Best Practices and Pitfalls for Building Products out of OpenDaylight

Colin Dixon, TSC Chair, OpenDaylight
Principal Software Engineer, Brocade
Devin Avery, Sr Staff Software Engineer, Brocade
Agenda

- Agenda / Introduction
- Common Pitfalls
- Best Practices
- Future Considerations
- Questions
Our experience productizing OpenDaylight

• Brocade has released multiple versions of a controller for Brocade
  • So far, based on Helium-SR1, Helium-SR2, and Helium-SR3
  • Rapidly approaching first Lithium-based release
  • Running in production with real customers

• We’ve been successful in doing this even when incorporating upstream changes

• Share our experiences -- pitfalls and successes

#ODSummit
Common Pitfalls
One way to build a release based on OpenDaylight

- Clone the OpenDaylight code (its public after all!)
- Modify the code to customize / brand
  - Add new code into the existing projects for your proprietary logic
- Run the build
- Test, Ship and Sell it!

- Great, that was easy...

- This is the first thing nearly everyone does
How (not) to build a custom fabric app and host tracker

MyFabric App

Flow Programming

Host Tracking + MyFabric Δs

Topology

#ODSummit
This doesn’t work!

merely

• By cloning OpenDaylight code you have essentially forked the code!

• Synchronizing code with upstream can be difficult
  • Your changes may not be accepted upstream (and you may not want them upstream)
  • Pulling changes from upstream is likely to result in constant merge conflicts
  • To avoid this pain, sync with upstream less often => increases pain => even less often

• Results in lower interaction in community since code base is out of sync
  • Don’t get bug fixes, security patches, etc.

• OpenDaylight source code is licensed under EPL – may put your proprietary changes at risk*

*Note: we are not lawyers. Be sure to discuss any legal / licensing issues with your legal team

#ODSummit
In Practice, this “forking” results in permanent divergence from upstream OpenDaylight. That is, you no longer have an OpenDaylight-based product in most of the ways you and your customers care about.
Best Practices
Treat OpenDaylight as a Third-party

- Don’t Download the OpenDaylight Source Code!
- Treat OpenDaylight as a collection read-only third-party artifacts
- Reference ODL artifacts, don’t build them
- Leverage maven to access, maintain and manage your third-party dependencies for you

For example:
- We don’t recompile the basic java.util.ArrayList Java class, we reference/use it
- We certainly don’t modify the java.util.ArrayList source code locally and build it
But OpenDaylight might not be a Third-party

• If you have modify OpenDaylight source code
  • Push the changes upstream
  • Wait for the next release—major or stability
  • Fold the changes in when bump your upstream versions

• What about critical bugfixes/patches?
  • Push the change upstream
  • *cherry-pick* the patches into a locally-created clone of the upstream repo
  • Build and “release” your own version of the relevant bundle(s)
  • *Note: this is complex and has risk, avoid it when possible.*
Make changes to OpenDaylight upstream!

• We need OpenDaylight’s core functionality to succeed
  • Otherwise our extensions, apps, etc. don’t matter

• You should
  • Provide bug fixes
  • Discuss new requirements
  • Share implementations, interfaces, extension points, etc.

• You will benefit from working more closely with upstream
  • Get patches/fixes faster
  • More likely have to have core OpenDaylight meet your needs faster

Dave Neary on Swimming Upstream:
http://www.slideshare.net/nearyd/swimming-upstream-45666354

#ODSummit
Leverage ODL Modularity / Isolate Proprietary Code

- Keep your proprietary code isolated to your own repositories, bundles, and Karaf features
- Leverage OSGi/Karaf/Maven to combine your code with OpenDaylight
- Leverage YANG/MD-SAL/Config system modularity to load proprietary implementations into the modeling system
- OSGi is generally friendly with the EPL license*

*Note: we are not lawyers. Be sure to discuss any legal / licensing issues with your legal team
Extension Example: Apps

My Fabric App

- Flow Programming
- Host Tracking
- Topology

OpenDaylight Code/Bundles
Your Proprietary Code/Bundles

#ODSummit
Extension Example: Replacing a Service

Write an OSGi bundle that populates the Host Tracker YANG model

Existing apps/services (either ODL or proprietary) will use the new implementation

Flow Programming

MyHostTracker

Topology

#ODSumbmit
Extension Example: Modifying an Existing Service

- Augment existing YANG models with new information
- Provide Java code to extend behavior
  - Requires Java extension points
- This is possible, but complex, needs thought and work

Slides from ONS talk:

#ODSummit
Stabilize Development Environment – No to Snapshots!

• Don’t build using OpenDaylight snapshots!
  • We don’t compile against beta versions of the JRE, we compile against released versions!

• Snapshots are volatile, and change constantly.
• Harder to determine if failures are related to your code, or snapshot change
• Customers want “stable” artifacts (won’t run on snapshots)
• If you use snapshots, then your release is tied directly to OpenDaylight release

• If you have time, you can compile dev versions of your code against dev versions of OpenDaylight to “scope out” potential issues
Stabilize Development Environment – No to Snapshots!

Your Company’s Artifacts

Released OpenDaylight Artifacts (Stable)

Active OpenDaylight Development (Volatile)

Ex: Multiple Proprietary Releases
The Result

- A product which references OpenDaylight artifacts

- Proprietary code is isolated, and can be updated without affecting OpenDaylight code
  - Also potentially avoid legal/licensing pain (ASK YOUR LAWYERS!)

- Changing versions of OpenDaylight code is just an update to a version in a pom file
  - ...and some testing

- Allows for a stable OpenDaylight controller on which to build proprietary implementations

#ODSummit
Future Considerations
# OD Summit

Focus more on our users

<table>
<thead>
<tr>
<th></th>
<th>Core Developer</th>
<th>App/Bus Log Dev</th>
<th>REST API Dev</th>
<th>Administrator</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Interface</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔✔✔</td>
</tr>
<tr>
<td>REST API</td>
<td>✔</td>
<td>✔</td>
<td>✔ ✔ ✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>App/Business Logic</td>
<td>✔</td>
<td>✔ ✔ ✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>MD-SAL/YANG</td>
<td>✔ ✔ ✔</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>South Bound Business Logic</td>
<td>✔ ✔ ✔</td>
<td>✔ ✔ ✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Meta-tasks, e.g., install &amp; upgrade</td>
<td>✔ ✔ ✔</td>
<td>✔ ✔ ✔</td>
<td>✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
</tr>
</tbody>
</table>
Focus more on our users

<table>
<thead>
<tr>
<th></th>
<th>Core Developer</th>
<th>App/Bus Log Dev</th>
<th>REST API Dev</th>
<th>Administration</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Interface</td>
<td>✔</td>
<td></td>
<td>✔ ✔ ✔</td>
<td>✔</td>
<td>✔✔✔</td>
</tr>
<tr>
<td>REST API</td>
<td>✔</td>
<td>✔</td>
<td>✔ ✔ ✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>App/Business Logic</td>
<td>✔</td>
<td>✔ ✔ ✔</td>
<td>✔ ✔ ✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>MD-SAL/YANG</td>
<td>✔ ✔ ✔</td>
<td></td>
<td>✔ ✔ ✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>South Bound Business Logic</td>
<td>✔ ✔ ✔</td>
<td></td>
<td>✔ ✔ ✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Meta-tasks, e.g., install &amp; upgrade</td>
<td>✔ ✔ ✔</td>
<td></td>
<td>✔ ✔ ✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

We haven’t done a good job defining our users and/or explicitly meeting their needs.

#ODS Summit
Example: Easy Upgrades for Users

• Upgrades are a MUST! Customers will not accept a “reinstall/reconfigure” story
• Upgrades in OpenDaylight are an afterthought
• Upgrades need to be a requirement
• No testing in OpenDaylight today!
• Some problem areas / considerations
  • Configuration & Implementation stored together
    • We have configuration options and implementations (classes, etc) defined in the same file.
  • Out-of-the-box and customer configurations stored together
    • If customer change behavior then we can no longer safely upgrade (replace) the file. Need to merge ← “expensive”
  • Upgrades not always backwards compatible with previous version

#ODSummit
Example: Minimizing risk of updates

- Currently everything needs to stay in one major monolithic release
  - Helium-SR1 vs Helium-SR2 – Interoperability / co-existence is not tested

- We find security vulnerability in OpenFlow
  - Patch OpenFlow
  - Rebuild *everything*
  - Release Helium-SR2 with the fix
  - We’ve also pulled in potentially changes to *everything* else and changed all versions

 karaf (All SR1)

Existing OpenDaylight Bundles (He-SR1)

- MD-SAL (He-SR1)
- OF (He-SR1)

MyFabric w/He-SR1

 karaf (All SR2)

Existing OpenDaylight Bundles (He-SR2)

- MD-SAL (He-SR2)
- OF (He-SR2)

MyFabric w/He-SR2
Example: Minimizing risk of updates

• We would like to be able to upgrade only what changed
  • Fix the OF vulnerability, release it, don’t have to worry about changes in *everything*

• This would allow
  • Faster delivery of fixes and features
  • Lower risk to adopting fixes and features

• Some complexity in testing, compatibility matrices, avoiding huge versions skew, etc.
Conclusions

- Treat OpenDaylight as a third-party (don’t fork it!)
- Leverage Karaf, Maven, OSGi, YANG, Config system for modularity
- We still need to do better at user focus
  - Where user is core developer, app developer, REST API developer, operator, administrator, etc.
  - Easy upgrades, more modularity, version compatibility, etc.