ODL InfraUtils

Release master

Sep 18, 2019

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 1

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

- *References*

1.2 Job Coordinator

https://git.opendaylight.org/gerrit/#/q/topic:JC

Job Coordinator is a framework for executing jobs in sequential/parallel based on their job-keys. One such job,to give an example, can be for MD-SAL config/operational datastore updates.

1.2.1 Problem description

The concept of datastore jobcordinator was derived from the following pattern seen in many ODL project implementations :

- The Business Logic for the configuration/state handling is performed in the Actor Thread itself. This will cause the Actor's mailbox to get filled up and may start causing unnecessary back-pressure.
- Actions that can be executed independently will get unnecessarily serialized. Can cause other unrelated applications starve for chance to execute.
- Available CPU power may not be utilized fully. (for instance, if 1000 interfaces are created on different ports, all 1000 interfaces creation will happen one after the other.)
- May depend on external applications to distribute the load across the actors.

Use Cases

This feature will support following use cases:

- Use case 1: JC framework should enable applications to enqueue their jobs based on a job key.
- Use case 2: JC framework should run jobs queued on same key sequentially, and different keys parallelly.
- Use case 3: JC framework should provide a framework for retry mechanism in case the jobs fail.
- Use case 4: JC framework should provide a framework for rollback in case the jobs fail permanently.
- Use case 3: JC should provide applications the flexibility to input the number of retries on a need basis.

1.2.2 Proposed change

The proposed feature adds a new module in infrautils called "jobcoordinator", which will have the following functionalities:

- "Job" is a set of operations, (eg : updates to the Config/Operational MD-SAL Datastore)
- Dependent Jobs [eg. Operations on interfaces on same port] that need to be run one after the other will continue to be run in sequence.
- Independent Jobs [eg. Operations on interfaces across different Ports] will be allowed to run parallel.
- Makes use of ForkJoin Pools that allows for work-stealing across threads. ThreadPool executor flavor is also available. But would be deprecating that soon.
- Jobs are enqueued and dequeued to/from a Hash structure that ensures point 2 & 3 above are satisfied and are executed using the ForkJoinPool mentioned in point 4.

- The jobs are enqueued by the application along with an application job-key (type: string). The Coordinator dequeues and schedules the job for execution as appropriate. All jobs enqueued with the same job-key will be executed sequentially.
- Job Coordination function gets the list of listenable futures returned from each job.
- The Job is deemed complete only when the onSuccess callback is invoked and the next enqueued job for that job-key will be dequeued and executed.
- On Failure, based on application input, retries and/or rollback will be performed. Rollback failures are considered as double-fault and system bails out with error message and moves on to the next job with that Job-Key.

YANG changes

N/A

Workflow

Define Job Workers

Applications can define their own worker threads for their job. A job is defined as a piece of code that can be independently executed.

Define Rollback Workers

Applications should define a rollback worker, which will have the code to be executed in case the main job fails permanently. In usual scenarios, this will be the code to clean up all partially completed transactions by the main worker.

Decide Job Key

Applications should carefully choose the job-key for their job worker. All jobs based on the same job-key will be executed sequentially, and all jobs on different keys will be executed parallelly depending on the available threadpool size.

Enqueue Job

Applications can enqueue their job worker to JC framework for execution.JC has a hash structure to handle the execution of the tasks sequentially/parallelly. Whenever a job is enqueued, JC creates a Job Entry for the particular job. A Job Entry is characterized by - job-key, the main worker, the rollback worker and the number of retries. This JobEntry will be added to a JobQueue, which inturn is part of a JobQueueMap.

Job Queue Handling

There is a JobQueueHandler task which runs periodically, which will poll each of the JobQueues to execute the main task of the corresponding JobEntry. Within a JobQueue, execution will be synchronized.

Retries in case of failure

The list of listenable futures for the transactions from the application main worker will be available to JC, and if at all the transaction fails, the main worker will be retried the 'max-retries' number of times which is application specified. If all the retries fail, JC will bail out and the rollback worker will be executed.

Configuration impact

N/A

Clustering considerations

- Job Coordinator synchronization is not cluster-wide
- This will still work in a clustered mode by handling optimistic lock exceptions and retrying of the job.
- Future scope can be : Cluster-Wide Datastore & Switch Job Coordination in:
- Fully replicated Followers also listening Mode.
- Distributed system where no. of replicas is less than the no. of nodes in the cluster.

Other Infra considerations

N.A.

Security considerations

N.A.

Scale and Performance Impact

This feature is aiming at improving the scale and performance of applications by providing the cabability to execute their functions parallelly wherever it can be done.

Targeted Release(s)

Carbon.

Known Limitations

JC synchronization is not currently clusterwide.

Alternatives

N/A

1.2.3 Usage

Features to Install

This feature doesn't add any new karaf feature.

REST API

N/A

CLI

N/A

JAVA API

JobCoordinator provides the below APIs which can be used by other applications:

```
void enqueueJob(String key, Callable<List<ListenableFuture<Void>>> mainWorker).

void enqueueJob(String key, Callable<List<ListenableFuture<Void>>> mainWorker,␣
↪RollbackCallable rollbackWorker).

void enqueueJob(String key, Callable<List<ListenableFuture<Void>>> mainWorker, int␣
↪maxRetries).

void enqueueJob(String key, Callable<List<ListenableFuture<Void>>> mainWorker,␣
↪RollbackCallable rollbackWorker,
        int maxRetries).
```

key is the JobKey for synchronization, mainWorker will be the actual Job Task, maxRetries is the number of times a Job will be retried if the mainWorker results in ERROR, rollbackWorker is the Task to be executed if the Job fails with any ERROR maxRetries times.

1.2.4 Implementation

Assignee(s)

Primary assignee: <Periyasamy Palanisamy>

Other contributors: <Yakir Dorani> <Faseela K>

Work Items

1. spec review.
2. jobcoordinator module bring-up.
3. API definitions.
4. Enqueue Job Implementation.
5. Job Queue Handler Implementation.

6. Job Callback Implementation including retry and rollback
7. Add CLI.
8. Add UTs.
9. Add Documentation.

1.2.5 Dependencies

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

N/A

CSIT

N/A

1.2.7 Documentation Impact

This will require changes to Developer Guide.

Developer Guide can capture the new set of APIs added by JobCoordinator as mentioned in API section.

1.2.8 References

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

Table of Contents

1.3 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.

1.3.1 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.

1.3.2 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

1.3.3 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

1.3.4 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

1.3.5 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

1.3.6 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

1.3.7 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

1.3.8 References

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

Table of Contents

1.4 Infrautils Caches

https://git.opendaylight.org/gerrit/#/q/topic:bug/8300 https://www.youtube.com/watch?v=h4HOSRN2aFc

Infrautils Caches provide a Cache of keys to values. The implementation of Infrautils Cache API is, typically, backed by established cache frameworks,such as Ehcache, Infinispan, Guava's, Caffeine, . . . , imcache, cache2k, . . . etc.

1.4.1 Problem description

Caches are not Maps!. Differences include that a Map persists all elements that are added to it until they are explicitly removed. A Cache on the other hand is generally configured to evict entries automatically, in order to constrain its memory footprint, based on some policy. Another notable difference, enforced by this caching API, is that caches should not be thought of as data structures that you put something in somewhere in your code to get it out of somewhere else. Instead, a Cache is "just a façade" to a CacheFunction's get. This design enforces proper encapsulation, and helps you not to screw up the content of your cache (like you easily can, and usually do, when you use a Map as a cache).

Use Cases

This feature will support following use cases:

- Use case 1: Enable creation of a brand new Cache, based on the passed configuration and policy.
- Use case 2: Enable creation of a brand new Cache, based on the passed configuration, with a default policy.
- Use case 3: Cache should allow to evict an entry based on the eviction policy.
- Use case 4: Cache should allow to put a new entry to the cache.
- Use case 5: InfraUtils Cache should expose APIs to retrieve the Cache Policies, Stats, and Fixed Configuration.
- Use case 6: InfraUtils Cache should expose APIs to set the Cache Policies, Stats, and Fixed Configuration.

1.4.2 Proposed change

The proposed feature adds a new module in infrautils called "caches", which will have the following functionalities:

- "InfraUtils Cache" has two variants - Cache and CheckedCache. Both are caches of key to values, with CheckedCache having support for cache function which may throw a checked exception.
- Cache can be configured to evict entries automatically, in order to constrain its memory footprint, based on some user configured policy.

- Cache implementation internally based on Guava Cache will be available(other implementations would be possible, later)
- Complete Karaf example with cache usage will be available (see infrautils/caches/sample)
- Karaf CLIs will be made available for cache operations

YANG changes

N/A

Workflow

Define a Cache

Applications can define their own Cache, with specified CacheFunction and Policies. Caches can be configured to set the maximum entries and also to enable stats.

Define Cache Function

Cache is "just a façade" to a CacheFunction's get(). When the user defines a CacheFunction for his cache, that will be executed whenever a get() is executed on the cache.

Define Anchor

Anchor refers to instance of the class "containing" this Cache. It is used by CacheManagers to display to end-user.

Define Cache Id and Description

Cache id is a short ID for this cache, and description will be a one line human readable description for the cache.

Cache Eviction

Cache Eviction Policy is based on the number of entries the cache can hold, which will be set during the cache creation time.

Use Cache

Configuration impact

N/A

Clustering considerations

- The current Cache Infra is node local.
- Future enhancements can be made by providing clustered backend implementations e.g. Infinispan.

Other Infra considerations

N.A.

Security considerations

N.A.

Scale and Performance Impact

This feature is aiming at improving the scale and performance of applications by helping to define a CacheFunction for heavy operations.

Targeted Release(s)

Carbon.

Known Limitations

Cache is currently neither distributed (cluster wide) nor transactional."

Alternatives

N/A

1.4.3 Usage

Features to Install

odl-infrautils-caches odl-infrautils-caches-sample

REST API

N/A

CLI

cache:clear cache:list cache:policy cacheID policyKey policyValue

JAVA API

Caches provides the below APIs which can be used by other applications:

```
CacheProvider APIs

<K, V> Cache<K, V> newCache(CacheConfig<K, V> cacheConfig, CachePolicy initialPolicy);
<K, V> Cache<K, V> newCache(CacheConfig<K, V> cacheConfig);
<K, V, E extends Exception> CheckedCache<K, V, E> newCheckedCache(
            CheckedCacheConfig<K, V, E> cacheConfig, CachePolicy initialPolicy);
<K, V, E extends Exception> CheckedCache<K, V, E> newCheckedCache(CheckedCacheConfig
↪<K, V, E> cacheConfig);

CacheManager APIs

BaseCacheConfig getConfig();
CacheStats getStats();
CachePolicy getPolicy();
void setPolicy(CachePolicy newPolicy);
void evictAll();
```

1.4.4 Implementation

Assignee(s)

Primary assignee: <Michael Vorburger>

Work Items

1. spec review.
2. caches module bring-up.
3. API definitions.
4. Cache Policy Implementation.
5. Cache and CheckedCache Implementation.
6. Backend Implementation
7. Add CLI.
8. Add UTs.
9. Add Documentation.

1.4.5 Dependencies

Following projects currently depend on InfraUtils:

- Netvirt
- Genius

1.4.6 Testing

Unit Tests

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

Integration Tests

N/A

CSIT

N/A

1.4.7 Documentation Impact

This will require changes to Developer Guide.

Developer Guide can capture the new set of APIs added by Caches as mentioned in API section.

1.4.8 References

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

CHAPTER 2

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).

Caches

See https://www.youtube.com/watch?v=h4HOSRN2aFc and play with the example in infrautils/caches/sample installed by odl-infrautils-caches-sample; history in https://git.opendaylight.org/gerrit/#/c/48920/ and https://bugs.opendaylight.org/show_bug.cgi?id=8300.

Diagstatus

To be documented.

Metrics

infrautils.metrics offers a simple back-end neutral API for all ODL applications to report technical as well as functional metrics.

There are different implementations of this API allowing operators to exploit metrics in the usual ways - aggregate, query, alerts, etc.

The odl-infrautils-metrics Karaf feature includes the API and the local Dropwizard implementation.

The odl-infrautils-metrics-sample Karaf feature illustrates how to use metrics in application code, see metrics-sample sources in infrautils/metrics/sample/impl.

Metrics API

Application code uses the org.opendaylight.infrautils.metrics.MetricProvider API, typically looked up from the OSGi service registry using e.g. Blueprint annotations @Inject @Reference, to register new Meters (to "tick/mark events" and measure their rate), Counters (for things that go up and down again), and Timers (to stop watch durations). Support for "Gauges" is to be added; contributions welcome.

Each metric can be labeled, possibly along more than one dimension.

The org.opendaylight.infrautils.metrics.testimpl.TestMetricProviderImpl is a suitable implementation of the MetricProvider for tests.

Metrics Dropwizard Implementation

Based on Dropwizard Metrics (by Coda Hale at Yammer), see http://metrics.dropwizard.io, exposes metrics to JMX and can regularly dump stats into simple local files; background slide https://codahale.com/codeconf-2011-04-09-metrics-metrics-everywhere.pdf

This implementation is "embedded" and requires no additional external systems.

It is configured via the local configuration file at etc/org.opendaylight.infrautils.metrics.cfg.

This includes a threads deadlock detection and maximum number of threads warning feature.

Metrics Prometheus Implementation

Implementation based on Linux Foundation Cloud Native Computing Foundation Prometheus, see https://prometheus.io

This implementation exposes metrics by HTTP on /metrics/prometheus from the local ODL to an external Prometheus set up to scrape that.

This presentation given at the OpenDaylight Fluorine Developer Design Forum in March 2018 at ONS in LA gives a good overview about the infrautils.metrics.prometheus implementation.

This implementation requires operators to separatly install Prometheus, which is not a Java OSGi application that can be feature:install into Karaf, but an external application (via Docker, RPM, tar.gz etc.). Prometheus is then configured with the URL of ODL nodes, and "scrapes" metrics from ODL in configurable regular intervals. Prometheus is

extensibly configurable for typical metrics use cases, including alerting, and has existing integrations with other related systems.

The odl-infrautils-metrics-prometheus Karaf feature install this. It has to be installed by feature:install or featuresBoot, BEFORE any ODL application feature which depends on the odl-infrautils-metrics feature (similarly to e.g. odl-mdsal-trace)

2.1.2 References

[1] Infrautils Metrics Prometheus Implementation

[2] ODL DDF - LA 2018

[3] ODL DDF 2017

[4] infrautils JavaDoc