Maven

This is meant to be a brief overview of the concepts necessary in understand how to use Apache Maven.

For a more complete reference, see: http://books.sonatype.com/mvnref-book/reference/

Prerequisites

- Docker
- Maven (if not using Docker)

Basic Concepts

For the below examples, a Maven **<command>** will typically be of the form **mvn clean package install** where mvn is the command being run, and clean /package/install/etc are **phases** of the Maven project you're building.

Maven includes several predefined phases to cover the default lifecycle:

- clean - clears out old build artifacts
- test - run the Maven surefire plugin to execute your tests and display the result (i.e. JUnit, TestNG, etc)
- package - download dependencies (if necessary) and build up new artifacts into the **target/** folder
- install - installs built artifacts (assumes **package**) from your **target/** folder into your local .m2 Maven repository, allowing local Maven projects to consume the built artifacts
- install-file - install a JAR file manually to your .m2 repo, allowing local Maven project to consume the JAR
- deploy - publish built artifacts (assumes package) to a remote Maven repository, allowing local Maven projects to consume the built artifacts
- deploy-file - publish a standalone JAR file manually to a remote Maven repository, allowing other remote Maven projects to consume the JAR

For a full list of these lifecycle goals: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference

How it Works

Every Maven project contains a file called **pom.xml** at the project root. This file is the POM or Project Object Model, which tells Maven everything it needs to know about the project's dependencies and build steps.

Running a Maven command will parse the current project's POM to determine available phases/goals, and execute them according to the details defined within.

The pom.xml can also link off to various other XML documents describing various pluggable aspects of the build process:

- build.xml: if you're using the Maven Antrun Plugin, this file tells Ant how to execute
- assembly.xml: If you're using the Maven Assembly Plugin, this file tells Maven how to package your application (JAR, WAR, ZIP, etc)

Nested Projects

POMs can also exist in subfolders of the root. Every subproject's POM inherits the values defined by the parent, creating a nested hierarchy.

Values in subprojects will override values in the parent project, so we must first compute an "effective POM" by walking down the POM tree from root project to our target subproject and collect all values encountered.

Building Artifacts (Basic)

Now that you've heard the basics, let's try it out!

If you're lucky, you'll only ever need to know a single command to use Maven...

Via Maven

The following command will run Maven's **clean**, **package**, and **install** phases natively:

```
$ mvn clean package install
```

This is a very common command to build a Maven project from source, and install its built artifacts into your .m2 cache for local consumption.

Via Docker

The following command will run the same phases within a Docker container, without needing to install Maven on your machine:

```
$ docker run -it --rm -v $(pwd):/build -w /build maven:3-jdk-8 mvn clean package install
```

Installing Artifacts (Intermediate)

Sometimes, your dependency might not be hosted publicly or may be unavailable for download.

If this is the case, your build will almost certainly fail with "Unable to locate artifact" errors if you don't already have the artifacts in your local cache.

You can manually install such missing dependencies using the following command:

```
$ mvn install:install-file -Dfile=./indri-5.11.jar -DgroupId=edu.illinois.lis -DartifactId=indri -Dversion=5.11
-Dpackaging=jar
```

For example, the above command will install i**ndri-5.11.jar** into your local Maven cache, allowing your <dependencies> section to be able to resolve it.

Deploying Artifacts (Advanced)

Most people will have stopped reading by now, but if you're part of the unlucky few, then you have my condolences.

In order to deploy artifacts to a remote repository, Maven recommends syncing them to The Central Repository (so that other projects can easily consume their artifacts for reuse).

In general, the requirements are as follows:

1. Final **pom.xml** must include:
    a. Project coordinates: groupId / artifactId / version
    b. Project name and description
    c. License information
    d. Developer information
    e. SCM information
2. Final **pom.xml** must **NOT** include:
    a. <repositories> tags
    b. <pluginRepositories> tags
3. Final **bundle.jar** must include:
    a. Project POM (i.e. ir-utils-0.1.0.pom)
    b. Compiled JARs (i.e. ir-utils-0.1.0.jar)
    c. JavaDoc JAR alongside each compiled JAR (i.e. ir-utils-0.1.0-javadoc.jar)
    d. Sources JAR alongside each compiled JAR (i.e. ir-utils-0.1.0-sources.jar)
    e. GPG signatures for all of the above (i.e. ir-utils-0.1.0.jar.asc, ir-utils-0.1.0-javadoc.jar.asc, etc)

Choose Project Coordinates

The first step is to confirm your project coordinates (groupId:artifactId:version).

See http://central.sonatype.org/pages/choosing-your-coordinates.html

First and foremost, your Maven groupId should be the reverse of a FQDN over which you have some management rights. For example, ***edu.illinois.lis*** or ***org.nationaldataservice*** or ***org.ndslabs.**** might be acceptable groupIds for the biocaddie code base.

Ideally, your Maven groupId and Java packages should match or at the very least be fairly similar.

There are several options for doing so, including hosting your own staging repository.

Choose Hosting Solution

You will then need to secure access to a staging server. I used OSSRH for my artifacts, but several other options exist

Self-Hosted via Docker (Development / Internal Use Only)

WARNING: I am ***assuming*** that this would run an interface similar to https://oss.sonatype.org/, but I have not tested it myself
OSSRH was easier configuration and more long-term than self-hosting, so try this at your own risk... you have been warned!

To run your own test Nexus repository via the official Docker image:

```
$ docker run -d -p 8081:8081 --name nexus sonatype/nexus:oss
```

You should then be able to follow the steps below (untested) using the IP/hostname of this machine and port 8081.

You can test it with the following command:

```
$ curl -u admin:admin123 http://localhost:8081/service/metrics/ping
```

NOTE: Don't forget to change the default password

Open-Source Software Repository Hosting (OSSRH)

See http://central.sonatype.org/pages/ossrh-guide.html

See http://central.sonatype.org/pages/apache-maven.html

For a more long-term solution for hosting your (open-source) Maven artifacts, you can follow these steps to deploy them to OSSRH.

If you don't already have one, create an account on the Sonatype JIRA

Next, create a **New Project** ticket describing the artifacts that you plan to deploy to OSSRH.

The Sonatype team will reach out to confirm that you "own" the domain used for the groupId (see above), and can answer any questions you might have. The turnaround time was really quick for my ticket, and they will respond in the comments with next steps or problems (if any arise).

Once approved, you should be able to follow the steps below to upload your artifacts, as well as access the staging server using your JIRA credentials to perform releases.

Other Options

See https://maven.apache.org/guides/mini/guide-central-repository-upload.html (Section: Approved Repository Hosting)

Approved Repository Hosting

Instead of maintaining repository rsync feeds for each projects, we now encourage projects to use an approved repository hosting location.

Currently approved repository hosting locations:

- Apache Software Foundation (for all Apache projects)
- FuseSource Forge (focused on FUSE related projects)
- Nuiton.org

Automatic publication will be provided for Forges that provide hosting services for OSS projects and other large project repositories that meet certain minimum criteria such as validation of PGP keys and pom contents as defined above. If you are interested in becoming an approved Forge, contact us at repo-maintainers@maven.apache.org.

Add Credentials to settings.xml

NOTE: Below I use OSSRH as an example, but this should work for any repository.

Plug your Nexus credentials into the <servers> block of your **settings.xml** file.

On OS X, this was located at */usr/local/Cellar/maven/3.5.0/libexec/conf/settings.xml*

For OSSRH, this is your JIRA username/password from the account that you created above.

For the Docker version, the default credentials are **admin:admin123** - please change this before using.

For example:

```
<servers>
  <!-- other <server> blocks -->

  <server>
    <id>ossrh</id>
    <username>YOUR_USERNAME</username>
    <password>YOUR_PASSWORD</password>
  </server>

  <!-- other <server> blocks -->
</servers>
```

Deploying a Single Artifact

NOTE: Below I use OSSRH as an example, but this should work for any repository.

You can use a syntax similar to that of **install-file** to deploy a single artifact to a remote Nexus repo:

```
# Deploying a SNAPSHOT (non-release) artifact
$ mvn deploy:deploy-file -Dfile=./indri-5.11.jar -DgroupId=edu.illinois.lis -DartifactId=indri -Dversion=5.11-
SNAPSHOT -Dpackaging=jar -Durl=https://oss.sonatype.org/content/repositories/snapshots/ -DrepositoryId=ossrh


# Deploying a non-SNAPSHOT artifact to the staging server
$ mvn deploy:deploy-file -Dfile=./indri-5.11.jar -DgroupId=edu.illinois.lis -DartifactId=indri -Dversion=5.11 -
Dpackaging=jar -Durl=https://oss.sonatype.org/service/local/staging/deploy/maven2/ -DrepositoryId=ossrh
```

Gotchas:

- SNAPSHOTs will overwrite each other... I still haven't figured out how to deploy a full set of SNAPSHOT artifacts. Maybe manually bundling them up before calling **deploy-file**?
- Deployment of SNAPSHOT artifacts to a staging server will result in failure (**HTTP 400: Bad Request**). Your artifacts must not end with SNAPSHOT to successfully deploy to staging.
- Similarly, deploying non-SNAPSHOT artifacts to a snapshot server will also fail with **HTTP 400: Bad Request**.

Configuring Upstream POMs for Deployment

NOTE: Below I use OSSRH as an example, but this should work for any repository.

Once your project ticket has been approved, you can add the following block to your project's **pom.xml** file:

```
        <distributionManagement>
          <!-- When your version ends with "-SNAPSHOT", it will be deployed here -->
          <snapshotRepository>
            <id>ossrh</id>
            <url>https://oss.sonatype.org/content/repositories/snapshots</url>
          </snapshotRepository>


          <!-- When your version does not end with "-SNAPSHOT", it will be deployed here -->
          <repository>
            <id>ossrh</id>
            <url>https://oss.sonatype.org/service/local/staging/deploy/maven2/</url>
          </repository>
        </distributionManagement>
```

NOTE: The <id> here should match the <servers> entry in your **settings.xml**.

You should now be able to deploy to the above repositories by running the following command:

```
$ mvn deploy
```

Configuring Downstream POMs for Consumption

NOTE: Below I use OSSRH as an example, but this should work for any repository.

Now that you've deployed your first SNAPSHOT, you should be able to download the SNAPSHOT artifacts for consumption during development/testing.

Add the following section to your root POM to automatically inherit the necessary configuration from OSSRH:

```
<project>
    <!-- the rest of your POM -->


    <parent>
        <groupId>org.sonatype.oss</groupId>
        <artifactId>oss-parent</artifactId>
        <version>7</version>
    </parent>


    <!-- the rest of your POM -->
</project>
```

NOTE: This <parent> can also be another module whose <parent> is OSSRH, and this should still work

You should now be able to pull SNAPSHOT artifacts for OSSRH projects when building using the usual command:

```
$ mvn clean package
```

If your build completed successfully, then congratulations! You are now able to access your development / pre-release SNAPSHOT artifacts!

For some projects, this will be far enough to sustain development and testing.

Central Sync Requirements (Expert)

NOTE: Below I use OSSRH as an example, but this should work for any Accepted Hosting Repository.

OSSRH (and probably other approved repository hosts) will even sync your released artifacts with Maven Central on your behalf.

Sadly, this process did not seem to lift any of the strict requirements for the artifacts synced/

See http://central.sonatype.org/pages/requirements.html

See https://maven.apache.org/guides/mini/guide-central-repository-upload.html•

If you used OSSRH, you will then need to comment back on your Project JIRA ticket to enable sync to The Central Repository after promoting your first release for them

After that, any future releases that you promote from the staging repository will be automatically synced to Central (within a few hours).

Generating JavaDoc via Maven

Add the following to your pom.xml to generate a ***-javadoc.jar** alongside each JAR that is output during the **package** phase of your build:

```
        <build>
          <plugins>
            <!-- More plugins... -->

                        <!-- Maven JavaDoc Plugin -->
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-javadoc-plugin</artifactId>
                                <version>2.10.4</version>
                                <configuration>
                                        <show>private</show>
                                        <nohelp>true</nohelp>
                                </configuration>
                                <executions>
                                        <execution>
                                                <id>attach-javadocs</id>
                                                <phase>package</phase>
                                                <goals>
                                                        <goal>jar</goal>
                                                </goals>
                                        </execution>
                                </executions>
                        </plugin>

            <!-- More plugins... -->

          </plugins>
        </build>
```

Generating Sources via Maven

Add the following to your pom.xml to generate a ***-sources.jar** alongside each JAR that is output during the **package** phase of your build:

```
	<build>
	  <plugins>
	    <!-- More plugins... -->


	                <!-- Maven Sources Plugin -->
	                <plugin>
	                        <groupId>org.apache.maven.plugins</groupId>
	                        <artifactId>maven-source-plugin</artifactId>
	                        <version>3.0.1</version>
	                        <executions>
	                                <execution>
	                                        <id>attach-sources</id>
	                                        <phase>package</phase>
	                                        <goals>
	                                                <goal>jar</goal>
	                                        </goals>
	                                </execution>
	                        </executions>
	                </plugin>
	    <!-- More plugins... -->

	  </plugins>
	</build>
```

Signing Your Artifacts via Maven

See http://central.sonatype.org/pages/working-with-pgp-signatures.html

This will ensure that consumers of your artifacts can verify the authenticity of those that they download (similar to MD5 checksum).

Add the following to your pom.xml to generate a ***.asc** alongside each output file during the **verify** phase of your build:

```
	<build>
	  <plugins>
	    <!-- More plugins... -->


	    <!-- Maven GPG Plugin -->
	                <plugin>
	                        <groupId>org.apache.maven.plugins</groupId>
	                        <artifactId>maven-gpg-plugin</artifactId>
	                        <version>1.6</version>
	                        <executions>
	                                <execution>
	                                        <id>sign-artifacts</id>
	                                        <phase>verify</phase>
	                                        <goals>
	                                                <goal>sign</goal>
	                                        </goals>
	                                </execution>
	                        </executions>
	                </plugin>


	    <!-- More plugins... -->


	  </plugins>
	</build>
```

NOTE: You will now be asked for your passphrase when running "mvn deploy"

NOTE: If your build fails with "gpg: signing failed: Inappropriate ioctl for device", run the following before the retrying the mvn command:

```
export GPG_TTY=$(tty)
```

Generate and Upload a Keypair

From Maven's own uploading guide:

PGP Signature
To improve the quality of the Central Maven repository, we require you to provide PGP signatures for all your artifacts (all files except checksums), and distribute your public key to a key server like http://pgp.mit.edu. Read Working with PGP Signatures for more information.

If you don't already have one, you will need to generate a keypair with which you can sign your artifacts.

You will need the **gnupg** command line tool (on OSX, this can be installed via **brew**):

```
$ gpg --gen-key
```

Once you have generated a keypair, you will need to upload your public key to a public keyserver.

To list existing keys:

```
$ gpg --list-keys
/Users/<USERNAME>/.gnupg/pubring.kbx
---------------------------------
pub   rsa2048 2017-05-05 [SC] [expires: 2019-05-05]
      <YOUR_KEY_ID_WHICH_WILL_BE_FAIRLY_LONG>
uid           [ultimate] Craig Willis (https://github.com/craig-willis) <willis8@illinois.edu>
uid           [ultimate] Garrick Sherman (https://github.com/gtsherman) <gsherma2@illinois.edu>
uid           [ultimate] Mike Lambert (https://github.com/bodom0015) <lambert8@illinois.edu>
sub   rsa2048 2017-05-05 [E] [expires: 2019-05-05]
```

Then execute the following command to upload your public key:

```
$ gpg --keyserver hkp://pool.sks-keyservers.net --send-keys <YOUR_KEY_ID_WHICH_WILL_BE_FAIRLY_LONG>
```

where ***<YOUR_KEY_ID_WHICH_WILL_BE_FAIRLY_LONG>*** is pulled from the output above

NOTE: You may need to share this public key with other keyservers, if your artifacts fail the Central Sync Checks for GPG signing due to a missing key

Manually Signing a Single File

To sign a file with your GPG key:

```
$ gpg -ab ir-utils-0.1.0.jar
```

NOTE: You will be asked to enter your GPG key's passphrase

This will output a **.asc** file that should be included alongside the file that produced the signature.

As a consumer, you can then verify the authenticity of the Maven artifacts by using the gpg command line tool:

```
$ gpg --verify target/ir-utils-0.1.0.jar.asc
gpg: assuming signed data in 'target/ir-utils-0.1.0.jar'
gpg: Signature made Thu Jun  1 21:21:16 2017 CDT
gpg:                using RSA key <YOUR_KEY_ID_WHICH_WILL_BE_FAIRLY_LONG>
gpg: Good signature from "Craig Willis (https://github.com/craig-willis) <willis8@illinois.edu>" [ultimate]
gpg:                 aka "Garrick Sherman (https://github.com/gtsherman) <gsherma2@illinois.edu>" [ultimate]
gpg:                 aka "Mike Lambert (https://github.com/bodom0015) <lambert8@illinois.edu>" [ultimate]
```

This will output the signature data from the **.asc** file, which should give some confidence that the artifacts were not modified or replaced by a malicious party.

Bringing It All Together

If you followed all of the above steps, then running the following command should build up all of the necessary artifacts and deploy them to the appropriate repository:

```
$ mvn clean deploy
```

This single command will:

- Clear out your existing build artifacts
- Compile and package up your code into a JAR
- Generate JavaDoc from your project and package it as a JAR
- Package up your source code as a JAR
- Sign the JAR and POM files with your GPG key
- Install the resulting artifacts into your local Maven cache
- Deploy the resulting artifacts into the remote Nexus repository

Staging a New Release (Expert)

See https://books.sonatype.com/nexus-book/reference/staging-repositories.html

At some point, you will likely want to assemble a release (non-SNAPSHOT) version of the code that people can be confident will not change out from under them suddenly. This is where releases come in - since they are unique and permanent, users can consume released artifacts confidently without fear of them changing in the future. without warning.

Automatically

There is a way to automate this process using the Maven Release Plugin (see below), but I did not fully explore the capabilities of this plugin, as the steps were easy enough to perform manually once I figured out the process.

Manually

Without the Maven Release Plugin, it is still fairly easy to upload your artifacts to the staging server using the steps below.

Bundle Your Artifacts

First, make sure your version string does not end with **-SNAPSHOT**, then perform a full build of your project using the big command from above and the *cd* to where the build artifacts were output:

```
$ mvn clean package verify install:install deploy:deploy
$ cd target/
```

NOTE: For complex project with multiple subprojects, it may be best to look into the Maven Bundle Plugin.

Bundle your build artifacts (JAR / POM / ASC files, see below for how instructions to generate these) into a single **bundle.jar** to upload them to OSSRH staging:

```
$ jar cvf bundle.jar *.pom *.jar *.asc
```

Upload Your Artifact Bundle

Then log into https://oss.sonatype.org/ with your JIRA username / password and choose "Staging Upload" on the left side.

Choose "Artifact Bundle" from the dropdown, and select your bundle.jar for upload.

After uploading, select "Staging Repositories" on the left side, and locate your bundle in the list.

To abort the process, you can choose "Drop" to delete your staged bundle.

Select the repository and choose "Close" at the top. This will run a set of checks to verify that the artifacts are of sufficient quality to upload to Maven Central.

You can see the list of checks by selecting the staging repository and viewing the "close" item of the "Activity" tab, and are examined in more details above (See "Central Sync Requirements")

Problems Closing?

- If you artifact(s) fail the GPG signing stage, you may need to upload them to an accepted keyserver (it should list the keyservers that its checking, just pick any one and give it some time to propagate the change before trying again):

```
$ gpg --keyserver <KEYSERVER_URL> --send-keys <YOUR_KEY_ID_WHICH_WILL_BE_FAIRLY_LONG>
```

- If your artifacts fail the sources or javadoc checks, make sure you have generated them as part of your build (ensure they each have an <executions> tag in your pom.xml). If they are third-party artifacts that are missing source or javadoc, see below.

If you repository "closed" successfully, then you are now ready to begin testing the staged artifacts!

Testing a Closed Repository

On the Nexus UI, click "Repositories" on the left side and from the small dropdown in the top-center of the screen, choose "Nexus Managed Repositories"

This will update the list to be quite lengthy. Locate your repository (which should be named after your project's **groupId**).

Closed repositories should be listed here with a URL in the right-most column, and will likely contain the name of the repository (i.e. **eduillinoislis-1007**).

Copy this url to a <repositories> section of your downstream POM (remember to remove this section after you finish testing):

```
<repositories>
  <repository>
    <releases>
      <enabled>true</enabled>
    </releases>
    <id>ossrh1</id> <!-- This does not need to match the <id> in your settings.xml -->
    <name>OSSRH Staging for indri-5.11 release</name>
    <url>https://oss.sonatype.org/content/repositories/eduillinoislis-1007/</url>
    <layout>default</layout>
  </repository>
</repositories>
```

You should then be able to build your downstream project using the staged artifacts you specified here! Running "mvn clean package", you should see something similar to the following:

```
Michaels-MacBook-Pro-2:ir-tools lambert8$ mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building IR Utils 0.1.0
[INFO] ------------------------------------------------------------------------
Downloading: https://oss.sonatype.org/content/repositories/eduillinoislis-1007/edu/illinois/lis/indri/5.11
/indri-5.11.pom
Downloaded: https://oss.sonatype.org/content/repositories/eduillinoislis-1007/edu/illinois/lis/indri/5.11/indri-
5.11.pom (2.1 kB at 1.5 kB/s)
Downloading: https://oss.sonatype.org/content/repositories/eduillinoislis-1007/edu/illinois/lis/indri/5.11
/indri-5.11.jar
Downloaded: https://oss.sonatype.org/content/repositories/eduillinoislis-1007/edu/illinois/lis/indri/5.11/indri-
5.11.jar (260 kB at 789 kB/s)
                   ...


[INFO] --- maven-dependency-plugin:2.8:copy-dependencies (default) @ ir-utils ---
                   ...
[INFO] Copying indri-5.11.jar to /Users/lambert8/workspace/ir-tools/target/lib/indri-5.11.jar
                   ...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 21.890 s
[INFO] Finished at: 2017-06-03T16:24:25-05:00
[INFO] Final Memory: 59M/746M
[INFO] ------------------------------------------------------------------------
```

And just like that you've built your downstream project against your staged upstream artifacts! These downstream artifacts can now be used to verify the correctness of your upstream artifacts before releasing them to the public! Typically, your Maven build would also run unit and/or integration tests against the code as well to verify that everything is working properly.

Release the Bundle

Now that your repository is closed and you've tested your artifacts *(and you did test your artifacts, didn't you?)*, you have the option to **Release**, **Promote**, or **Drop** the repository

Release

Choosing **Release** will finalize the artifacts and release them for public consumption on OSSRH. Choose this when the artifacts are ready to be deployed to concrete version.

NOTE: Make sure your artifacts are ready, as once they are released they are **extremely difficult to remove, if not impossible**.

Drop

Choosing **Drop** will remove this staging repository and delete the artifacts within. Choose this if a problem is found during testing and the artifacts need to be re-staged.

Promote

In more complex setups, choosing **Promote** will supposedly "promote" your artifacts to a "Build Profile", enqueueing them for some larger build process before release. This option was unavailable to me, so I may be misinterpreting its use.

Enabling Maven Central Sync

If you've made it this far, then your artifacts are ready to be synchronized to Maven Central.

Don't forget to comment back on your OSSRH to enable Maven Central sync for your project's groupId.

NOTE: You will only need to perform this once per groupId, not per project

Potential Plugins of interest

Some plugins that might be worth looking into:

- Maven Bundle Plugin, which supposedly helps to automate the process of creating a bundle.jar for upload

- Maven Release Plugin or Nexus Staging Maven Plugin, which supposedly helps to automate the process of uploading, closing, and promoting uploaded bundles

For example:

```
            <build>
                  <plugins>
            <!-- More plugins... -->

                        <!-- Maven Release Plugin -->
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-release-plugin</artifactId>
                                <version>2.5.3</version>
                                <configuration>
                                        <mavenExecutorId>forked-path</mavenExecutorId>
                                        <useReleaseProfile>false</useReleaseProfile>
                                        <arguments>-Psonatype-oss-release</arguments>
                                </configuration>
                        </plugin>


            <!-- More plugins... -->


          </plugins>
        </build>
```

Third-Party Dependencies

What do you do if one of your dependencies has not been uploaded to The Central Repository? All is not lost!

You could reach out to them to see if they are willing to perform the steps necessary to publish them, but it's unlikely that they will be open to this idea if it was not already in their plans, or if they don't use Maven at all..

If the project is open-source and accepts contributions, you could contribute a modified pom.xml allowing them to publish their artifacts to OSSRH or similar.

If you can obtain permission from the developers, they may allow you to publish it on their behalf:

1. First and foremost, ask for the developers' permission to publish their code, consult the license, etc
    a. ***DO NOT*** publish another developer's code without their permission!
2. Locate the source code (if possible)
    a. If found, generate a JAR containing the source code (don't forget to include the LICENSE, if necessary):

        ```
        jar -cvf indri-5.11-sources.jar *
        ```

    b. If found, generate JavaDoc HTML from the source code and bundle it into a separate JAR (don't forget to include the LICENSE, if necessary):

        ```
        mkdir -p ./target/javadoc
        javadoc -d ./target/javadoc lemurproject.indri
        cd target/javadoc
        jar -cvf indri-5.11-javadoc.jar *
        ```

    NOTE: Scala projects can supposedly skip this step
3. Locate compiled binaries (you can choose to download them, or build them yourself if you are able)
4. Author a simple POM for this dependency. Below is an example POM.xml that might be used for **edu.illinois.lis.indri**:

```
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>edu.illinois.lis</groupId>
    <artifactId>indri</artifactId>
    <version>5.11</version>

    <name>Indri 5.11</name>
    <description>Indri is a text search engine developed at UMass. It is a part of the Lemur project.<
/description>
    <url>https://sourceforge.net/projects/lemur/</url>

    <licenses>
      <!-- Indri actually uses BSD-old / "original BSD", but its link is no longer available via
opensource.org.
             This is likely due to the controversy surrounding the 4th clause -->
      <license>
        <name>BSD-3-Clause</name>
        <url>https://opensource.org/licenses/BSD-3-Clause</url>
      </license>
    </licenses>

    <scm>
      <connection>scm:svn:https://svn.code.sf.net/p/lemur/code/ lemur-code</connection>
      <developerConnection>scm:svn:https://svn.code.sf.net/p/lemur/code/ lemur-code</developConnection>
      <url>scm:svn:https://sourceforge.net/p/lemur/code/HEAD/tree/indri/tags/release-5.11/</url>
    </scm>

    <developers>
      <developer>
        <name>David Fisher</name>
        <email>dfisher@cs.umass.edu</email>
        <organization>University of Massachusetts at Amherst</organization>
        <organizationUrl>http://www.umass.edu/</organizationUrl>
      </developer>

      <developer>
        <name>Stephen Harding</name>
        <email>harding@hobart.cs.umass.edu</email>
        <organization>University of Massachusetts at Amherst</organization>
        <organizationUrl>http://www.umass.edu/</organizationUrl>
      </developer>

      <developer>
        <name>Cameron VandenBerg</name>
        <email>cmw2@andrew.cmu.edu</email>
        <organization>Carnegie Mellon University</organization>
        <organizationUrl>http://www.cmu.edu/</organizationUrl>
      </developer>
    </developers>
</project>
```

5. Sign all artifacts with your GPG key (*.pom, *.jar, *-sources.jar, *-javadoc.jar):

```
gpg -ab indri-5.11.pom
gpg -ab indri-5.11.jar
gpg -ab indri-5.11-sources.jar
gpg -ab indri-5.11-javadoc.jar
```

   a. This will produce a ***.asc** file for each artifact

```
lambert8:5.11 lambert8$ ls -al
total 65168
drwxr-xr-x  14 lambert8  staff       476 Jun  2 16:50 .
drwxr-xr-x   5 lambert8  staff       170 Jun  2 15:48 ..
-rw-r--r--   1 lambert8  staff     93941 Jun  2 15:18 indri-5.11-javadoc.jar
-rw-r--r--   1 lambert8  staff       488 Jun  2 15:42 indri-5.11-javadoc.jar.asc
-rw-r--r--   1 lambert8  staff  16384085 Jun  2 15:41 indri-5.11-sources.jar
-rw-r--r--   1 lambert8  staff       488 Jun  2 15:42 indri-5.11-sources.jar.asc
-rw-r--r--   1 lambert8  staff    259518 May 22 10:54 indri-5.11.jar
-rw-r--r--   1 lambert8  staff       488 Jun  2 15:42 indri-5.11.jar.asc
-rw-r--r--   1 lambert8  staff      2035 Jun  2 16:49 indri-5.11.pom
-rw-r--r--   1 lambert8  staff       488 Jun  2 16:49 indri-5.11.pom.asc
```

6. Bundle everything up and upload this bundle to the Staging repo in the same way as described above:

```
jar -cvf indri-5.11-bundle.jar *.jar *.pom *.asc
```

TL;DR

Do the one-time steps:

- Generate or import a PGP key
- Upload your key to the keyserver
- Create an account on OSSRH JIRA
- Create New Project ticket

Add this to your pom.xml:

```
        <build>
          <plugins>
            <!-- More plugins... -->

            <!-- Maven GPG Plugin -->
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-gpg-plugin</artifactId>
                                <version>1.6</version>
                                <executions>
                                        <execution>
                                                <id>sign-artifacts</id>
                                                <phase>verify</phase>
                                                <goals>
                                                        <goal>sign</goal>
                                                </goals>
                                        </execution>
                                </executions>
                        </plugin>


                        <!-- Maven JavaDoc Plugin -->
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-javadoc-plugin</artifactId>
                                <version>2.10.4</version>
                                <configuration>
                                        <show>private</show>
                                        <nohelp>true</nohelp>
                                </configuration>
                                <executions>
                                        <execution>
                                                <id>attach-javadocs</id>
                                                <phase>package</phase>
                                                <goals>
                                                        <goal>jar</goal>
                                                </goals>
                                        </execution>
                                </executions>
                        </plugin>


                        <!-- Maven Sources Plugin -->
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-source-plugin</artifactId>
                                <version>3.0.1</version>
                                <executions>
                                        <execution>
                                                <id>attach-sources</id>
                                                <phase>package</phase>
                                                <goals>
                                                        <goal>jar</goal>
                                                </goals>
                                        </execution>
                                </executions>
                        </plugin>

            <!-- More plugins... -->
          </plugins>
        </build>
```

Execute this:

```
export GPG_TTY=$(tty)
mvn clean deploy
```

If all goes well, you should be prompted to enter the passphrase for your PGP key, and your artifacts should be deployed!

Common Problems

- ioctl failed to allocate tty
  - Execute **export GPG_TTY=$(tty)** before executing
- ***.asc.asc** files in **target/** folder
  - GPG plugin is executing twice, double-check your POM settings against the command executed
  - **NOTE**: It is unnecessary to explicitly specify the **gpg:sign** or **verify** goals
- HTTP 400
  - You are deploying a SNAPSHOT to a release repo, or vice versa