

Build Profiles

Build profiles are an important concept for unifying configuration differences between different environments. They also should capture information across all aspects of the POM.

Design

General Objectives

- All profile information should be kept together to create a single view of it, rather than dispersing it throughout the POM
- Based on the design decisions below, the profile is a subset of a POM that is applied onto the current POM as a form of inheritance.
- Each profile has a globally unique identifier.

Multiple Profiles

Multiple profiles can be active simultaneously. Each profile selected is cumulative. This allows each profile to do specific things, and various combinations do not need to be created. There is no need for duplication or special inheritance of profiles.

To activate a profile, they can be added to `settings.xml`, or by the `-p profile(s)` command line option. The specification in `settings.xml` would be:

```
<activeProfiles>
  <activeProfile>jdk-1.4</activeProfile>
  <activeProfile>os-windows</activeProfile>
</activeProfiles>
```

Profiles for Dependencies

This allows some dependencies to be OS specific, JDK specific, etc.

Options:

1. Map a defined dependency to a particular implementation based on profile
2. Define new dependencies based on the profile.

Mapping would be as much work to specify and is less flexible, so the latter is the best choice.

Defining the profile of a dependency is **not** done in the dependency definition itself for the following reasons:

- previous goal to keep all profile information together
- same reason as for elimination of dependency properties - hard to manage over transitive dependencies, and awkward to specify more than one without making specification of just 1 verbose.

Proposed solution:

```

<profiles>
  <profile>
    <id>jdk-1.4</id>      <!-- Use the id of
"default" if it is the profile to use when
none is given -->
    <dependencies>
      <dependency>
        <groupId>bouncycastle</groupId>

<artifactId>bouncycastle-provider</artifactI
d>
        <version>1.1.6-jdk14</version> <!--
The JDK portion may be part of the
classifier instead (see other doc)
      </dependency>
      ...
    </dependencies>
    ...
  </profile>
  ...
</profiles>

```

This section should be added to the POM at the top level.

Profiles for Plugin Configuration

Plugin configuration and management can also be specified in the profile. Definitions here will add to the existing lists and merge as appropriate. For consistency - it behaves identically to inheritance, however regardless of the `<inherit>` and `@inheritByDefault` flag, the plugins from the current pom are always inherited into the profile as conceptually for the user, it is just overlaid, not inherited.

```
<profile>
  <id>jdk-1.4</id>
  <build>
    <plugins>
      <groupId/>
      <artifactId/>
      <version/>
      <configuration>
        ...
      </configuration>
    </plugins>
  </build>
</profile>
```

This will always bind the plugin with this version. If a version is specified in both the main plugins section and one or more profiles, the versions must match or an error occurs.

Profiles for Repositories

When switching between repositories for profiles, its possible that the snapshot and release version information is quite different.

Rather than having to maintain multiple local repositories for the segments where an alternate repository is used, just the snapshot metadata needs to be in a different location when looking to a repository from a different profile.

The resulting metadata path in the local repository is:

```
/groupId/artifactId/version/profileId/artifa  
ctId-version.version.txt
```

Another option is to write the information into one snapshot file, but that makes it different to the remote version which is undesirable and would affect timestamped checks.

The above information could possibly be duplicated in the remote repository to allow different profiles to operate on the same repository. At this point, this is not being implemented.

All elements allowed in a profile

As previously mentioned, a profile is a subset of another POM. In modello, this can be achieved by the model and profile inheriting a common base class.

The following elements will be supported by profile specifications in various locations (noted inline):

- `build` (*pom.xml*)
 - `plugins`
 - `pluginManagement`
- `dependencies` (*pom.xml*)
- `dependencyManagement` (*pom.xml*)
- `distributionManagement` (*pom.xml*)
- `repositories` (*pom.xml, profiles.xml, settings.xml*)
- `pluginRepositories` (*pom.xml, profiles.xml, settings.xml*)
- `modules` (*pom.xml*)
- `reports` (*pom.xml*)
- `configuration` (*profiles.xml, settings.xml*)

Note that `build` is included for the benefit of the plugin definitions, and that will bring along other elements such as `finalName` and `outputDirectory`. These can also be beneficial. One of the unexpected behaviours may be the ability to override `sourceDirectory` which will completely substitute a new tree. This is probably undesirable, and so may need to be forbidden by validation.

UPDATE

`build` as specified inside of a profile is not a full implementation of the traditional `build` POM element. This `build` is really another class in the model - from which the POM `build` is derived - and only allows the `plugins` and `pluginManagement` subelements when defined here. This sidesteps any issues with secondary validation after the `pom.xml` is parsed in this case.

Inheritance and Transitivity

Note that the profiles are inherited, but only in such a way as they are applied to a parent POM before the inheritance of the parent and child is performed, so the settings required are already present. This ensures the proper precedence occurs (child and its profiles win over the profile of a parent).

For this reason it is unimportant if the IDs of profiles match or not, unless they are to be specified on the command line.

Location of Profiles

The requirements listed that there would need to be all of the following:

- per user
- per project
- global

Project based settings that apply to all users can be stored in `pom.xml` as previously shown.

For user specific settings that apply to a particular project, they should use a `user-*` profile ID. These can go into the POM (and be stored in SCM), or be in `profiles.xml` in the same directory (which may not be stored in the

SCM).

`profiles.xml` are transient profiles that are for per-user customisations to an individual project. They are generally not stored in SCM, and are never deployed. Before of this, they are not inherited, except when a USD or reactor build is present that will locate them and match them to the parent POM.

For any settings that span multiple projects, the profiles need to be stored inside `settings.xml` instead of `pom.xml`.

These should take on a similar format, however is restricted to:

- repositories (which are added to the list in the POM)
- plugin repositories (As above)
- general configuration properties

General configuration properties are used instead of allowing plugin configuration to be given. This avoids a settings file circumventing a POM. For example, the following general configuration may be given:

```
<settings>
  ...
  <profiles>
    <profile>
      <id>default</id>
      <properties>

<tomcat5x.home>/usr/local/jakarta-tomcat-5.5
.9</tomcat5x.home>
      </properties>
    </profile>
  </profiles>
</settings>
```

This will be made available to the POM interpolation, so could be used as configuration in a plugin:

```
<plugin>

<artifactId>cactus-maven-plugin</artifactId>
  <configuration>
    <container>
      <id>tomcat5x</id>
      <home>${tomcat5x.home}</home>
    </container>
  </configuration>
</plugin>
```

The main drawback to this is that it requires some consistency of property naming - however this outways the problems of directly injecting plugin configuration from settings.

Preset Profiles

It should be possible to preset the activation of some profiles. Eg, JDK, OS, host and user profiles could all be made active based on the settings of the machine running the build. However, should any profiles be made active via `settings.xml`, this will override all of the existing ones, rather than add to them.

Likewise, specifying profiles on the command line will override all of those from `settings.xml` as well as any predefined ones.

This will be done by some special activation tags (the format of which is yet to be completely decided), for example:

```
<profile>
  <id>hsqldb</id>
  <activation>
    <property>hsqldb.enabled</property>
  </activation>
  <dependencies>
    <dependency>
      <groupId>hsqldb</groupId>
      .
      .
</profile>
```

```
<profile>
  <id>jdk-1.4+</id>
  <activation>
    <jdk>1.4+</jdk>
  </activation>
  <build>
    <pluginManagement>
      <plugin>

<artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.4</source>
      .
      .
</profile>
```

Resource Filtering

Under the above design, filtering can simply be a configuration element on the resources plugin (as per Kenney's patch, already available). The profiles will allow the introduction of different filter sets.

Lifecycle Binding

The plexus use case requires that a different goal be bound to the lifecycle. However, this does not entail any changes to the lifecycle design with the above solution. Instead, the goals would simply be added to the `<plugins>` section as needed in each profile.

Additional Features

These are some thoughts on additional features that may or may not be made available:

- `m2 --display-profiles` switch that will show all possible profiles from the pom, its parents, and transitive dependencies.
- features to disable transient profiles in a deployment/release scenario to ensure the build is pure.