Lighty Project

Name

lighty

Repo Name

lighty

Description

lighty is designed as a simple, lightweight and easy-to-use set of ODL components.

lighty is NOT a framework. It is a set of reusable libraries and/or components. That is why it can be easily integrated with various libraries and frameworks in the vast Java ecosystem. The end-user has to have a possibility to re-use *lighty* components as separate libraries, or as a whole pre-assembled stack.

lighty components are ready to be used in micro-service deployments.

Scope

The *lighty* project starts and provides services from core ODL projects (like controller, MD-SAL, YANG Tools) in a way, where they can be used without the dependency on OSGi frameworks like Karaf and Blueprint DI. The main benefits of using *lighty* to get these services, instead of OSGi, are:

- simpler development
- faster build process
- better maintainability
- distribution with a smaller footprint
- fewer dependencies
- easier integration with other libraries/frameworks.

All services/beans from these core ODL projects are created and started in a common and simple Java way - by creating instances with *constructors*. By default, no dependency injection framework is used. Services created this way are exposed to other projects/modules via simple *get* methods.

A base building block of applications, based on *lighty*, is called a **LightyModule**. LightyModule is an interface that provides *start/stop* methods to this module. Implementations, or extensions of LightyModule, may expose started services for use in another LightyModule if needed.

One LightyModule represents one comprehensive functionality. There can be one or more LightyModules per project - if it makes sense from a functionality perspective.

Dependencies of the implementation of LightyModule are specified in its Java constructor. This constructor specifies, which services must be started, before this LightyModule.

All LightyModules should be fully configurable.

Runnable applications based on *lighty* (so-called LightyApplication), consist of multiple LightyModules. These modules should be created and started in an order, which is defined by the dependencies of the LightyModules (by services required in constructors of LightyModules). A simple Java *main* method, used as a LightyApplication, is sufficient.

LightyApplications can be distributed as .zip archive, that can be generated by the Maven parent *lighty-app-parent*. This .zip distribution will contain a runnable .jar file of the application, a configuration and all necessary dependencies (as .jar files).

Projects creating their LightyModules (and LightyApplication), cannot use OSGi libraries in their code (*import org.osgi.**) and all of their ODL dependencies (for example MD-SAL, RESTCONF, etc.) must be already integrated with *lighty*.

Open questions

There are a number of unresolved questions, which need to be addressed before we move into execution and rollout of this project. These are summarized in this section.

Relationship with other projects

Current implementation assumes it is downstream of all projects that are integrated. This lays the onus of ensuring integration and compliance into this project, with little in the way of validating whether a particular upstream is compliant with the requirements and can actually be integrated.

Furthermore this means there is an unclear governance around use-case packaging. We certainly do not want for this project to become an architectural hotspot the way controller's 'odl-mdsal-broker' feature is a hotspot for current Karaf packaging.

Monolithic vs. modular services

Current Lighty approach assumes all services present in odl-mdsal-broker are required for a particular deployment. This is not actually required -- and is perhaps detrimal to actual use cases -- for example there are use cases which do not need Entity Ownership Service, and hence it should be possible to create a use case package without those components. This has a direct tie-in into how services are injected, below.

Dependency Injection

Current ODL approach is centered around OSGi Service Registry being present, with some amount of leakage from ODL-simple, where there are some JSR250-annotated classes and generated BluePrint bindings. The ODL-simple approach here is to introduce a hard dependency on Guice/Mycila as a replacement for dynamic/BP services. Mycila is a dead project, so any attempt to reuse this approach should re-visit the integration to use an actively-maintained upstream.

The overall approach here need to be well defined, so as not to tie us to a single DI framework, as there is a number of different frameworks available:

- Guice, which is in heavy maintenance mode, possibly restricting our use of JPMS
- JMPS itself, which provides ServiceLoader based service injection
- Dagger, which seems to be a thriving project, but has not been triaged for usefulness
- Jakarta CDI, which may provide some amount of standardization, especially as we have moved out of the Java 8/9 flux

Dynamic services

Components leveraging Entity Ownership Service and Cluster Singleton Service have an inherently dynamic lifecycle which, at least in the case of BGPCEP, involves instantiating services in OSGi Service Registry and having components react to them. In the case of BGPCEP this is the case of RIB instance becoming master and Topology Manager starting local node processing. We need to have a solid answer on how to migrate these use cases without having each project sprout their own dynamic lifecycle API/implementation component, as those are bound to be buggy.

Deployment configuration

A number of our components currently rely on OSGi Config Admin interface to provide deployment-specific 'static' configuration. This configuration would typically be read once during ODL startup although Karaf does have provisions for updating this at runtime. A static wiring component should provide a well-standardized interface, so that individual projects do not end up rolling their own "solution", wreaking havoc to users being able to integrate this ability.

Dynamic configuration

Some components rely on dynamic configuration, which is typically held in MD-SAL, but there are also separate mechanisms invented to deal with some other requirements -- for example BGPCEP's config-loader components and NETCONF's model side-loading capability. These should be analyzed and strict guidelines on lifecycle should be created, so that interactions with persistence are clearly defined, especially in terms of lifecycle interaction.

CLI integration

Karaf provides a CLI infrastructure, which is deemed useful for a number of projects. The need for a CLI in cloud-native deployments is to be determined and this item needs to be addressed.

Resources Committed (developers committed to working)

Samuel Kontriš - *samuel.kontris@pantheon.tech*

Róbert Varga - *robert.varga@pantheon.tech*

Martin Bobák - *martin.bobak@pantheon.tech*

Tejas Nevrekar - *tejas.nevrekar@gmail.com*

Balaji Varaaraju "bvaadar@luminanetworks.com"

Jamo Luhrsen "jluhrsen@luminanetworks.com"

Initial Committers

Samuel Kontriš - *samuel.kontris@pantheon.tech*

Róbert Varga - *robert.varga@pantheon.tech*

Martin Bobák - *martin.bobak@pantheon.tech*

Vendor Neutral

Current codebase - github.com/PantheonTechnologies/lighty-core

License is compatible (Eclipse Public License - v 1.0) and packages will be renamed from *io.lighty.** to *org.opendaylighty.lighty.**

Meets Board Policy (including IPR)