How we build Sakai using Maven

November 8, 2005

Please direct questions or comments about this document to:
Glenn R. Golden, ggolden@umich.edu

We use the Maven project management tool from Apache (http://maven.apache.org/) to build Sakai. The Sakai source code is organized in a way that works with Maven. This organization is reflected in both our local working directories where we build and edit code, and in the Sakai source code repository.

Sakai 2 introduces a new structure for the source code, with a new and better use of Maven features, and a new Sakai Maven plugin. This document describes how the Sakai source is organized and how it is built using Maven.

Software Organization

To understand the way Sakai source code is organized, we introduce some terms:

- **SAKAI_DEV or Root**: this is the folder at the root of all your Sakai source code.

- **Module**: a module is a major unit of Sakai. SAF, the Sakai Application Framework, is a module. Each Application developed for Sakai is a separate module. Modules form the directories that live in the root.

- **Project**: A project is one part of a module; it is the set of code that produces a single "artifact" (usually a .jar or .war file). Each Module includes one or more projects. Projects make up the directories that live inside the module directory, i.e. they are two levels down from the root. A project also corresponds to one Eclipse project

All the code we work with in Sakai can be placed somewhere below the SAKAI_DEV root directory. Each application and the framework are separately packaged in a Module that has a folder in the root. Each Module has multiple project folders within. Each project folder is organized to work with Maven.

Here's an abstract example of the structure:

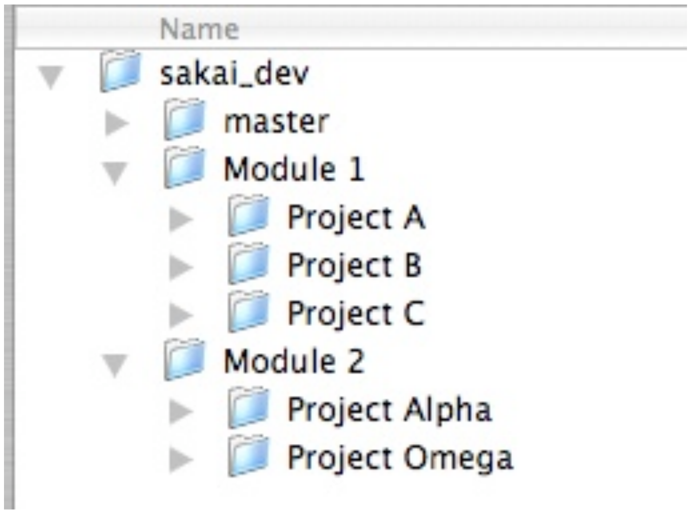

Figure 1: Example Sakai Dev Directory Structure

The master directory holds some build-wide configuration information. See the section on maven configuration later in this document.

Here's another example with some actual Sakai components: a small part of the framework and a "test" application with three projects:

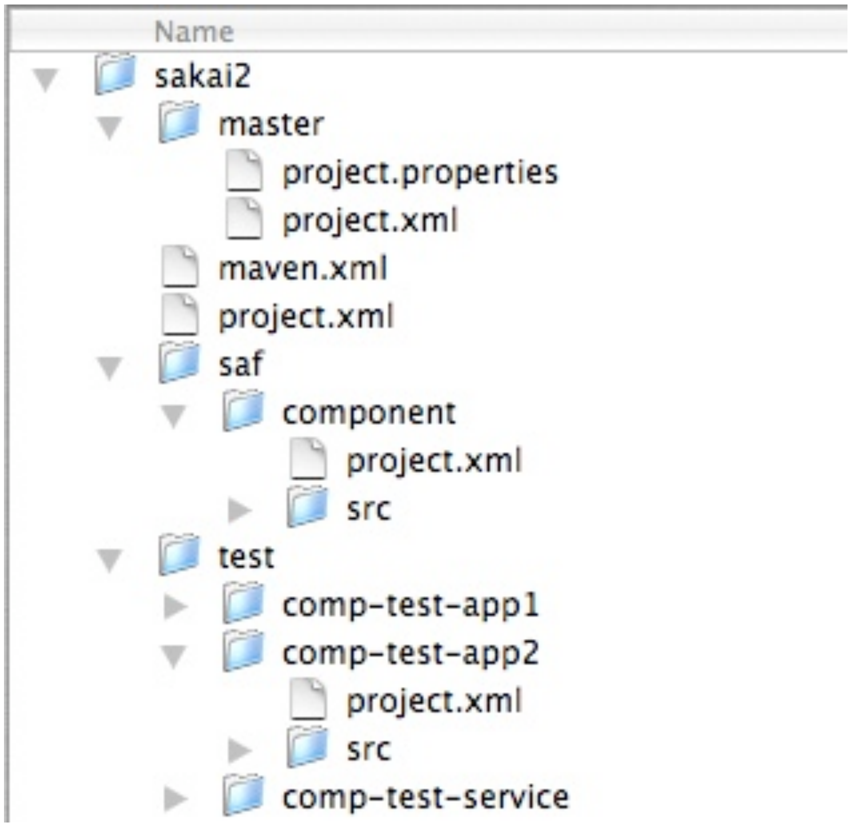

Figure 2: Actual Sakai Dev Directory Structure

Modular Sakai

Sakai has many optional features; different applications we can choose to use (or not), and different options within the applications and the framework. The Sakai integrator will collect together all the parts of Sakai that she wants for her Sakai installation, placing each module into her SAKAI_DEV directory. She may also collect various optional projects to place into various modules. Any project or module not desired can be removed from SAKAI_DEV.

Once the set of software is selected, the build and deploy process automatically adjusts to what parts were selected, and orders the process steps based on the declared dependencies of these parts.

Maven Projects and Goals

Maven works by using the files project.xml, and optionally maven.xml, located in the SAKAI_DEV root, in each module directory, in each project directory, and in the master directory. These contain all the build related instructions and project dependencies. See the Maven web site for more details about the project model and Maven goals.

Maven can be extended by plugins. We have packaged the goals we use for Sakai into a plugin. This plugin will be automatically installed as soon as you invoke a Maven goal from the SAKAI_DEV folder; you can optionally build and install this manually.

The Maven goals defined for Sakai treat all the modules and projects below as a single unit to weave together dependencies and know what to build. We use the "Maven:reactor" to do this. These goals are defined in our Sakai Maven plugin. Goals can be refined for each project or module by including special Maven code in that project or module's maven.xml file. Otherwise, the maven.xml files are not required.

By issuing Maven goals in the SAKAI_DEV directory, we build all of Sakai and deploy this with dependencies. By issuing the same Maven goals in a module, we build just the projects of the module, and deploy these and their dependencies. By issuing the same Maven goals in a project, we build just the project, and deploy it and its dependencies.

Maven's Artifacts and Repositories

Maven works by building projects that create a single file or artifact. The two most common types of artifacts we create are .jar files, containing sets of java classes, and .war files, containing the files for a web application to deploy in the container. Each project creates a single artifact.

This places a design restriction on how we organize our Sakai applications. If we have an application that produces some shared Service APIs, and components which satisfy them, we need to create two different artifacts: a .jar with the APIs, which can be installed in our shared area, and a .war for a webapp that hosts the service components. This application's module would have two projects.

When Maven builds a project, the artifact is stored into a local repository. The default location of the repository is ~/.Maven/repository/sakaiproject. The local repository is also used to download and collect externally found dependencies.

When Maven builds, it satisfies dependencies by taking them from the local repository, and stores the results of the build to the local repository.

When we “deploy”, or move our .jar and .war and other files into our servlet container (i.e. Tomcat) to be ready to package a release or run locally, we also use Maven, which collects all the needed files from the local repository.

Sakai Maven Goals

Maven is controlled with goals. To enter a goal, enter the directory in which you want to work, usually the SAKAI_DEV, and issue the command:

```
> maven <goal or list of goals>
```

The following are the official and namespace-protected goals are defined for Sakai in our plugin. They may be used at the root, module or project level:

- sakai:clean - remove any prior build’s byproducts

- sakai:build - compile and package Sakai, installing all artifacts into the local repository

- sakai:deploy - install the needed files to the local Servlet container

- sakai:undeploy - remove the installed files from the local Servlet container

- sakai:clean_build - clean then build

- sakai:clean_build_deploy - clean then build then deploy

- sakai - clean then build then deploy

- sakai:deploy-report – report on the jars deployed by Sakai

- sakai:javadoc – Create the javadoc documentation at target/sakai-javadoc.zip

- sakai:taglibdoc – Create the JSF tag library documentation at target/sakai-taglibdoc.zip

You can ask Maven for a description of the goals for the Sakai plugin by issuing the command “maven –P sakai”. You can have Maven describe all goals in all available plugins by issuing the “maven –g” command.

To avoid so much typing, we define some 3 letter goals for use with Sakai. These are not protected by the “sakai:” namespace, so they live with all the other Maven internal and plugin goals. It’s possible we might someday run into a conflict with these:

- cln – alias for sakai:clean

- bld – alis for sakai:build

- dpl – alias for sakai:deploy

- und – alias for sakai:undeploy

Multiple goals can be given to Maven to run in sequence. For instance, to clean, build and deploy, you can say:

```
> maven cln bld dpl
```

You can also use the “sakai” goal, which does the same thing:

```
> maven sakai
```

In the root folder, we provide a maven.xml file to define the default goal to run if no goal is specified on the command line to Maven. For convenience, the “sakai:build” goal is selected as the default. You can do the same for projects and modules you work with often.

Deploy

The deploy and undeploy goals need careful setup in the project.xml files. There are two places we need to declare additional project information: at the root of the project, where we declare what sort of artifact this project creates and where it needs to be deployed; and with each dependency, where we declare if and where the dependent .jar file should be deployed.

The project artifact type and destination are declared at the top level of the project.xml as properties:

```
<project>
      <pomVersion>3</pomVersion>
      <extend>../master/project.xml</extend>
      <name>Sakai Component Manager</name>
      <groupId>sakaiproject</groupId>
      <id>sakai2-component</id>
      <currentVersion>2.0.0</currentVersion>

      <properties>
            <!-- deploy as a jar -->
            <deploy.type>jar</deploy.type>
            <!-- deploy to "shared", "common" or "server" -->
            <deploy.target>shared</deploy.target>
      </properties>
```

The property “deploy.type” should be set to “jar” to build and deploy the project’s artifact as a .jar file, or “war” to build and deploy the project’s artifact as a .war file. Note: the sakai:build goal depends on this setting as well, and if it is not specified, the project must have a maven.xml which overrides the default sakai:sakai.build goal.

“jar” deploy.type artifacts should also specify where the .jar should be deployed. If the jar is intended to be part of any .war file that needs it, don’t specify any deploy.target for this project; this will keep it from being deployed. Otherwise, it’s probably going into the shared/lib, so set the “deploy.target” property to “shared”. In special cases, we need to put the artifact into common/lib or server/lib. Use the values “common” or “server” for these.

“war” deploy.type artifacts are deployed into webapps.

The second place we specify a deploy.target is in the dependencies of a project. Any dependency that we need to have available at runtime we need to deploy in some way. Note that some dependencies (like the Servlet API) are assumed to be already present in the Servlet container and need not be deployed by us.

There are two options for dependencies. You might want the dependent .jar in the webapp, for “war” deploy.type artifacts. Maven has a mechanism to indicate that, the “war.bundle” property:

```
            <dependency>
                  <groupId>velocity</groupId>
                  <artifactId>velocity</artifactId>
                  <version>1.3.1</version>
                  <properties>
                        <war.bundle>true</war.bundle>
                  </properties>
            </dependency>
```

Set this to “true” to cause the dependent jar to be included in the .war artifact.

The second option is to deploy the dependency to the shared/lib, common/lib or server/lib. Use our “deploy.target” property to indicate this:

```
            <dependency>
                  <groupId>springframework</groupId>
                  <artifactId>spring</artifactId>
                  <version>1.1.4</version>
                  <properties>
                        <!-- deploy dependency … -->
                        <deploy.target>shared</deploy.target>
                  </properties>
            </dependency>
```

There are two forms of redundancies that can enter our project.xml files. First, each of our artifacts will have a declaration of where to deploy. Our artifacts will also be dependent on each other, so they may also get instructed to be deployed in a dependency declaration.

The second form of redundancy is that different modules might have the same dependencies, so those dependencies will be declared for deployment multiple times when the entire system is deployed.

The deploy and undeploy process can deal with these redundancies, as long as they are consistent. Make sure that if an artifact is declared to be deployed to shared/lib, that any dependency declaration also has it going to shared/lib and not somewhere else. Deploy will only deploy the same file to the same location once in a deploy execution, no matter how many times it is referenced.

By declaring all the deployment needs and by being redundant, we better support the running of the sakai:deploy goal from the root or from modules or projects. We could also decide to only declare deployment needs for external dependencies and at the artifact level, which is a bit easier to maintain.

Some external dependencies have additional dependencies that our code does not directly depend on to compile, but need to be present at runtime. We must declare these as well as normal dependencies that get deployed. This is a case where we might benefit from declaring the full set of dependencies at one strategic place so we don't have to replicate it everywhere.

For example, the Spring framework has a list of other .jar files it needs at runtime. We can declare the deploy dependencies for Spring and its needs only in the framework module's component project, which is where Spring is most used.

Project's optional maven.xml

Since Sakai defines all goals in the sakai plugin, the maven.xml files at the root, module and project level are optional. They can be defined to:

- define a default goal for Maven

- override and extend clean, build, deploy or undeploy instructions

To define a default goal for Maven, your maven.xml would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<project default="sakai:clean_build" />
```

Set whatever goal you wish for the default.

To override or extend the build instructions, the maven.xml would also define some goals. These are:

- sakai:sakai.clean

- sakai:sakai.build

Here's how they look in the plugin:

```
        <goal name="sakai:sakai.clean">
              <attainGoal name="clean:clean" />
        </goal>

        <goal name="sakai:sakai.build">
              <j:if test="${pom.getProperty('deploy.type')=='jar'}">
                    <attainGoal name="jar:install" />
              </j:if>
              <j:if test="${pom.getProperty('deploy.type')=='war'}">
                    <attainGoal name="war:install" />
              </j:if>
        </goal>
```

If there is more to do, start with these and add additional Maven Jelly commands to the goals.

Deploy and undeploy are much more complex Maven code, and cannot easily be overridden at the module or project level.

The Reactor

The Sakai Maven goals use the reactor to invoke the corresponding internal goal in all the sub modules and their sub projects:

```
              <goal name="sakai:build">
              <Maven:reactor
                    includes="**/project.xml"
                    basedir="${basedir}"
                    goals="sakai:sakai.build"
                    banner="building:"
                    ignoreFailures="true"
                    />
        </goal>
```

The reactor examines all the projects it finds, and computes a build order based on the dependencies declared in each project's project.xml. It then builds the desired goal for each in order:

```
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0.2
```

```
Starting the reactor...
Our processing order:
Sakai
Sakai Component Manager
Sakai Component Manager Test Service
Sakai Component Manager Test 1
Sakai Component Manager Test 2
+----------------------------------------
| building: Sakai
| Memory: 2M/3M
+----------------------------------------
+----------------------------------------
| building: Sakai Component Manager
| Memory: 2M/3M
+----------------------------------------
…
```

Maven configuration: build.properties and project.properties

Some options in Maven will likely need to be adjusted for each person building Sakai. These are controlled from the configuration file build.properties. This should be located in your user's home directory (i.e. ~/build.properties).

There are two important values that must be set in our build.properties. One specifies the location of the remote repositories that Maven will search to download project dependencies. The other identifies where our Tomcat Servlet container is for deploys.

Here's an example of build.properties:

```
maven.repo.remote =
http://cvs.sakaiproject.org/maven,http://www.ibiblio.org/maven
maven.tomcat.home = /usr/local/tomcat/
```

Notice that we identify the repository at sakaiproject.org, where some special dependencies are kept, and the one at ibiblio, where you can find just about anything in the open-source world for download.

Some options for a Sakai build will vary by the particular release of Sakai. Sakai has a high level 'master' directory that contains a project wide project.xml and project.properties file. The master project.properties file provides specific build-wide values of sakai.version (the current version of Sakai) and sakai.plugin.version (the required version of the Sakai maven plugin). By default sakai.version is set to TRUNK. This indicates that the code is based off the head of the current trunk code and should be considered unstable. When a stable branch is created the value of sakai.version in the master project.properties in the branch is changed to an appropriate value. If you have a dependency on a Sakai artifact the version name should be specified in the project.xml file as `<version>${sakai.version}</version>` to ensure consistency.

The project.xml files for Sakai code should extend this master project.xml to ensure consistency during builds. The project.xml file shown in the Deploy section above gives an example of how to extend a project.xml file. The path required will vary depending on the location of the file in the source tree.

Installing the Plugin

You can install the Sakai plugin into your maven environment. This makes the plugin available whenever you use maven, not just for those projects that declare a dependency on the plugin. This is useful to let you run maven commands from the modules and projects within Sakai, instead of always building the entire code base.

The latest version of the plugin is kept current in the sakaiproject maven repository. To install this into your local maven environment, use this maven command:

```
maven plugin:download -DgroupId=sakaiproject
-DartifactId=sakai -Dversion=2.0
```

Update the version to the version of Sakai you are working with, 2.0 or later.