

Mercury Repository Abstraction

- [Repository operations](#)
- [RepositoryReader API explanation](#)
- [RepositoryWriter API explanation](#)
- [Artifact versions special treatment](#)
- [Virtual Repository concept](#)
- [Repository Metadata](#)

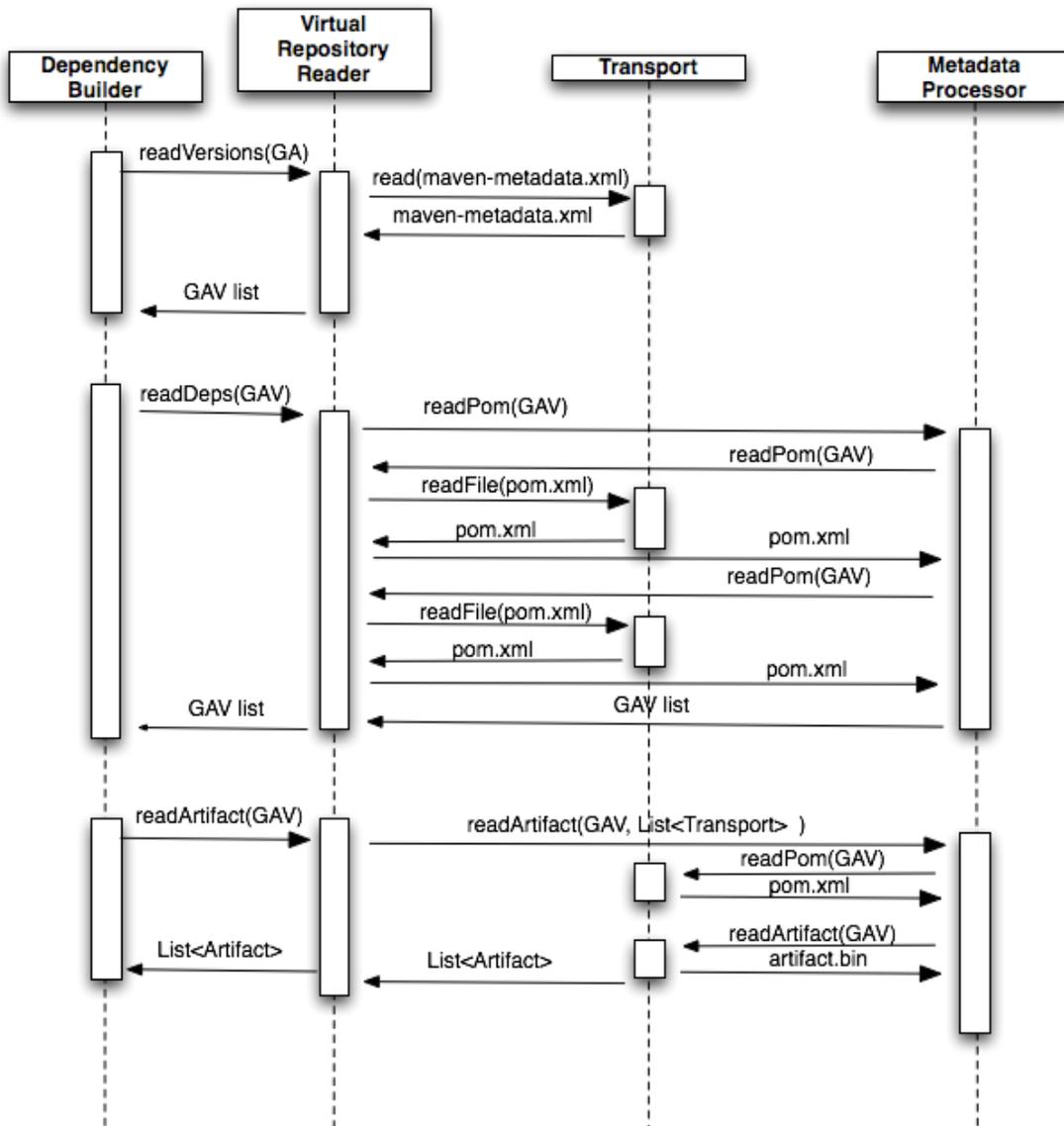
Repository operations

Mercury replaces the good old `Repository.pathOf(GAV)` with an active **Repository** API and implementations for local and remote M2 repositories.

Repository exposes only GAV-based operations to the clients, leaving storage details inside. This allows us to clearly separate concerns of using repository from gory storage details.

- **Artifact** is now represented by 3 staged of completeness
 - **ArtifactBasicMetadata** is just bare **GAV** coordinates, where **V** could be a **VersionRange**
 - **ArtifactMetadata** adds dependency as `List<ArtifactBasicMetadata>`, POM order preserved
 - **DefaultArtifact** adds the reference to a binary file
- **Repository**
 - attributes
 - **type** - a.k.a. layout, it's now used to instantiate a particular implementation
 - **local** - indicates whether it's close by and remote artifacts should be stored in it
 - **readOnly** - if local - don't try to install into it, if remote - do or do not deploy
 - **releases**
 - **snapshots**
 - methods
 - **getReader** - below
 - **getWriter** - below
- **RepositoryReader** is the abstraction that does the hiding. Simplified description:
 - `readVersions(ArtifactBasicMetadata)` - call, analogous to reading `maven-metadata.xml` or all folder names from GA directory, produces a `List<ArtifactBasicMetadata>` that meet the range criteria of the supplied parameter
 - `readMetadata(ArtifactBasicMetadata)` - call, analogous to reading `pom.xml` or all folder names from GA directory, produces a `ArtifactMetadata` with non-transitive dependencies already in. It uses supplied **MetadataProcessor** to process the `pom.xml`, **ProjectBuilder** will have to play that role until we separate dependencies from the rest of project metadata
 - `readArtifact(ArtifactBasicMetadata)` - actually reads the binary and places it into local repositories, if necessary.
- **RepositoryWriter** a simpler component with only one purpose in life - write a **DefaultArtifact** into a repository

A picture can substitute 1000 words, what about a sequence diagram ? 😊



This approach, for instance, allows one to easily write a database-based repository, and much more.

RepositoryReader API explanation

RepositoryReader has 3 major calls, that all the rest is based upon:

- **readVersions()** it receives a version query and interprets it depending on the repository (metadata for remote) contents. It is the only API that deals with version ranges and special suffixes like LATEST or SNAPSHOT, all other APIs accept version literally and don't try to interpret it
- **readDependencies()** - based on the artifact coordinates and environment (passed as Hashtable) returns artifact dependencies. It actually calls out to **DependencyProcessor** implementation to interpret POM files, thus is just a thin layer that interprets what **DependencyProcessor** digested and spit back. It repo reader also provides a **readMetadata()** callback so that **DependencyProcessor** can access all the metadata it needs
- **readArtifacts()** is the simplest of them - just reads artifacts based on supplied coordinates. It can also use results of previous calls to **readVersions()** because the latter transiently records the repository where a version was found.

RepositoryWriter API explanation

RepositoryWriter is a primitive **writeArtifacts()** implementation. Due to intimate relationship with JettyTransport, implementation of this API call in remote-repository-m2 is also transactional; it ensures all-or-nothing deployment.

Artifact versions special treatment

On top of version ranges, Repository Reader and Writer are responsible for correctly interpreting special version queries, like g:a:1.1-SNAPSHOT, g:a:1.1-20080820.195323-15, g:a:LATEST, g:a:RELEASE because those do not follow standard M2 convention for repository structure. As a result:

- g:a:1.1-**SNAPSHOT**
 - in local repo - I scan the repo dir for latest snapshot
 - in remote repo - I rely on GAV-level maven-metadata.xml to find the latest snapshot
- g:a:1.1-**20080820.195323-15**
 - in local repo - I scan the repo snapshot gav dir for this particular snapshot
 - in remote repo - I rely on GAV-level maven-metadata.xml to find this snapshot
- g:a:**LATEST**
 - in local repo - I scan the repo ga dir for latest version. Latest is defined by DefaultArtifactVersion.compareTo()
 - in remote repo - I rely on GA-level maven-metadata.xml to find latest release or snapshot
- g:a:**RELEASE** same as LATEST, but excludes SNAPSHOTs
 - in local repo - I scan the repo ga dir for latest version, excluding SNAPSHOTs. Latest is defined by DefaultArtifactVersion.compareTo()
 - in remote repo - I rely on GA-level maven-metadata.xml to find latest release

Virtual Repository concept

Working with repositories I saw a lot of places where we pass (LocalRepository, Set<RemoteRepository>) and then process them. To abstract this usage pattern in Mercury I introduced **VirtualRepositoryReader**. This object is constructed out of List<Repository> and then shifts repositories inside so that local ones are upfront, and then performs requested operations that mimic single repository operations.

There is less need in **VirtualRepositoryWriter** so I still debating the necessity of it. One use case is caused by introduction of repository **QualityRange** - see in [Mercury Version Ranges](#), here VirtualRepositoryWriter can try several repositories before writing the one, accepting quality of supplied artifact. Beyond that - we almost always write to a particular repository.

Repository Metadata

[MERCURY-5](#)