ODL InfraUtils

Release master

Faseela K, Ravit Peretz

Jun 30, 2022

CONTENTS

This documentation provides critical information needed to help you write ODL Applications/Projects using Infrautils, which offers various generic utilities and infrastructure for ease of application development.

Contents:

CHAPTER

ONE

INFRAUTILS DESIGN SPECIFICATIONS

Starting from Carbon, InfraUtils project uses RST format Design Specification document for all new features. These specifications are perfect way to understand various InfraUtils features.

Contents:

Table of Contents

1.1 Title of the feature

[link to gerrit patch]

Brief introduction of the feature.

1.1.1 Problem description

Detailed description of the problem being solved by this feature

Use Cases

Use cases addressed by this feature.

1.1.2 Proposed change

Details of the proposed change.

Yang changes

This should detail any changes to yang models.

Configuration impact

Any configuration parameters being added/deprecated for this feature? What will be defaults for these? How will it impact existing deployments?

Note that outright deletion/modification of existing configuration is not allowed due to backward compatibility. They can only be deprecated and deleted in later release(s).

Clustering considerations

This should capture how clustering will be supported. This can include but not limited to use of CDTCL, EOS, Cluster Singleton etc.

Other Infra considerations

This should capture impact from/to different infra components like MDSAL Datastore, karaf, AAA etc.

Security considerations

Document any security related issues impacted by this feature.

Scale and Performance Impact

What are the potential scale and performance impacts of this change? Does it help improve scale and performance or make it worse?

Targeted Release

What release is this feature targeted for?

Alternatives

Alternatives considered and why they were not selected.

1.1.3 Usage

How will end user use this feature? Primary focus here is how this feature will be used in an actual deployment.

For most InfraUtils features users will be other projects but this should still capture any user visible CLI/API etc. e.g. Counters

This section will be primary input for Test and Documentation teams. Along with above this should also capture REST API and CLI.

Features to Install

odl-infrautils-all

Identify existing karaf feature to which this change applies and/or new karaf features being introduced. These can be user facing features which are added to integration/distribution or internal features to be used by other projects.

REST API

Sample JSONS/URIs. These will be an offshoot of yang changes. Capture these for User Guide, unit tests etc.

CLI

Any CLI if being added.

1.1.4 Implementation

Assignee(s)

Who is implementing this feature? In case of multiple authors, designate a primary assigne and other contributors.

Primary assignee:
<developer-a>

Other contributors:
<developer-b> <developer-c>

Work Items

Break up work into individual items. This should be a checklist on Trello card for this feature. Give link to trello card or duplicate it.

1.1.5 Dependencies

Any dependencies being added/removed? Dependencies here refers to internal [other ODL projects] as well as external [OVS, karaf, JDK etc.] This should also capture specific versions if any of these dependencies. e.g. OVS version, Linux kernel version, JDK etc.

This should also capture impacts on existing project that depend on InfraUtils. Following projects currently depend on Infrautils: * Netvirt * GENIUS

1.1.6 Testing

Capture details of testing that will need to be added.

Unit Tests

Integration Tests

1.1.7 Documentation Impact

What is impact on documentation for this change? If documentation change is needed call out one of the <contributors> who will work with Project Documentation Lead to get the changes done.

Don't repeat details already discussed but do reference and call them out.

1.1.8 References

Add any useful references. Some examples:

- Links to Summit presentation, discussion etc.
- Links to mail list discussions
- Links to patches in other projects
- Links to external documentation

[1] OpenDaylight Documentation Guide

[2] https://specs.openstack.org/openstack/nova-specs/specs/kilo/template.html

---

Note: This template was derived from [2], and has been modified to support our project.

Table of Contents

1.2 Status And Diagnostics Framework

https://git.opendaylight.org/gerrit/#/q/topic:s-n-d

Status reporting is an important part of any system. This document explores and describes various implementation options for achieving the feature.

1.2.1 Problem description

Today ODL does not have a centralized mechanism to do status and diagnostics of the various service modules, and have predictable system initialization. This leads to a lot of confusions on when a particular service should start acting upon the various incoming system events, because in many cases(like restarts) services end up doing premature service handling.

The feature aims at developing a status and diagnostics framework for ODL, which can :

- Orchestrate predictable system initialization, by enabling external interfaces, including northbound and southbound interfaces depending on a set of selected services declaring their availability. This in turn can prevent the system from processing northbound (eg: OpenStack), or southbound (eg: OVSDB or OpenFlow) events prematurely before all services are ready.
- Perform continuous monitoring of registered modules or internal services to ensure overall health of the system. This can additionally trigger alarms, or node reboots when individual services fail.

Use Cases

This feature will support following use cases:

- Use case 1: diagstatus module uses apis to create an application which can declare the system as UP.
- Use case 2: Core services can include existing netvirt and genius services like ELAN, L3VPN, ITM, interface-manager, DataStore and additional services may be ACL, QoS etc as needed. Applications can take necessary actions based on the aggregate system status, for eg: OpenFlow port open, OVSDB port open, and S&D status update(for consumption by other NBs such as ODL Mechanism Driver)
- Use case 3: Registered Service Modules should expose their status to diagstatus module which inturn will use this information to expose the service status to others.
- Use case 4: All southbound plugins should leverage the status provided by diagstatus module, as well as config file settings, to block or unblock the southbound interface
- Use case 5: diagstatus module should monitor the health of all dependant services on a regular basis using JMX.
- Use case 6: diagstatus module should raise traps whenever health check on a module fails.
- Use case 7 : diagstatus module should develop the capability to do a node/cluster reboot in future for scenarios mentioned in usecase 6.
- Use case 8 : diagstatus module should leverage on the counters support provided by infrautils to expose some debug and diagnostics counters.

1.2.2 Proposed change

The proposed feature adds a new module in infrautils called “diagstatus”, which allows CLI or alternative suitable interface to query the status of the services running in context of the controller (interface like Openflow, OVSDB, ELAN,ITM, IFM, Datastore etc.). This also allows individual services to push status-changes to this centralized module via suitable API-based notification. There shall be a generic set of events which application can report to the central monitoring module/service which shall be used by the service to update the latest/current status of the services.

Service startup Requirements

- Since the statuses are stored local to the node and represents the states of individual services instances of the node, there is no data-sync-up requirements for this service
- When the service starts-up, required local map for managing service-wise status entries shall be initialized
- It must be ensured that the status-monitoring-service starts-up fast as service whenever is started/re-booted.

Service API requirements

Status model object encapsulating the metadata of status such as:

- Node-name – may be this could be populated internally by framework if the node-name is available from within the framework with lesser / no external dependencies
- Module-name – populated by status-reporting module
- Service-name – populated by status-reporting module
- Service-status – populated by status-reporting module
- Current timestamp – internally populated

- Status Description – Any specific textual content which service can add to aid better troubleshooting of reported status

Service Internal Functionality Requirements

- Data for current status of the changes alone must be maintained. Later we can improve it to maintain history of statuses for a given service
- Since the statuses of services are dynamic there is no persistence requirement to store the statuses
- Status entry of given service shall be updated based on the metadata of provided by services
- Entries for service statuses shall be created lazily - if they are not already present, as and when first API invocation is made by the application-module towards the status/health monitoring service
- Read APIs of Monitoring-Service expose the service statuses on per cluster-node basis only. A separate module shall be developed as part of "cluster-services" user-story which can combine cross-cluster status collation
- All output of the read-APIs shall return results as Map with URI as key and current service-status and last-update timestamp combined as value
- In order to check the status of registered services, Status-Monitoring Service shall use standard scheduled timer service to invoke status-check callback on registered services
- Scheduled probe timer interval shall be configurable in config.ini. Any changes to this configuration shall require the system restart

Service Shutdown Requirements

- Currently no specific requirements around this area as restarting or node moving to quiescent state results in loss of all local data

Instrumentation Requirements

Applications must invoke status-reporting APIs as required across the lifecycle of the services in start-up, operational and graceful shutdown phases In order to emulate a simpler state-machine, we can have services report following statuses * STARTING – at the start of onSessionInitiated() on instrumented service * OPERATIONAL – at the end of onSessionInitiated() on instrumented service * ERROR – if any exception is caught during the service bring-up of if the service goes into an ERROR state dynamically * REGISTER – on successful registration of instrumented service * UNREGISTER – when a service does unregister from diagstatus on its own

YANG changes

N/A

Workflow

Register Service

Whenever the new service comes up, the service provider should register new service in service registry.

Report Status

Application can report their status using diagstatus APIs

Read Service Status

Whenever applications/CLI try to fetch the service status, diagstatus module will query the status through the respective Service implementations exposed by each service, and an aggregated result is provided as response.

Clustering considerations

- The CLIs/APIs provided by diagstatus module will be cluster wide.
- Every node shall expose a Status Check MBean for querying the current status which is local to the node being queried.
- Every node shall also expose a Clusterwide Status Check MBean for querying the clusterwide Status of services.
- For local status CLI shall query local MBean.
- For clusterwide status CLI shall query local MBean AS WELL AS and remote MBean instances across all current members of the cluster by accessing respective PlatformMBeanServer locally and remotely.
- It is assumed that IP Addresses of the current nodes of cluster and standard JMX Port details are available for clusterwide MBeans
- CLI local to any of the cluster members shall invoke clusterwide MBean on ANY ONE of current set of cluster nodes
- Every node of cluster shall query all peer nodes using the JMX interface and consolidate the statuses reported by each node of cluster and return combined node-wise statuses across the cluster

Other Infra considerations

N.A.

Security considerations

N.A.

Scale and Performance Impact

N/A as it is a new feature which does not impact any current functionality.

Targeted Release(s)

Carbon.

Known Limitations

The initial feature will not have the health check functionality. The initial feature will not have integration to infrautils counter framework for dispalying diag-counters.

Alternatives

N/A

1.2.3 Usage

Features to Install

This feature adds a new karaf feature, which is odl-infrautils-diagstatus.

JAVA API

Following are the service APIs which must be supported by the Framework :

- Accept Service-status from services which invoke the framework
- Get the current statuses of all services of a given cluster-node
- A registration API to allow monitored service to register the callback
- An interface which is to be implemented by monitored module which could be periodically invoked by Status-Monitoring framework on each target module to check status
- Each service implements their own logic to check the local-health status using the interface and report the status

CLI

Following CLIs will be supported as part of this feature:

- showSvcStatus - get remote service status

OSGI Services

Following osgi services will be supported as part of this feature:

- DiagStatusService - provides APIs for application to register and unregister services and report service status
- ServiceStatusProvider - provide information of registered service from ServiceDescriptor

1.2.4 Implementation

Assignee(s)

Primary assignee:
<Faseela K>

Other contributors:
<Nidhi Adhvaryu>

Work Items

1. spec review
2. diagstatus module bring-up
3. API definitions
4. Aggregate the status of services from each node
5. Migrate All Application to Diagstatus
6. Integrate all application
7. Add CLI.
8. Add UTs.
9. Add Documentation

1.2.5 Dependencies

This is a new module and requires the below libraries:

- org.apache.maven.plugins
- com.google.code.gson
- com.google.guava

This change is backwards compatible, so no impact on dependent projects. Projects can choose to start using this when they want.

Following projects currently depend on InfraUtils:

- Netvirt
- Genius

1.2.6 Testing

Unit Tests

Appropriate UTs will be added for the new code coming in once framework is in place.

Integration Tests

Since Component Style unit tests will be added for the feature, no need for ITs

CSIT

N/A

1.2.7 Documentation Impact

This will require changes to User Guide and Developer Guide.

User Guide will need to add information on how to use status-and-diag APIs and CLIs

Developer Guide will need to capture how to use the APIs of status-and-diag module to derive service specific actions. Also, the documentation needs to capture how services can expose their status via Mbean and integrate the same to status-and-diag module

1.2.8 References

- https://wiki.opendaylight.org/view/Infrastructure_Utilities:Carbon_Release_Plan

CHAPTER

TWO

INFRAUTILS GETTING STARTED GUIDE

Table of Contents

2.1 Infrautils Features

This project offers technical utilities and infrastructures for other projects to use.

The conference presentation slides linked to in the references section at the end give a good overview of the project.

Check out the JavaDoc on https://javadocs.opendaylight.org/org.opendaylight.infrautils/fluorine/.

2.1.1 Features

@Inject DI

See https://wiki.opendaylight.org/view/BestPractices/DI_Guidelines

Utils incl. org.opendaylight.infrautils.utils.concurrent

Bunch of small (non test related) low level general utility classes à la Apache (Lang) Commons or Guava and similar incl. utils.concurrent:

- ListenableFutures toCompletionStage & addErrorLogging
- CompletableFutures completedExceptionally
- CompletionStages completedExceptionally
- LoggingRejectedExecutionHandler, LoggingThreadUncaughtExceptionHandler, ThreadFactoryProvider, Executors newSingleThreadExecutor

Test Utilities

- LogRule which logs (using slf4j-api) the start and stop of each @Test method
- LogCaptureRule which can fail a test if something logs any ERROR from anywhere (This work even if the LOG.error() is happening in a background thread, not the test's main thread… which can be particularly interesting.)
- RunUntilFailureRule which allows to keep running tests indefinitely; for local usage to debug "flaky" (sometimes passing, sometimes failing) tests until they fail
- ClasspathHellDuplicatesCheckRule verifies, and guarantees future non-regression, against JAR hell due to duplicate classpath entries. Tests with this JUnit Rule will fail if their classpath contains duplicate class. This could be caused e.g. by changes to upstream transitive dependencies. See also http://jhades.github.io (which this internally uses, not exposed). see https://github.com/opendaylight/infrautils/blob/master/testutils/src/test/java/org/opendaylight/infrautils/testutils/tests/ExampleTest.java
- Also some low level general utility classes helpful for unit testing concurrency related things in infrautils.testutils.concurrent:
  - AwaitableExecutorService
  - SlowExecutor
  - CompletionStageTestAwaiter, see CompletionStageAwaitExampleTest

Job Coordinator

JobCoordinator service which enables executing jobs in a parallel/sequential fashion based on their keys.

Ready Service

Infrastructure to detect when Karaf is ready.

The implementation internally uses the same Karaf API that e.g. the standard "diag" Karaf CLI command uses. This checks both if all OSGi bundles have started as well if their blueprint initialization has been successfully fully completed.

It builds on top of the bundles-test-lib from odlparent, which is what we run as SingleFeatureTest (SFT) during all of our builds to ensure that all projects' features can be installed without broken bundles.

The infrautils.diagstatus modules builds on top of this infrautils.ready.

What infrautils.ready adds on top of the underlying raw Karaf API is operator friendly logging, a convenient API and a correctly implemented polling loop in a background thread with SystemReadyListener registrations and notifications,

instead of ODL applications re-implementing this. The infrautils.ready project intentionally API isolates consumers from the Karaf API. We encourage all ODL projects to use this infrautils.ready API instead of trying to reinvent the wheel and directly depending on the Karaf API, so that application code could be used outside of OSGi, in environment such as unit and component tests, or something such as honeycomb.

Applications can use this SystemReadyMonitor registerListener(SystemReadyListener) in a constructor to register a listener for and get notified when all bundles are "ready" in the technical sense (have been started in the OSGi sense and have completed their blueprint initialization), and could on that event do any further initialization it had to delay in the original blueprint initialization.

This cannot directly be used to express functional dependencies BETWEEN bundles (because that would deadlock infrautils.ready; it would stay in BOOTING forever and never reach SystemState ACTIVE). The natural way to make one bundle await another is to use Blueprint OSGi service dependency. If there is no technical service dependency but only a logical functional one, then in infrautils.ready.order there is a convenience sugar utility to publish "marker" FunctionalityReady interfaces to the OSGi service registry; unlike real services, these have no implementing code, but another bundle could depend on one to enforce start up order one (using regular Blueprint <reference> in XML or @Reference annotation).

A known limitation of the current implementation of infrautils.ready is that its "wait until ready" loop runs only once, after installation of infrautils.ready (by boot feature usage, or initial single line feature:install). So SystemState will go from BOOTING to ACTIVE or FAILURE, once. So if you do more feature:install after a time gap, there won't be any further state change notification; the currently implementation won't "go back" from ACTIVE to BOOTING. (It would be absolutely possible to extend SystemReadyListener onSystemBootReady() with an onSystemIsChanging() and onSystemReadyAgain(), but the original author has no need for this; as "hot" installing additional ODL application features during operational uptime was not a real world requirement to the original author. If this is important to you, then your contributions for extending this would certainly be welcome.)

infrautils' ready, like other infrautils APIs, is available as a separate Karaf feature. Downstream projects using infrautils.ready will therefore NOT pull in other bundles for other infrautils functionalities.

Integration Test Utilities (itestutils)

See https://bugs.opendaylight.org/show_bug.cgi?id=8438 and https://git.opendaylight.org/gerrit/#/c/56898/

Used for non-regression self-testing of features in this project (and available to others).

Diagstatus

To be documented.

2.1.2 References

[2] ODL DDF - LA 2018

[3] ODL DDF 2017

[4] infrautils JavaDoc