Apache UIMA™

Apache UIMA™ Development Community

Version 3.6.0-SNAPSHOT

The document is a manual for maintainers of Apache UIMA, specifically focusing on aspects such as preparing releases etc. Although this is part of the documentation of the UIMA Java SDK, many aspects also apply to other Apache UIMA sub-projects.

License and Disclaimer

Trademarks

UIMA Maintainer's Guide

Chapter 1. Maintainer Setup

1.1. Platform: Java

Most Apache UIMA code is written in Java. To build it, you need a recent Java Development Kit (JDK) which can be obtained from the Eclipse Adoptium project - but there are also plenty of other JDK vendors offering free JDKs (or even ones with commercial support).

The minimum Java version required for building Apache UIMA is: **17** The minimum Java version required for using Apache UIMA as a library is: **17**

It is recommended to install the latest long term support (LTS) Java version and use it for development. When performing a release, a minimum-version JDK should be used to ensure compatibility for downstream users.

1.2. Version control: Git and Subversion

Apache UIMA projects use Git although some projects still reside in Subversion and releases are staged to the ASF Subversion repository. Thus, you should install a Subversion client as well as a Git client - best the latest available version. If you are using an IDE, you may also care to install a suitable Subversion in addition to the Git plugin for the IDE. Most IDEs come with Git plugins already pre-installed. If you are lucky, it might be enough for you. If you need to work with one of the older Subversion repositories of the project, having the Subversion plugin is handy as well.

NOTE Configure your SVN client to set the eol-style to native, for newly created files; see https://apache.org/dev/svn-eol-style.txt for instructions on how to do this.

1.3. IDE: Eclipse or another one

Eclipse is usually the IDE of choice for UIMA developers because UIMA provides several Eclipse plugins to facilitate editing UIMA XML descriptor files and the example projects are also currently provided as Eclipse projects.

NOTE You can also use another IDE like IntelliJ, Netbeans or even VSCode because the builds themselves are largely driven by Maven.

1.4. Build tool: Maven

Most Apache UIMA sub-projects are built using Apache Maven version 3 or higher. Maven offers a largely declarative and convention-driven build process and in particular automatically downloads required third-party libraries from the internet, in particular from the Maven Central repository.

You can download the latest Maven version from https://maven.apache.org/download.html or install it using your favourite package manager. Most Java IDEs already come with Maven support pre-installed. If your IDE does not offer Maven support out-of-the-box, you may want to install a suitable plugin.

- Download the latest Apache Maven from .
- Set up your `PATH` to use this version.
- (Optional, but is needed for some JVM/platforms, to give the JVM enough room to build things). Set the environment variable `MAVEN_OPTS` to `-Xmx800m -XX:MaxPermSize=256m`

1.4.1. Maven toolchains

To ensure that an Apache UIMA build is actually compatible with the minimum system requirements, it is important to build it against a minimum requirements environment. However, it is possible that Maven or plugins we use no longer are able to run on the same minimum system requirements that Apache UIMA aims to meet and thus for building, you may need e.g. a never Java version. To ensure that we can still build against the desired target environment, we use the Maven Toolchains Plugin.

Using this plugin requires that you have a `toolchains.xml` file, typically in your `~/.m2` folder. This file should contains at least a toolchain declaration for the Apache UIMA minimum Java version, e.g.:

Example `toolchains.xml` file

```
<toolchains>
  <toolchain>
    <type>jdk</type>
    <provides>
      <version>17</version>
    </provides>
    <configuration>
      <jdkHome>/PATH/TO/A/JDK_17/INSTALLATION</jdkHome>
    </configuration>
  </toolchain>
</toolchains>
```

Chapter 2. Project structure

`README.md`
: Version-independent description of the project

`RELEASE_NOTES.md`
: Specific information about the last release.

`NOTICE.txt`
: Copyright information

`LICENSE.txt`
: License information

`Jenkinsfile`
: Configuration file for building the project on the ASF Jenkins.

`pom.xml`
: Entry point for Maven-based (Java) projects.

`.asf.yaml`
: Configuration file for the ASF GitHub integration.

`.gitignore` / `.gitattributes``
: Git configuration files as required

`.github`
: GitHub configuration and tempalte files

Chapter 3. Maven Build

This section describes various functionalities provided by the UIMA Parent POM and how to use them in downstream projects.

3.1. Issues fixed report

In order for your project to generate an **issues fixed** report from the ASF Jira, add an empty file called `marker-file-enabling-changes-report` to the root of your Maven project. This activates the `generate-changes-report` profile from the parent POM.

You can also generate this report manually (for instance, if you want to have a look at what it will produce) by going to top level project being released (e.g., `uima-uimaj`) and issuing the maven command:

```
mvn changes:github-report -N
```

Each time this profile/plugin is run, it creates an updated report in the top level of this project. This report doesn’t need to be checked into source control.

3.2. Auto-staging of release candidates

Once the release build is complete, the artifacts need to be uploaded to the ASF Subversion repository for voting. To avoid having to perform this step manually, the Maven build includes an auto-staging mechanism. To use this mechansim, place an empty file called `marker-file-enabling-auto-staging`. This file activates the profile `apache-release-rc-auto-staging`.

Additionally, you have to add an execution to your project’s root Maven POM that copies the release artifacts to a local staging folder. At the start of the build, the auto-staging profile will check out the ASF Subversion repository RC staging spot to that local staging folder. Then your execution kicks in to copy the RC artifacts into the local folder. Finally, the auto-staging profile will add and commit the artifacts to Subversion.

Below is an example of how to copy the release artifacts to the local staging spot. The variables `$/home/jenkins/jenkins-agent/workspace/UIMA/UIMA Java SDK Documentation nightly/uimaj-documentation/target/staging` and `$uimaj-documentation-3.6.0-SNAPSHOT-RC-${staging-timestamp}-${candidate-id}` are provided by the UIMA Parent POM.

```
<profiles>
  <profile>
    <id>apache-release-rc-auto-staging-config</id>
    <activation>
      <property>
        <name>!disable-rc-auto-staging</name>
      </property>
    </activation>
    <build>
```

```
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-antrun-plugin</artifactId>
          <inherited>false</inherited>
          <executions>
            <execution>
              <phase>install</phase>
              <goals>
                <goal>run</goal>
              </goals>
              <configuration>
                <target>
                  <mkdir dir="${staging-local-root}/${staging-folder}" />
                  <copy todir="${staging-local-root}/${staging-folder}">
                    <fileset dir="${project.build.directory}">
                      <include name="uimafit-${project.version}-*.zip" />
                      <include name="uimafit-${project.version}-*.zip.asc" />
                      <include name="uimafit-${project.version}-*.zip.sha512" />
                    </fileset>
                  </copy>
                </target>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

Also, it is necessary to add credentials to the `settings.xml` for the server to which the release candidate artifacts are staged. Normally, the server is `dist.apache.org`, so you need to add an entry such as e.g.

```
<server>
  <id>dist.apache.org</id>
  <username>USERNAME</username>
  <password>ENCRYPTED_PASSWORD</password>
</server>
```

To test the auto-staging mechanism, you can set up a local Subversion repository and then run a build that skips the Maven Deploy Plugin and which is configured to use your local Subversion repo for auto-staging:

- Initialize a local svn repo: `svnadmin create /my/local/testrepo`
- Do a test build: `mvn -Papache-release -DskipTests -Dmaven.deploy.skip -Dstaging-scm-root='scm:svn:file:///my/local/testrepo/' clean deploy`

- Check if the commit made it in: `svn log file:///my/local/testrepo/`

Chapter 4. Release Process

The Apache UIMA project mainly releases:

- Sub-projects like Apache UIMA Java SDK, uimaFIT, RUTA etc.
- Some Maven build tooling components that need to be in the Maven repositories to support our Maven processes.

Releases show up in the Maven central repository and/or as downloadable artifacts listed on our downloads pages.

4.1. Release planning

Release planning happens in the issue tracker of the particular project on GitHub.

As part of the planning, new issues are created and typically one somebody assigns the issue to themselves, they also set target version to the next bugfix or feature release in line.

There is no fixed release schedule. Anybody may ask for the preparation of a new release. A committer then has to step up to act as the release manager and to perform the release process.

4.2. Performing the release

4.2.1. Preparations

▼ *Create release issue and release preparation branch*

Our development branches (i.e. `main` and `maintenance/*`) should be protected, so you cannot run a release directly on them. So in order to start a release, first create a release issue to track the release progress and then a corresponding release preparation branch in the repository. Release preparation branches for feature releases should be based off `main` whereas branches for preparing bugfix releases should be based off a `maintenance/XXX` branch. Once the release vote is complete, the preparation branch is then merged just like any other pull request.

▼ *Update `README.md` file*

If `README.md` file contains version references, update them. E.g. if you have a Maven dependency snippet in there. Optimally, the `README.md` file should not have any contents that need to change from version to version.

▼ *Update `RELEASE_NOTES.md` file*

Update the release notes for the release. In particular, include the notable changes (typically all features and bug fixes). You can use this list later for the release announcement mail as well.

Also mention any important changes regarding backwards compatibility.

▼ *Check the `LICENSE` and `NOTICE` files for the binary release*

There may be a `[project-root]/src/main/bin_distr_license_notices` folder containing `LICENSE` and `NOTICE` files which are used for preparing the binary release packages. If the release includes new or updated dependencies bundled in the binary release packages, then these files need to be updated with the respective content from the `LICENSE` and `NOTICE` files that may be present in these bundled dependencies (inside the JARs going to the `lib`) folder.

▼ *Make sure to remove all SNAPSHOT repositories and SNAPSHOT dependencies*

The Maven release plugin will complain if there are still any `SNAPSHOT` dependencies being referenced that are not part of the release. However, it will **NOT** complain if there are still Maven SNAPSHOT repository declarations in the POMs. Check in particular the parent pom for SNAPSHOT repositories and comment them out or remove them.

▼ *Update API reference version in POM*

Update the parent-pom settings for API change reports setting `api_check_old_version` to the correct previous version to use.

▼ *Check that the Eclipse `.classpath` files which are checked in as part of the examples have the proper version numbers for their JARs*

If the release includes Eclipse projects as examples and the release includes also new or updated dependencies, the Eclipse `.classpath` files in the example projects may need to be updated to include the new libraries.

NOTE There may be a generation process involved. E.g. in the UIMA Java SDK, the template for the `.classpath` files can be found in `uimaj-examples/src/main/eclipseProject/classpath`.

4.2.2. Building

▼ *Clean local m2 repository*

Purge your local maven repository of artifacts being built by running in the top level directory you will be building from:

```
mvn dependency:purge-local-repository
```

Note that this will immediately re-resolve the dependencies from the maven repositories you have configured.

For many multi-module projects, this will fail because it purges things that other modules need. So, the alternative is to just delete the `.m2/…/org/apache/uima/…`` directory on your build machine.

▼ *Commit all changes and check out in a new clean build location*

Make sure all changes are checked into source control. Then checkout (not export) from source control the project(s) you'll be building, into a new **build** location, and do all the building from there.

If you instead choose to build from your **working** source control checkout, insure it's up-to-date with all changes that others may have checked into the release branch.

▼ *Run a trial build locally with* `-Papache-release`

Do a trial build of the release candidate:

```
$ cd YOUR-BUILD-DIRECTORY
$ mvn clean install -Papache-release -Ddisable-rc-auto-staging
```

The `-Papache-release` is used to have the build mimic the build actions that would be taken when the release plugin is running the release build. The `-Ddisable-rc-auto-staging` ensures that you do not need Subversion and also that you do not (accidentally) stage the release candidate artifacts to the Apache staging area.

▼ *Check the issues report in* `issuesFixed` *if it looks ok*

The build includes a generated set of issues fixedin this release. To make this accurate, go through the issues and ensure the ones you are including in the release are closed, and that the milestone is set for each issue that is part of the release.

▼ *Do the release build (*`mvn -DautoVersionSubmodules=true release:prepare release:perform`*)*

We use the `maven-release-plugin` to do the releasing. In the prepare phase, it updates the trunk artifacts to remove the `-SNAPSHOT` suffix, commits it to trunk, and then does an SVN copy or GIT Branch of the trunk or master to create the tag. Then it updates the trunk artifacts to the next version-SNAPSHOT, and commits that.

The `release:perform` goal checks out the tag and builds/tests/installs and deploys it to the NEXUS staging repository.

During `release:prepare`, the release plugin asks what the next levels should be and what the tag name should be, and unless there's a good reason, we take the defaults (by just hitting enter).

When releasing a multi-module project where all the submodules have the same release version as the root project (e.g., uimaj-distr), you can have the release plugin set the version for all the submodules the same value as the root, automatically, just use this form of the `release:prepare`:

```
$ mvn release:prepare -DautoVersionSubmodules
```

In the past, we added a suffix representing the release candidate to the tag, e.g. `-rc1` for release candidate 1, etc. However, the URL for this tag becomes part of the released POM. After a successful vote, we would have upgraded the release candidate to the final release by renaming the tag in source control. At that point, the URL in the POM would have become invalid. For this reason, it was decided to **NOT** add the `-rc1` to the tag anymore.

The release plugin automatically signs everything that needs signing using gpg. It also builds the sources.jar, and one overall (for multi-module projects) source-release.zip file, which can be later obtained and should be an (approximate) copy of the tag for that artifact, and once unzipped, should be buildable, using `mvn install`.

Normally, everything built is uploaded to the Apache's Nexus Staging repository. However, for the (large) distribution objects, such as the source and binary distributions for UIMA Java SDK etc., the "deploy" step is skipped. These artifacts, instead of being "distributed" using the Maven central repository, are distributed using the Apache Mirroring System.

POMs can refer to other artifacts in several ways, for example via the `<parent-pom>` element, or via a `<dependency>` element. Often, a release will involve releasing together multiple modules (all at `-SNAPSHOT` levels) that refer to one another using these elements. When that happens, the references in these two elements are automatically updated during the release process, from `xx-SNAPSHOT` to `xx` for the tag, and then to the next development level, for the trunk.

Exception to this: `-SNAPSHOT` suffixes are not updated for references within plugins.

Note that any JARs, Zips, Tars, tar.gz artifacts must be signed by the Release Manager. When `-Papache-release` is active, the GPG Maven Plugin runs and signs the artifacts with the user's default GPG key. If you have multiple keys on your system, make sure to switch default to the right key before the release.

4.2.3. Staging

▼ *Close the staging repository at http://repository.apache.org/*

You can upload to the Nexus Staging repository several independent artifacts; they will all get added to the same unique temporary staging repository Nexus creates. Once all the artifacts are in place, you log into https://repository.apache.org using your ASF LDAP credentials, go to your staging repository, and **close** the repository. After that, nothing more can be added. If you deploy another artifact, it will create a new staging repository.

NOTE

If you **forget to close the repo**, it will be open when you do your next release candidate, and then you'll have in the repo both release candidates, (with later files overwriting newer), which if any file names have changed, will **create a mess.** So be sure to **close** (and **drop** as appropriate) any previous repo before starting a `release:perform` for a new release candidate, so they deploy into a **fresh** empty staging repo.

If you have several artifacts to release, and you want subsequent artifacts to depend on the released versions of earlier ones, you can do this, by releasing the first one, then releasing subsequent ones that depend on that, etc. This works because the first one you release will get built with the release version and installed to your local repository, as well as the Nexus staging repository. So subsequent ones that depend on the release version of previous ones, will find that in your local repository.

If you forget something and close the staging repository too soon, just continue as if you hadn't. Subsequent release artifacts will go into another newly created staging spot on Nexus. The downside of this is that you'll have to tell the **voters** about multiple staging repos.

4.2.4. Voting

▼ *Call for a vote on the developer mailing list using the email template below*

The release candidate typically consists of

- assembly source and binary distributions,
- the associated source control tag, and
- the individual Maven module artifacts.

The source and binary distributions are manually copied by the release manager to the Apache distribution SVN in the dev/uima spot, to make them available for review. The Maven module artifacts are found in the Nexus staging repository, and are available once the release manager "closes" the repository.

After things are staged, you write a note to the dev list, asking for an approval vote. You need to provide the url(s) of the closed staging repository in the note so the approvers can find the code to check, the source control tag corresponding to the release, and if needed, and the place in the distribution SVN where the source and binary distributions being proposed are found. The [VOTE] email should be based on similar previous votes, and include instructions to testers on how to set up their maven settings.xml file to specify the particular staging repository (or repositories, if more than one is being used).

Release candidate vote email template

```
Subject: [VOTE] UIMA Java SDK X.Y.Z RC-N

Hi,

the Apache UIMA Java SDK X.Y.Z RC N has been staged.

This is a bugfix / feature release.

__Paste list of issues from the RELEASE_NOTES file here__

Issues:
https://issues.apache.org/jira/issues/?jql=project%20%3D%20UIMA%20AND%20fixVersion
%20%3D%20X.Z.YSDK
Dist. artifacts:     https://dist.apache.org/repos/dist/dev/uima/uima-uimaj-X.Z.Y-
RC-N/
Eclipse Update Site: https://dist.apache.org/repos/dist/dev/uima/uima-uimaj-X.Z.Y-
RC-N/eclipse-update-site-v3/uimaj/
Maven staging repo:
https://repository.apache.org/content/repositories/orgapacheuima-1268
GitHub tag:          https://github.com/apache/uima-uimaj/tree/uimaj-X.Z.Y

Please vote on release:

[ ] +1 OK to release
[ ] 0   Don't care
[ ] -1 Not OK to release, because ...

Thanks.

-- __Release manager name__
```

▼ *Mandatory release candidate signatures and hashes validation*

Before casting +1 binding votes, individuals are REQUIRED to download all signed source code packages onto their own hardware, verify that they meet all requirements of ASF policy on releases as described below, validate all cryptographic signatures, compile as provided, and test the result on their own platform.

Source: https://www.apache.org/legal/release-policy.html#release-approval

Create a release candidate validation folder

```
% mkdir rc-validation
% cd rc-validation
```

First we fetch the staged artifacts from the ASF staging spot as well as from the ASF Maven staging repository. If there is an Eclipse update site, it should be included under the ASF staging spot.

Fetch artifacts

```
% svn export https://dist.apache.org/repos/dist/dev/uima/RELEASE-CANDIDATE-ID
% lftp -e "mirror org; exit"
https://repository.apache.org/content/repositories/STAGING-REPO-ID
```

All files (except hash files,detached signatures, or a few files like `maven-metadata.xml` or 'LICENSE' etc.) should have a detached signature. Check that this detached signature exists and validates against the file.

Check GPG signatures

```
% find . -not '(' -name '*.md5' -or -name '*.sha*' -or -name '*.asc' -or -name
'maven-metadata.xml' -or -name 'DEPENDENCIES' -or -name 'LICENSE' -or -name
'NOTICE' ')' -type f -print0 | xargs -I '{}' -0 -n1 -S 2000 gpg --verify '{}'.asc
'{}'
```

For the same set of files for which we checked the signatures, also check the SHA512 hashes are valid.

Check SHA512 hashes

```
% find . -not '(' -name '*.md5' -or -name '*.sha*' -or -name '*.asc' -or -name
'maven-metadata.xml' -or -name 'DEPENDENCIES' -or -name 'LICENSE' -or -name
'NOTICE' ')' -type f -print0 | xargs -I '{}' -0 -n1 -S 2000 zsh -c 'cd `dirname
{}`; sha512sum -c `basename {}.sha512`'
```

Checking SHA1/MD5 files generated by the Maven Repository is a bit tedious because the hashsum files do not contain the filename and so the `sha1sum` and `md5sum` checking functionality does not work. We have to do set up a little script to help us.

Check SHA1/MD5 hashes

```
% cat > checkHashes.sh <ENTER>
tmp="$(mktemp /tmp/tmp.XXXXXXXXXX)"
md5hash=`cat $1.md5`
sha1hash=`cat $1.sha1`
echo "$sha1hash $1" > "$tmp"
printf "SHA1 "
sha1sum -c "$tmp"
echo "$md5hash $1" > "$tmp"
printf "MD5  "
md5sum -c "$tmp"
rm "$tmp"
<CTRL-D>

% chmod +x checkHashes.sh

% find . -not '(' -name '*.md5' -or -name '*.sha*' -or -name '*.asc' -or -name
'maven-metadata.xml' -or -name 'DEPENDENCIES' -or -name 'LICENSE' -or -name
'NOTICE' ')' -type f -print0 | xargs -I '{}' -0 -n1 -S 2000 ./checkHashes.sh '{}'

% rm checkHashes.sh
```

Check that the contents of the sources ZIP match the repository tag

```
% wget https://github.com/apache/[project]/archive/[release-commit-hash].zip
% unzip [release-commit-hash].zip
% diff -r ...
```

Run a local build from the sources ZIP

Typically, we have Maven projects, so this would running a Maven Build

```
% mvn clean install
```

For more information on the ASF release policy, please see the Apache Release Policy document.

▼ *Additional release quality checking*

- Check the issues-fixed report
- Check the release notes
- Install plugins into Eclipse
- Try out an example
- …

▼ *Vote*

Send an email to the developer mailing list indicating your vote.

For more information, please refer to the Apache Voting Process.

▼ *Close vote and post vote results to the developer mailing list (wait at least for 72 hours, at least 3 +1 votes required for release, sign mail using same GPG key that was used to sign release)*

Example vote results mail

```
Subject: [RESULT][VOTE] UIMA Java SDK X.Y.Z RC-N

Hi all,

the vote passes, with X +1 and no other votes received.

+1 Person A
+1 Person B
+1 Person C
...

No other votes received.

Thanks to all who voted!

-- __Release manager name__
```

4.2.5. Publishing

▼ *Copy the release artifacts from the staging spot to the dist spot in SVN*

The staging spot and the release spot are in the same (large) ASF Subversion repository. So instead of uploading the artifacts again, we can simply copy them from the staging spot at https://dist.apache.org/repos/dist/dev/uima/ to the proper locations under https://dist.apache.org/repos/dist/release/uima/.

Note that the Eclipse Update Site which was a subfolder in the staging spot must now be copied to the proper location in the P2 composite update site.

▼ *Copy existing Eclipse update site to the archive spot*

```
svn copy -m "create eclipse plugin archive for uimaj-3.0.0-3.2.0"
https://dist.apache.org/repos/dist/release/uima/eclipse-update-site-v3/uimaj
https://dist.apache.org/repos/dist/release/uima/archive-eclipse-update-site/uimaj-
3.0.0-3.2.0
```

▼ *Delete existing Eclipse update site*

```
svn delete -m "reset main Eclipse update subsite for uimaj - delete old one"
https://dist.apache.org/repos/dist/release/uima/eclipse-update-site-v3/uimaj
```

▼ *Remove the old update site and move the new one at https://dist.apache.org/repos/dist/release/uima/eclipse-update-site-v3*

```
svn delete -m "reset main Eclipse update subsite for uimaj - delete old one"
https://dist.apache.org/repos/dist/release/uima/eclipse-update-site-v3/uimaj
```

▼ *Release staging repository at http://repository.apache.org/*

```
Promote the release(s) from the staging repositories: log on to the staging
repository again, and release the staged artifacts. This will make the artifacts
available in the Maven Central repository.
```

▼ *Create a new git tag e.g. `rel/uimaj-3.2.0` and remove the one not prefixed with `rel`*

Tags starting with `rel/` should be protected in all Apache UIMA git repositories. By prefixing the release tag with `rel/`, you make sure the tag cannot be accidentally deleted.

▼ *Merge the release preparation pull request*

Merge the release preparation pull request just like any other PR via the GitHub website.

▼ *Update website*

Update downloads

Update the download page of the UIMA website to make the new release artifacts available. This is done indirectly, by editing both the `downloads.xml` page and also by adding entries to the `xdocs/stylesheets/project.xml` page - follow the previous examples.

Update documentation

Also, things not needed to be mirrored go into our website: in the `docs/d` directory. Currently, this includes `the RELEASE_NOTES` (plus `issuesFixed`) for the release, the new documentation, and the Javadocs.

Copy `RELEASE_NOTES` and `issuesFixed` from the top level project (where the mvn `release:perform` was done from) in the directory `target/checkout/` … to the the website in `docs/d/[project]-current`. The documentation for the latest release should always be in the `[project]-current` folder. A copy of that should be created under `project-[version]`.

Example

```
svn copy -m "Creating versioned copy of current release documentation in archive"
https://svn.apache.org/repos/asf/uima/site/trunk/uima-website/docs/d/ruta-current
https://svn.apache.org/repos/asf/uima/site/trunk/uima-website/docs/d/ruta-3.2.0
```

Update news

Our main UIMA website has a **News** section that should be updated with news of the release. There are 2 place to update: One is the `index.xml` file, which has a one-line summary (at the bottom) that references a link within the `new.xml` page; and a new entry in the `news.xml` page itself. Follow previous examples.

▼ *Close the release in the issue tracker*

Update Jira version info to reflect the release status and date

▼ *Post release announcement to* `announce@apache.org` *(Cc:* `dev@uima.apache.org`, `user@uima.apache.org` *— once release has arrived at https://repo1.maven.org/maven2/org/apache/uima/uimaj-core/ — sign mail using same GPG key that was used to sign release)*

After release appears on maven central, post an appropriate announce letter.

To announce the published release send and email to

- `announce@apache.org`
- `user@uima.apache.org`

and describe the major changes of the release. Announcements should be posted from the release manager's `@apache.org` address, and signed by the release manager using the same code-signing key as was used to sign the release. For more details please refer to A Guide To Release Management During Incubation.

4.3. Re-doing a release randidate

There are two ways to reset things back so you can do another release candidate; depending on how far through the release process you've progressed.

If you've just done `release:prepare`, you can reset things back to as they were before that command by issuing `mvn release:rollback`.

Check to confirm that the source control tag for the release candidate is deleted; if not, remove it manually.

If you've done a `release:perform`, to reset the source, try doing the `release:rollback`; this may work if you haven't done a `release:clean`.

Otherwise, you have to manually change the `<version>x.y.z-SNAPSHOT</version>` back to their previous value. You can use Eclipse's search/replace to do this, or the mvn versions plugin.

If a Nexus staging repo was already created, drop it.

4.4. Preparing the next development cycle

- Consider updating dependencies and plugins as necessary
  - `mvn versions:display-dependency-updates`
  - `mvn versions:display-plugin-updates`
  - `mvn versions:display-property-updates`

4.5. Shared build resources

There are several projects in the build tooling. The following special procedure is used to release updates to these.

The parent-pom has the `uima-build-resources's version number encoded as the property

```
<uimaBuildResourcesVersion>XXXXXX</uimaBuildResourcesVersion>
```

This value will normally be set to the last released version number of the `uima-build-resource` artifact.

If that artifact is changing, during development, this will be set to the `XX-SNAPSHOT` value corresponding to the development version. When releasing, first do a release (to the Nexus Staging repository, as usual) of the `uima-build-resources` artifact, which will create a version without the `-SNAPSHOT`. Then change the `<uimaBuildResourcesVersion>` value to correspond to the non-SNAPSHOT version number of this, before proceeding to release the parent-pom artifact.