

Toolchains

Goal

Have a way for plugins to discover what JDK (or other tools) are to be used, without configuring the plugins. The current Maven way of achieving this is to run Maven itself with the required JDK. After toolchains, the JDK that Maven is running within, shall be irrelevant to the project in question.

Motivation

Current way of enforcing project's JDK version (via the enforcer or otherwise) by forcing the user to run Maven itself with the given JDK version breaks embedded use.

Additionally toolchains will allow a type of user interaction that IDE users are used to. Just set the JDK to the project and go.

Types of toolchains

Toolchain	Relevant Plugins
jdk	maven-compiler-plugin maven-surefire-plugin maven-javadoc-plugin keytool-maven-plugin exec-maven-plugin webstart-maven-plugin
j2me sdk	j2me-maven-plugin
native tools? c#?	
netbeans-platform	nbm-maven-plugin The various goals could make use of it.

Design

Note: I'll be focusing on JDK toolchain. I don't have enough background information for other types of toolchains.

The associated issue is: [MNG-468](#)

3 basic points of view:

1. **Plugin** denotes what toolchain type it requires for its operation. So compilation, surefire, jnlp, ... need a JDK toolchain instance for example. The actual instance of the toolchain is passed into the plugin by the infrastructure (using MavenSession in current implementation). Most current plugins that use JDK for processing, use the maven's JDK instance by default. The only change introduced is to use the toolchain in the MavenSession if found. If not, do as we did so far.

Q1: how shall the plugin tell what toolchains it needs? parameter?

parameter's or mojo's @toolchain annotation? The actual retrieval of the toolchain from build context can be done completely behind the scenes, so marking the plugin as toolchain-aware is primarily documentation oriented.

2. **User** defines the toolchain instances that are available in his current setup. Shall be user based, project independent, stored in \$HOME/.m2/toolchains.xml file.

Example toolchains.xml file:

```

<toolchains>
  <toolchain>
    <type>jdk</type>
    <provides>
      <version>1.5</version>
      <vendor>sun</vendor>
      <id>for_mevenide</id>
    </provides>
    <configuration>

<jdkHome>/home/mkleint/javatools/jdk</jdkHome
e>
      </configuration>
    </toolchain>
    <toolchain>
      <type>jdk</type>
      <provides>
        <version>1.6.0</version>
      </provides>
      <configuration>

<jdkHome>/home/mkleint/javatools/jdk1.6.0</j
dkHome>
      </configuration>
    </toolchain>
  </toolchains>

```

3. **Project** shall be allowed to state which instance of the given toolchain type it requires for the build. Therefore making sure that all plugins use jdk15 for example. if such toolchain instance is not found in user's local setup, the

build shall fail early.

For this purpose a new plugin (maven-toolchain-plugin) is introduced.

It is responsible for matching the correct

toolchain instances from the user's setup against the requirements of the project and make it available for other plugins to use (in the build context).

```
<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-toolchains-plugin</artifactId>

  <version>1.0-SNAPSHOT</version>
  <executions>
    <execution>
      <phase>validate</phase>
      <goals>
        <goal>toolchain</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <toolchains>
      <jdk>
        <version>[1.4)</version>
        <!--vendor>sun</vendor-->
      </jdk>
    </toolchains>
  </configuration>
</plugin>
```

The process of matching required toolchain against the provided ones is following.

1. Get all toolchains for a given type (eg. jdk)
2. go through them one by one and try match the requirements in the toolchain-plugin configuration against the provided tokens. Some like 'version' are to be specially handled and allow for range matching etc, the rest is exact match only. The creator of the particular type of toolchain can define the specially handled tokens.
3. first successful match is pushed into the maven build environment (MavenSession) for use by other plugins down the road
4. if there are no matches, the build fails.

Backward compatibility

1. The Toolchain core components need to go to 2.0.9-SNAPSHOT and 2.1-SNAPSHOT codebase.
2. maven-toolchains-plugin gets the maven prerequisite of 2.0.9-SNAPSHOT. That way anyone actively using toolchains needs to be using 2.0.9-SNAPSHOT+
3. Any other plugins using the toolchains components don't need to explicitly set the maven prerequisite to 2.0.9-SNAPSHOT. The components will work but will fail to deliver a configured toolchain, thus they will keep behaving the same way as they did before. That's critical for further development of plugins.

What changes are needed in plugin code to start using toolchains?

The changes are rather simple. An additional dependency on the toolchains component artifact is necessary and then a short code snippet added to mojo's execute method.

```
/**
 * @component
 */
private ToolchainManager
toolchainManager;

/**
 * The current build session instance.
This is used for
 * toolchain manager API calls.
 *
 * @parameter expression="${session}"
 * @required
 * @readonly
 */
private MavenSession session;
```

```

public void execute() {
    .....

    //get toolchain from context
    Toolchain tc =
toolchainManager.getToolchainFromBuildContext( "jdk", //NOI18N
                                                    session );

    if ( tc != null )
    {
        getLog().info( "Toolchain in
javadoc-plugin: " + tc );
        //when the executable to use is
explicitly set by user in mojo's parameter,
ignore toolchains.
        if ( javadocExecutable != null)
        {
            getLog().warn( "Toolchains
are ignored, 'javadocExecutable' parameter
is set to " + javadocExecutable );
        }
        else
        {
            //assign the path to
executable from toolchains
            javadocExecutable =
tc.findTool( "javadoc" ); //NOI18N
        }
    }
}

```

```
} . . . . .
```

Implementation

The feature is developed mainly on trunk. [toolchains components](#) [toolchains plugin](#)
Only the changes to existing plugins are on [branch](#).

after compiling, make sure you put the jar with toolchain components into the 2.1-SNAPSHOT maven binary, into the lib/ subfolder.

Attached is sample project and toolchain definition.