Resilient micro service packaging

We are testing new emerging workload (e.g. Big Data, Openstack etc.) on VMWare SDDC platform. In current QE automation, all test cases are compiled together as one java application. One typical case is that given a certain vCenter information, ovftool utility is triggered first to deploy Big Data Extension OVA, followed by management server configuration, big data cluster creation, cluster verification, day 2 operations and final cleanup tasks. A test case is designed for it, and furthermore some generic methods may be well refactored as common java methods. It works well for majority of QE jobs like CI, but some limitations are seen in practice:

- Programming language bindings: extra big effort required to provide language neutral interfaces
- Interface re-entering: imagine this scenario that a cluster verification tool is required for existing big data clusters, either manually built-up or patched/upgraded/restored. Test case mentioned above no longer tackles it. QE needs to add more and more cases in Java to do support, or re-visit underneath codes and repeat them manually for ad-hoc situation.
- Portability: big effort to deliver QE automation to other engineering teams and customers, especially regarding adoption and patching

It's practical to overcome the problems by exposing internal java methods to public REST APIs and hosting these REST services in certain end points. Test case is changed to a workflow of calling REST services and each invocation is language neutral. In short, it's a conversion from applications to services.

There are quite a few internal functions to be adopted, by different individuals for different projects. It's ideal that new service API is easy to develop and maintain for long term. Service modularity looks a good approach, by isolating functions between service framework and concrete services, as well as services for different components. It's turned to a micro service system in design now, other than previous monolithic application for QE automation.

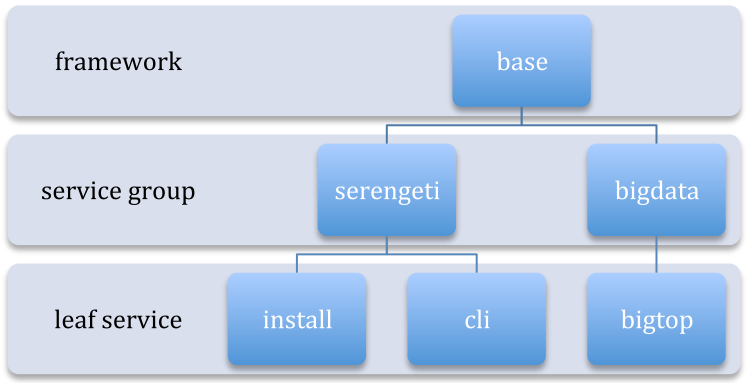
In this paper, a new build & packaging idea with prototype is described based on this micro service framework. The original purpose was very simple: per a bundle of fine-grained REST services, we may run them in different end points (JVM listening on a host/port pair for this moment), but it's favorable if there is a way we can easily combine all or part of them into one end point for testing convenience. The idea expanded very soon: encapsulate micro services easily and elastically for each end point. Imagine that when we combine all services into one, it looks like monolithic and is handy for deployment, testing and a quick try by others.

The prototype is available at https://github.com/zhangc119/earay .

Each micro service is defined and developed in a java project with its own build file. Except project “base” which takes care of framework scope functions (Dropwizard, SwaggerUI etc.) and provides a few common REST APIs (File, Groovy etc.), all other projects are organized in two levels:

- Service group: parent of several associated leaf services
- Leaf service: aimed at specific REST services

The figure below illustrates build dependency among the projects in the prototype.



It’s straightforward to build a leaf project (use dependency tag in Maven, for example). Here the focus is the capability of encapsulating a variety of leaf services easily and flexibly. Imagine the scenarios like:

- Build all services into one package for local testing;
- Build one service group and relative leaf services (says “serengeti” group and “bigtop” in the figure above)
- Pick some leaf services (for example, “cli” and “bigtop” to provide REST APIs for bigtop cluster day 2 operations and verification)

In the prototype, these targets are easily reached by one option “-Papps” in build command. Let us walk through one sample build command : “./gradlew -p base -Papps=bigdata clean earayThinCapsule”.

- gradlew, a build utility in Gradle , which is a build tool similar to ant, Maven. Different from xml formatted build file in Maven, Groovy programing language is used in Gradle build file, which allows us to control build flows directly. Actually the magic in resilient micro service packaging exists in build.gradle files.
- “-p base”, target project is “base”, which means base/build.gradle is used
- “-Papps=”, which service groups and leaf services are built together. E.g.
    - “-Papps=bigdata” for service group “bigdata”
    - “-Papps=serengeti,bigdata:bigtop” for “serengeti” group and leaf “bigtop”
    - “-Papps=serengeti:cli,bigdata:bigtop” for leaves “cli” and “bigtop”
- “clean earayThinCapsule”, two gradle tasks to do re-packaging on all sources and dependencies.

In terms of implementation, the option "-Papps" is well interpreted in build.gradle under "base" and other projects. In base/build.gradle, several steps are added before code compiling if the option is on:

1. Make a project array by splitting ":" for the option
2. For each project, either service group or leaf service, append its compiling dependencies and source sets to project base
3. For project at service group level, iterate its children by repeating step #2

For other projects, remove dependency on base project when the option is on.

By manipulating source sets and dependencies in build scripts, we make it come true: a tool to package Java micro services easily and resiliently. The effort for newly added services is minimal: just copy few lines to the build file and let others be tackled in base project.

Another solution was considered to package micro services via dependency mechanism, but it turned out a failure due to circular dependency issue.

The project is growing. First more service projects will be added for routine QE tasks and a workflow engine is on demand for re-writing traditional QE automation. Then other methodologies for end points (Docker other than current JVM, some brain storming under way) will be visited. For longer term, a platform for service management, clustering for scale needs to be considered when service end points reach a big number. It's challenging, but why not? We are working on Cloud Platform.