other dependencies or configuration.

### 8.7.4 Implementing notifications in the `KitchenServiceImpl`

Modified code of the `KitchenServiceImpl` class:

```

public class KitchenServiceImpl implements ..., ToasterListener {

    private volatile boolean toasterOutOfBread;

    ...

    private ListenableFuture<RpcResult<Void>> makeToast( Class<? extends ToastType> toastType,
        int toastDoneness ) {

        if( toasterOutOfBread ) {
            LOG.info( "We're out of toast but we can make eggs" );
            return Futures.immediateFuture( combinedFutures.<Void> getRpcResult( true, Arrays.
                asList( RpcErrors.getRpcError( "", "partial-operation", null, ErrorSeverity.WARNING,
                    "Toaster is out of bread but we can make you eggs", ErrorType.APPLICATION, null ) ) ) );
        }

        ...
    }

    /*****
    * ToasterListener methods
    *****/
    /*
    * Implemented from the ToasterListener interface.
    */

    @Override
    public void onToasterOutOfBread( ToasterOutOfBread notification ) {
        LOG.info( "ToasterOutOfBread notification" );
        toasterOutOfBread = true;
    }

    /*
    * Implemented from the ToasterListener interface.
    */

    @Override
    public void onToasterRestocked( ToasterRestocked notification ) {
        LOG.info( "ToasterRestocked notification - amountOfBread: " + notification.getAmountOfBread() );
        toasterOutOfBread = false;
    }
}

```

---

The `onToasterOutOfBread` and `onToasterRestocked` notification methods simply set and clear the `toasterOutOfBread` attribute. When we call the `make-toast` RPC, if our toaster is `toasterOutOfBread`, we can not make the toast but it attempts to make the eggs.

## 8.7.5 Wiring the KitchenService for notifications

As the notification service is provided by the binding-aware broker, and we have already injected it, we do not have to add any other dependencies or configuration.

## 8.7.6 Testing notifications

We will make the toaster run out of bread. To do this, as the default value for the `amountOfBreadInStock` attribute is 100, we will change it to 3 in order to make it faster. Figure 8.16 illustrates how to do this.

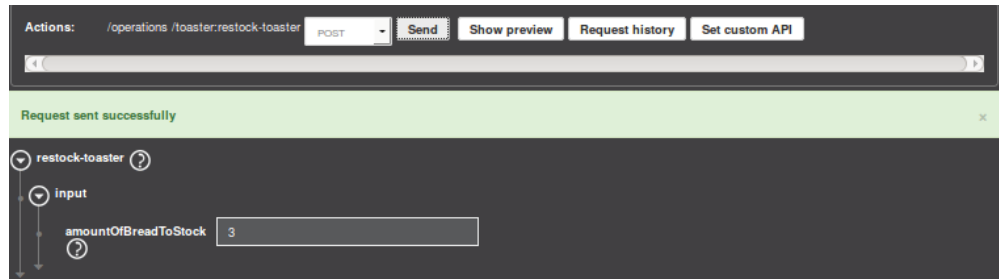


Figure 8.16: Changing the `amountOfBreadInStock` value

Then, we will call 3 times the `make-scrambled-with-wheat` RPC method.

After that, the toaster will go out of bread and will send a notification to the LOG, which is located into the `data/log/` folder inside the controller. Figure 8.17 shows the `ToasterOutOfBread` notification.

```
2015-10-13 09:37:16,967 | INFO | pool-54-thread-1 | OpendaylightToaster | 409 -  
org.opendaylight.controller.samples.sample-toaster-provider - 1.2.0.Lithium | Toaster is out of bread!  
2015-10-13 09:37:16,969 | INFO | pool-29-thread-1 | KitchenServiceImpl | 408 -  
org.opendaylight.controller.samples.sample-toaster-consumer - 1.2.0.Lithium | ToasterOutOfBread  
notification
```

Figure 8.17: `ToasterOutOfBread` notification

If we try to call again the `make-scrambled-with-wheat` RPC, another notification will be sent to the LOG indicating that it does not have bread but can make eggs instead. Figure 8.18 shows the received notification.

```
2015-10-13 09:43:06,427 | INFO | on(38)-127.0.0.1 | KitchenServiceImpl | 408 -  
org.opendaylight.controller.samples.sample-toaster-consumer - 1.2.0.Lithium | We're out of toast but we  
can make eggs
```

Figure 8.18: `MakeScrambledWithWheat` notification

Now, we will call the `restock-toaster` RPC method. Another notification will be sent to the LOG indicating the amount of bread we have restocked, as shown in the figure 8.19.

```
org.opendaylight.controller.samples.sample-toaster-consumer - 1.2.0.Lithium | ToasterRestocked  
notification - amountOfBread: 3
```

Figure 8.19: `ToasterRestocked` notification

# Bibliography

- [1] Software defined networking definition. <https://www.opennetworking.org/sdn-resources/sdn-definition>.
- [2] The road to sdn. <https://www.cs.princeton.edu/courses/archive/fall13/cos597E/papers/sdnhistory.pdf>.
- [3] Opendaylight: Md-sal architecture. [https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:MD-SAL:MD-SAL\\_Document\\_Review:Architecture](https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:MD-SAL_Document_Review:Architecture).
- [4] Opendaylight: Membership. <https://www.opendaylight.org/membership>.
- [5] Opendaylight: releases. <https://www.opendaylight.org/software/release-archives>.
- [6] Opendaylight: Tsc charter. <https://www.opendaylight.org/tsc-charter>.
- [7] Opendaylight: Lithium. <https://www.opendaylight.org/lithium>.
- [8] Opendaylight: Developer guide. <https://www.opendaylight.org/sites/opendaylight/files/bk-developers-guide-20150831.pdf>.
- [9] Ad-sal vs md-sal. <http://sdntutorials.com/difference-between-ad-sal-and-md-sal/>.
- [10] Opendaylight: Project life-cycle. <https://www.opendaylight.org/project-lifecycle-releases>.
- [11] Opendaylight: The open source sdn controller. <https://clnv.s3.amazonaws.com/2015/usa/pdf/BRKSDN-2761.pdf>.
- [12] Opendaylight: Download. <http://www.opendaylight.org/downloads>.
- [13] Official.opendaylight git repository. <https://git.opendaylight.org/gerrit/>.
- [14] Yangtools project. [https://wiki.opendaylight.org/view/YANG\\_Tools:Main](https://wiki.opendaylight.org/view/YANG_Tools:Main).
- [15] What is the document object model? <http://www.w3.org/TR/DOM-Level-3-Core/introduction.html>.
- [16] Xml dom tutorial. [http://www.w3schools.com/xml/dom\\_intro.asp](http://www.w3schools.com/xml/dom_intro.asp).
- [17] Opendaylight: Yang schema and model. [https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:YANG\\_Schema\\_and\\_Model](https://wiki.opendaylight.org/view/OpenDaylight_Controller:YANG_Schema_and_Model).
- [18] Opendaylight: Binding model. [https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:Binding\\_Model](https://wiki.opendaylight.org/view/OpenDaylight_Controller:Binding_Model).
- [19] Opendaylight: Binding-aware components. [https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:Binding\\_Aware\\_Components](https://wiki.opendaylight.org/view/OpenDaylight_Controller:Binding_Aware_Components).

- [20] Opendaylight: Config subsystem. [https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:Config:Main](https://wiki.opendaylight.org/view/OpenDaylight_Controller:Config:Main).
- [21] Notes on the config subsystem. <http://sdntutorials.com/notes-on-config-subsystem-in-opendaylight/>.
- [22] Osgi alliance. <http://www.osgi.org/Main/HomePage/>.
- [23] Osgi: An overview of its impact on the software life-cycle. <http://java.sys-con.com/node/1765471>.
- [24] Apache karaf. <http://karaf.apache.org/>.
- [25] Apache maven project. <https://maven.apache.org/>.
- [26] Maven: The complete reference. <http://books.sonatype.com/mvnref-book/reference/>.
- [27] Downloading maven. <http://maven.apache.org/download.cgi>.
- [28] Downloading eclipse. <https://www.eclipse.org/downloads/>.
- [29] Maven software for eclipse. <http://www.eclipse.org/m2e/>.
- [30] Maven tutorial. <http://www.tutorialspoint.com/maven/>.
- [31] Yangtools user guide. [https://wiki.opendaylight.org/view/YANG\\_Tools:User\\_Guide](https://wiki.opendaylight.org/view/YANG_Tools:User_Guide).
- [32] Opendaylight start-up project archetype. [https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:MD-SAL:Startup\\_Project\\_Archetype](https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Startup_Project_Archetype).
- [33] M. Bjorklund. Yang - a data modeling language for the network configuration protocol (netconf), October 2010. RFC6020.
- [34] Opendaylight: Yang to java mapping. [https://wiki.opendaylight.org/view/YANG\\_Tools:YANG\\_to\\_Java\\_Mapping](https://wiki.opendaylight.org/view/YANG_Tools:YANG_to_Java_Mapping).
- [35] Yangtools git repository. <https://github.com/opendaylight/yangtools>.
- [36] Opendaylight: Toaster example overview. [https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:MD-SAL:Toaster\\_Tutorial](https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Toaster_Tutorial).
- [37] Opendaylight: Toaster example step-by-step. [https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:MD-SAL:Toaster\\_Step-By-Step](https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Toaster_Step-By-Step).
- [38] Toaster git repository. <https://github.com/opendaylight/coretutorials/tree/master/toaster>.