

Repository - SNAPSHOT Handling



This documentation was targetted at Maven 2.0 alpha 1. It is here only for historical reference and to be updated and integrated into the Maven documentation.

Overview

We will always *reference* artifacts using the SNAPSHOT identifier, but will never *deploy* a file with that. Deployed files will be given a version that matches the timestamp and build number of the artifact.

At release, while SNAPSHOTs may be disallowed, if not they will be resolved to the actual timestamped versions and stored in the POM. In this case, they are no longer treated as a SNAPSHOT - this is an actual version of the artifact.

Metadata will be used entirely rather than server filesystem timestamps as not all protocols support exposing them.

Assumptions

There is an assumption that there is only one repository for deployment. This is enforced by the POM currently, and a good idea nonetheless. Mirroring should be external.



Deployment Profiles

Note that this does not exclude having different repositories for different profiles (eg, dev, test, QA, production) - but in those environments where SNAPSHOTs are still used it will have its own build number.

When it comes to implement profiles, the snapshot version may also include a profile identifier, eg:

20040101.134532-qa-12

Remote Repository

In the remote repository, the files are deployed with the filename containing the timestamp and build number. An additional file is updated, which contains the timestamp and the build number. The filename depends on the repository layout:

old: /poms/plexus-container-0.15-SNAPSHOT.version.txt

new: /plexus-container/0.15-SNAPSHOT/version.txt

See the [repository layout](#) for information on the repository layout.

The format of the file is a single line containing the latest version, which is composed of the timestamped version (UTC) and the build number, eg:

0.15-20040101.100011-10

The build number is increment on each deployment to the remote repository.



Concurrent Deployment

Note that there is currently no mechanism in place to ensure atomicity, so care must be taken not to have multiple servers deploying to the same repository at the same time (though the worst effect that can happen is that the older of the two is actually recognised as the most recent)

Artifact Resolution

Whether an artifact is downloaded depends solely on the build number portion of the version: if the remote version is greater when checked, it is retrieved. If it is the same, it is not. If the version on the remote server is older, a warning is produced as this should not happen in normal usage.

The use of the build number will avoid any issues of clock skew.



Interim Migration

If the `version.txt` file does not exist, then an attempt is made to get the actual version (eg: 0.15-SNAPSHOT) as a fallback. This for now will produce a small warning, but eventually an error after we have setup a repository tool to convert existing SNAPSHOT files to their timestamped equivalents.

Local Repository

Timestamped files are not created on install in the local repository for reasons of disk space preservation. However, when a SNAPSHOT is resolved and downloaded, it is saved with its timestamp version number (eg: 0.15-20050401.150432-2).

When installing an artifact in the local repository during a build, the `version.txt` file is **not** updated. This is because the local last modification time of that file is used to determine when the next check should occur.

Instead, the file is stored using the format such as `0.15-SNAPSHOT`. On future attempts, the filesystem timestamp on this file is compared to the filesystem timestamp on the `version.txt` to determine which is newer (where `version.txt` only has its timestamp updated when its contents have been updated).



This means that every time a new remote snapshot is published, it will overwrite a local snapshot regardless of age. This is the only way to provide consistent behaviour and avoid clock skew - for example, while it might make sense to honour a local snapshot if it were newer than the remote snapshot, it may be that the local one was built from older sources and so is, in fact, older.

After re-examining use cases, we should try to isolate local development from remote development so that selection of a type of snapshot is generally "sticky" - however I believe to simply always use a local snapshot until it is removed may be confusing at this stage.

Modifying Download Behaviour

Snapshot Policies

Each repository in the project has its own update policy:

- `always` - always check when Maven is started for newer versions of snapshots
- `never` - never check for newer remote versions. Once off manual updates can be performed.
- `daily` (*default*) - check on the first run of the day (local time)
- `interval:XXX` - check every XXX minutes

This will be added as a

```
<snapshot-policy>
```

element in the POM inside a repository definition.

The check interval is based on the modification time of the `version.txt` file in the local repository, as mentioned in the previous section.

Snapshot Update Argument

The command line option `-U` or `--update-snapshots` will be added which forces the `always` behaviour for every repository. By doing this, you can have your snapshots behave more like a CVS update command: you only run with this command when you'd like to sync. up with the rest of the team, but set the repository to `never` by default.

This option is especially useful in a continuous integration environment.

Ignoring Local Snapshots

Not yet decided, but there may be the need for an option to ignore your own local snapshots, removing them and re-retrieving them from the remote repository. This is a form of clean, so `--clean-snapshots` might be appropriate. If this were the case, it may also be appropriate to always use a local snapshot if one exists over a remote build.

Once per Session

The code shall only check remote repositories for a particular SNAPSHOT once per session (as any deployment also installs locally so is in sync) - this will avoid the overhead of always checking, or doing local file calculations multiple times in a reactor build.

Share timestamp

Currently, as in m1, the timestamp will be the time of deployment of that artifact. This is not a big problem as the timestamps are generally hidden from view for the user.

However, especially when resolving them for a release, it would be much nicer for everything in a reactor to use the same timestamp to deploy with (possibly the reactor start time). This should be implemented for alpha-2.

Universal Source Directory

The implementation of the universal source directory will bring about additional functionality in relation to SNAPSHOTs.

If `<parent/>` has its version omitted, it should be treated as SNAPSHOT. This will ease the process of sharing a version amongst multiple projects in a reactor. Locally, this will use the reactor or universal source directory to locate the POM (finding the one closest to it). In the remote repository, it will look for the POM with a version matching its own (which can be derived from the path/request, even if it doesn't exist explicitly in the POM). Alternatively, for clarity, we may populate the version in the POM when it is deployed.

Topics not yet tackled

The following topics are in line with the above design, but not explicitly specified yet.

- referencing the latest release. There are very few use cases for this, and while SNAPSHOTs can be used, an actual release doesn't push up a SNAPSHOT.
- inconsistency of the 0.15-SNAPSHOT versioning approach. This means that the version bears no resemblance to Maven to the 0.15 version, though theoretically the final snapshot and the 0.15 release should be almost identical.
- branches. 1.0-SNAPSHOT and 1.1-SNAPSHOT may actually be the same branch, but different to 2.0-SNAPSHOT. The branch most likely should be separate to the actual version. Each branch could have a specific reference to the current version under development/release so that the previous point can work.
- derived artifacts. The snapshot version is associated with a POM, ensuring any derived artifacts will be resolved to the same version as the main one. This is a good thing, but it means we need to ensure that the install/deploy mechanism honours this, and that the resolver groups them for resolution as well.