rSmart Rice Customizations and Maven

Leo Przybylski

January 18, 2013

Contents

Abstract

When an institution wants to implement customizations and/or fixes from another rice distribution, how do these customizations get pushed out to other projects? The best way is through maven and the institution's Maven repository. The trouble is that Rice is a modular

project where modules depend on the parent. It is important to keep the following in mind:

- Original project parents
- Do not use the org.kuali groupId

This document is about how to go about this.

1 Overview

The following is the module layout for Rice:

```
|-> rice
  \-> client-contrib
  |-> config
   \-> access
   |-> archetype
   |-> checkstyle
   |-> deploy
   |-> ide
  |-> core
   \-> api
   |-> framework
   |-> impl
   |-> web
  |-> core-service
   \-> api
   |-> framework
   |-> impl
   |-> web
  |-> db
   \-> impex
     \-> client
      \-> bootstrap
      |-> demo
     |-> master
     |-> server
```

```
    \-> bootstrap
    |-> demo
 |-> sql
|-> development-tools
|-> dist
|-> edl
 \-> framework
 |-> impl
|-> impl
|-> it
 \-> config
 |-> core
 |-> edl
 |-> impl
 |-> internal-tools
 |-> kcb
 |-> ken
 |-> kew
 |-> kim
 |-> krad
 |-> krms
 |-> ksb
 |-> location
 |-> vc
|-> ken
 \-> api
|-> kew
 \-> api
 |-> framework
 |-> impl
|-> kim
 \-> kim-api
 |-> kim-framework
 |-> kim-impl
```

```
 |-> kim-ldap
|-> kns
|-> krad
 \-> krad-app-framework
 |-> krad-web-framework
|-> krms
 \-> api
 |-> framework
 |-> impl
|-> ksb
 \-> client-impl
 |-> server-impl
|-> location
|-> sampleapp
|-> serviceregistry
|-> serviceregistry
|-> standalone
|-> web
```

Each module is a child that has a Maven parent back to the another module closer to the root parent. For example, `impl`'s parent is:

Listing 1: Impl Module Parent Definition

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0"
2       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4  ____________________________http://maven.apache.org/maven-v4_0_0.
       xsd">
5    <name>Rice Implementation</name>
6    <modelVersion>4.0.0</modelVersion>
7    <parent>
8      <groupId>org.kuali.rice</groupId>
9      <artifactId>rice</artifactId>
10     <version>2.3.0-SNAPSHOT</version>
11   </parent>
12   <artifactId>rice-impl</artifactId>
13 ...
14 ...
15 </project>
```

This creates a problem because if you create a new custom rice instance with a separate *groupId*, it may look like this:

Listing 2: Custom Impl Module Parent Definition

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                          http://maven.apache.org/maven-v4_0_0.
    xsd">
  <name>Rice Implementation</name>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.rsmart.kuali.rice</groupId>
    <artifactId>rice</artifactId>
    <version>2.3.0-SNAPSHOT</version>
  </parent>
  <artifactId>rice-impl</artifactId>
...
...
</project>
```

The problem with this is that it makes your custom impl module inherit changes from your rice parent instead of the foundation impl module. See illustration below:

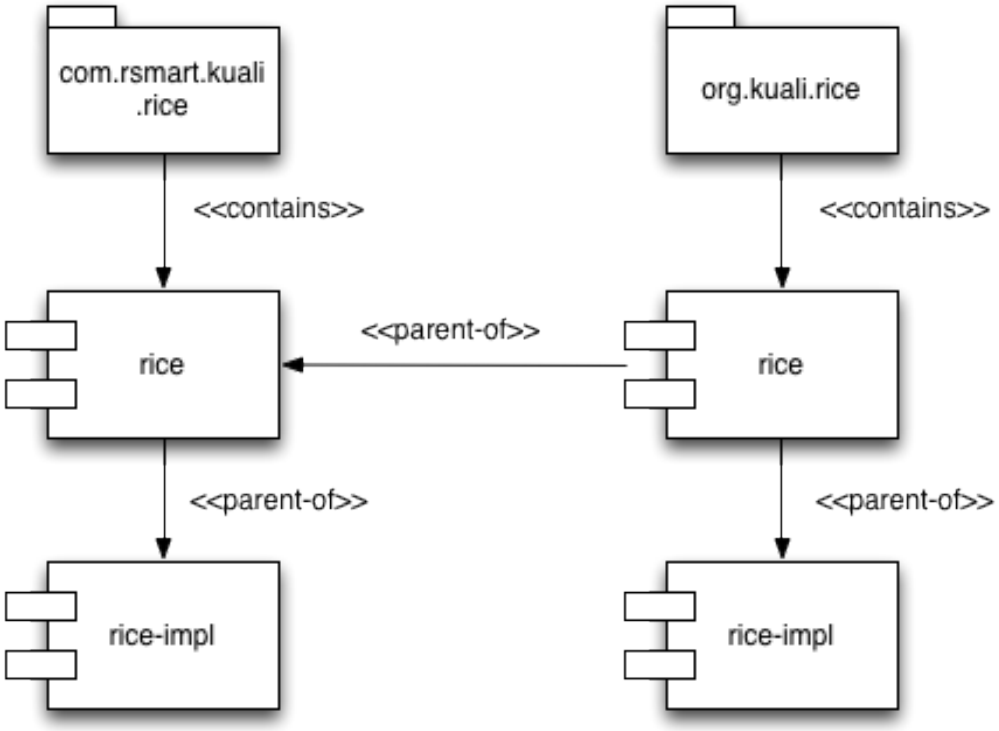

Figure 1: Parallel Hierarchy from org.kuali.rice:rice as the parent

As the figure shows, this creates a parallel hierarchy where the customized **rice-impl** regards the **org.kuali.rice:rice** parent module as its parent instead of **org.kuali.rice:rice-impl** as its parent. This means whatever changes are made to the **org.kuali.rice:rice-impl** will not be reflected.

2 Proposal

To summarize, we want to make it so each module is basically an overlay of the original rice module. We also want to maintain the behavior of the rice parent module.

The proposed solution involves a few parts, but by no means is it complicated. It is simply some effort because of the large number of modules in rice.

1. Create a project skeleton.
2. Change the module parents to point to the original rice.
3. Unpack source dependencies from *org.kuali.rice* into their respective modules.
4. Make modules behave like overlays.

1 Create project skeleton

Just like overlays, we want to assume any software in our project is going on top of the original; therefore, there shouldn't be any code in it. It is basically a rice skeleton.

2 Change the module parents to point to original Rice modules

Since **org.kuali.rice:rice-impl** already inherits from **org.kuali.rice:rice**, then **org.kuali.rice:rice-impl** should be made the parent instead.

Listing 3: Custom Impl Module Alternative Parent Definition

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0"
2       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4  ____________________________http://maven.apache.org/maven-v4_0_0.
       xsd">
5    <name>Rice Implementation</name>
6    <modelVersion>4.0.0</modelVersion>
7    <parent>
8      <groupId>org.kuali.rice</groupId>
9      <artifactId>rice-impl</artifactId>
10     <version>2.3.0-SNAPSHOT</version>
11   </parent>
12   <artifactId>rice-impl</artifactId>
13   <version>2.3.0-SNAPSHOT</version>
14 ...
15 ...
16 </project>
```

Now the **rice-impl** module gets the benefits of being a child of the original. The **com.rsmart.kuali.rice:rice** parent should just be a dummy parent for managing the version and keeping the same version for all the other modules. The biggest issue with this proposed solution is that all the modules (there are many) would need to be updated with a new parent in order for this to work properly.

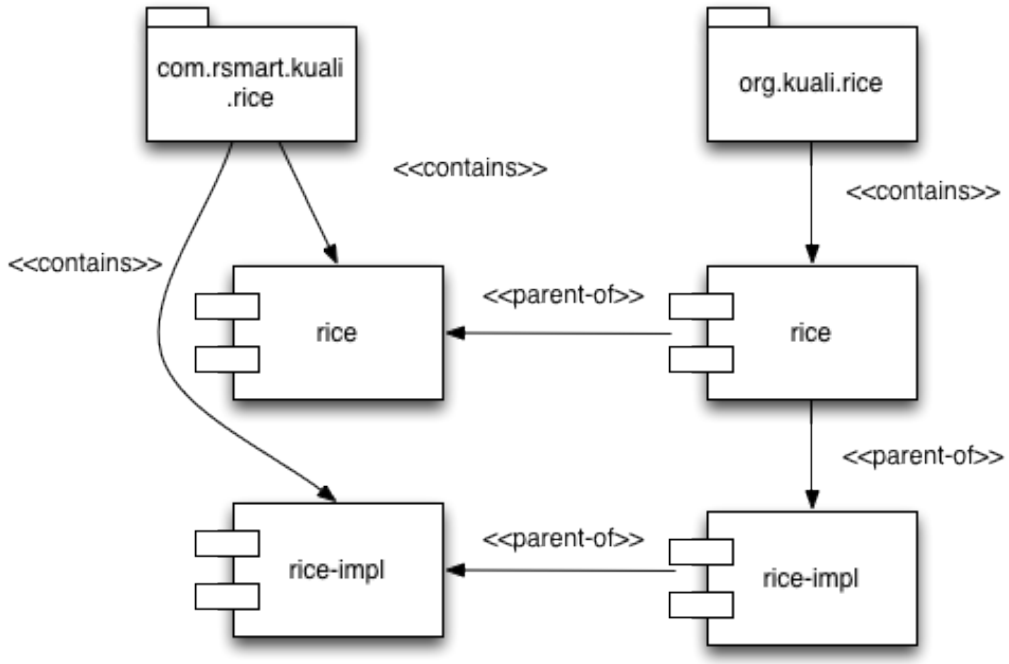

Figure 2: Hierarchy with org.kuali.rice:rice-impl as the parent to com.rsmart.kuali.rice:rice-impl

3 Unpack source dependencies from *org.kuali.rice* into their respective modules

We want our modules to behave like overlays. Unfortunately, overlays only exist for WAR projects and not for JAR. That is not to say we cannot create this scenario. A WAR overlay is just extracting the WAR file, throwing new source on top of it, and building a new WAR from that. We can do this with a normal JAR and the dependency plugin.

With each snapshot and release, the Rice project publishes to the Maven repository a *sources* jar for each module. This means that the **rice-impl** module has a sources jar. We can download and unpack the sources jar into an alternate location. Then, it can be compiled, and subsequently compiled into our project to build the overlay.

Listing 4: Impl Example for unpacking the org.kuali.rice:rice-impl source

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                          http://maven.apache.org/maven-v4_0_0.
    xsd">
  <name>Rice Implementation</name>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.kuali.rice</groupId>
    <artifactId>rice-impl</artifactId>
    <version>2.3.0-SNAPSHOT</version>
  </parent>
  <artifactId>rice-impl</artifactId>
  <version>2.3.0-SNAPSHOT</version>
...
...
<build>
...
...
  <plugins>
...
...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>${maven-dependency-plugin.version}</version>
      <executions>
...
...
        <execution>
          <id>unpack-rice-impl-sources</id>
          <phase>generate-sources</phase>
          <goals>
            <goal>unpack</goal>
          </goals>
```

```
35            <configuration>
36              <artifactItems>
37                <artifactItem>
38                  <groupId>${project.parent.groupId}</groupId>
39                  <artifactId>${project.parent.artifactId}</
                        artifactId>
40                  <version>${project.parent.version}</version>
41                  <type>jar</type>
42                  <classifier>sources</classifier>
43                  <overWrite>false</overWrite>
44                </artifactItem>
45              </artifactItems>
46              <includes>**/*.java</includes>
47              <excludes>**/*.properties,**/*.xml</excludes>
48              <outputDirectory>${project.build.outputDirectory}/
                    generated-sources/${project.parent.artifactId}</
                    outputDirectory>
49            </configuration>
50          </execution>
51          <execution>
52            <id>unpack-rice-impl-resources</id>
53            <phase>generate-resources</phase>
54            <goals>
55              <goal>unpack</goal>
56            </goals>
57            <configuration>
58              <artifactItems>
59                <artifactItem>
60                  <groupId>${project.parent.groupId}</groupId>
61                  <artifactId>${project.parent.artifactId}</
                        artifactId>
62                  <version>${project.parent.version}</version>
63                  <type>jar</type>
64                  <classifier>sources</classifier>
65                  <overWrite>false</overWrite>
66                </artifactItem>
67              </artifactItems>
68              <excludes>**/*.java</excludes>
69              <includes>**/*.properties,**/*.xml</includes>
70              <outputDirectory>${project.build.outputDirectory}/
                    generated-resources/${project.parent.artifactId}<
                    /outputDirectory>
71            </configuration>
72          </execution>
73        </executions>
74      </plugin>
75    </plugins>
76  ...
77  ...
78  </build>
79  </project>
```

The above illustrates how the dependency plugin can be used to extract the rice-impl source into a separate area for compilation at build time to produce a single jar. The above goes for all JAR projects. Rice also has a sampleapp, standalone, and web WAR project. These

will simply behave like normal overlays. That means we will no need the above code. We just add the normal overlay code.

4 Make modules behave like overlays

I think the above will do this for us, but I am leaving this section to outline our goal.

3 Use Cases

3.1 Committing modifications to the WAR

It should be maintained that in all cases, the preferred solution is not to modify the core code, but add new code that overrides its original behavior. Only modify core code when there is no other choice. This mostly happens with datadictionary and configuration of Spring sources. It can also happen when modifying inherent behavior that needs to process through a hierarchy or modification of private members.

3.2 Committing modifications to a JAR

List of Figures

Listings