

# GEP 9 - Modularization

## Metadata

Number:	GEP-9
Title:	Modularization
Version:	4
Type:	Feature
Status:	Draft
Leader:	Paul King
Created:	2011-10-26
Last modification:	2011-10-26

## Abstract: Modularization

 Parts of GEP-9 have been incorporated into Groovy 2.0. Please see the Groovy 2.0 documentation for further details, e.g.: [Creating an extension module](#).

This GEP introduces the goals and proposed details behind a modularization effort that will:

- provide a more streamlined jar for core Groovy with other jars comprising what might be considered optional parts of Groovy
- provide Groovy library writers with hooks into the groovy runtime to allow them to extend Groovy in various ways - allowing "grapes" to be more like plugins than just traditional libraries
- auxiliary details about how the modularization might affect other artifacts, e.g. the Groovy build, wiki, jira components, etc.

This activity builds upon the work discussed previously: [Groovy 2.0 modularization](#) and the somewhat older profiles discussion [Packaging and Modularity](#).

 This document is part work in progress and part proposals and ideas. The final details may change from what is discussed here. User feedback (preferably on the user mailing list) is greatly appreciated.

## Out of scope

While we should keep a watching brief on other modularization efforts, at this stage we are anticipating structural changes to the Groovy classes which are orthogonal to eventually using something like OSGi or Java 8's Jigsaw modularization.

## Jar groupings

At the moment, Groovy provides two main jar artifacts:

- `groovy.jar` contains the core classes for Groovy and has a dependency on a small handful of critical dependent jars (i.e. `asm`, `antlr`, etc.)
- `groovy-all.jar` contains the core classes for Groovy and bundled versions of the critical dependent jars

After modularization, the main Groovy codebase will be packaged into additional jar artifacts:

- `groovy.jar` or `groovy-core.jar` (name to be finalized) will contain a smaller set of core classes for Groovy and will retain its dependency on a small handful of critical dependent jars (i.e. `asm`, `antlr`, etc.)
- numerous smaller "module/component" jars (exact names/numbers/contents to be finalized) but possibly things like `groovy-bsf.jar`, `groovy-sql.jar`, `groovy-swing.jar`, `groovy-jmx.jar`, `groovy-xml`, `groovy-tools` etc.
- `groovy-all.jar` contains the core classes for Groovy, bundled versions of the critical dependent jars and all of the classes from the module jars
- optionally we may support other "profiles" - profiles being other "fat" jars we bundle together for other environments, e.g. `groovy-android.jar` might exclude the JMX or Swing modules for instance.

Recommendations/suggestions for jar groupings are welcome. See some more details in the accompanying discussion: [Groovy 2.0 modularization](#).

One of the goals of this bundling effort is to facilitate Groovy's evolution as a language by enabling:

- less relevant modules to become deprecated and removed from the main project but still available as external libraries, e.g. `groovy-bsf` might be deemed inessential now that most users will have `jsr-223` (but it can be moved into a separate module and still available for anyone who needs it)
- the controlled migration from legacy to new versions of some functionality, e.g. we could produce a `groovy-xml-v2` library which might have breaking changes - the `v2` library might be an optional module at first but later become the main supported option - legacy users would be able to "point back to the old version" so that their scripts don't break if they wished (further details later)
- external modules could more easily be incorporated into the core distribution (with users able to customize certain aspects of what gets bundled)
- we can provide "legacy" jars which contain deprecated features or unforeseen breaking changes - library writers can incorporate the legacy jar into their libraries to more easily support using their libraries across multiple versions of Groovy

## Configuring a Groovy installation

Groovy bootstraps its configuration using a file in the `conf/groovy-starter.conf` file. It has entries like this:

```
# load required libraries
load !{groovy.home}/lib/*.jar
```

with required jars being in a `lib` directory which is part of a Groovy install.

The groovy install will now likely have a "modules" (or components or repository) directory. The `groovy-starter.conf` file will have additional entries such as:

```
# load SQL component
grab org.codehaus.groovy groovy-sql 1.9.0
# load XML component
grab org.codehaus.groovy groovy-xml 1.9.0
```

Grapes are loaded via Ivy and configured from a settings file. This file might have an additional entry:

```
<ibiblio name="modules"
root="file:${groovy.home}/modules/"
m2compatible="true"/>
```

The Ivy library (or perhaps the wharf or aether libraries) will likely become a required jar - though it may be that only a very limited form of grab is supported in which case no additional library might be necessary.

## Runtime hooks/registration

When downloading a grape, the classes from its jar are added (currently appended) to the classpath. In addition, we are planning on supporting additional integration points:

potential hook	purpose
META-INF/services/org.codehaus.groovy.runtime.CategoryMethods	allow the module to define additional category methods
META-INF/services/org.codehaus.groovy.runtime.StaticCategoryMethods	allow the module to define additional static category methods
META-INF/services/org.codehaus.groovy.runtime.ExpandoMethods	allow the module to define additional expandometaclass methods
META-INF/services/org.codehaus.groovy.runtime.DefaultMetaClasses	allow the module to define additional metaclasses similar to the current magic package mechanism**
META-INF/services/org.codehaus.groovy.runtime.SerializedCategoryMethods	allow the module to define additional category methods which have been serialized
META-INF/services/groovy/defaultImports	allow the module to define additional normal, star, static imports, aliases
META-INF/services/groovy/defaultExtensions	allow the module to define supported file extensions

META-INF/services/groovy/defaultAstTransforms	allow the module to define AST transforms
?	provide a way to register builder metadata
?	should there be a way to 'publish' new commandline level startup scripts e.g. java2groovy
?	provide a way to register a runner class, e.g. EasyB - there might also be a need to detect runner types
?	provide a way to register special compiler flags, e.g. '--indy'
?	provide a way to inject special AST customizations

\*\*This needs to mesh in with the existing magic package mechanism for defining custom metaclasses.

This is also where we could specify additional requirements, e.g. require 'invoke dynamic' - but see later also. Could we declaratively specify our security policy requirements. Or can we disable specific aspects of functionality, e.g. disable some standard global AST transform because we want to provide a better one. (There are obviously security implications for this!)

Another question is whether all these features will be available also via an API. Would such an API allow modules to be "unregistered"?

## Groovydoc/source pointers

Some modularization systems support the "installation" of documentation (and/or sources) along with the module. Should a module have a pointer to (or be bundled with) its GroovyDoc and/or source.

In the Java ecosystem, many libraries are published with their javadoc/sources according to common conventions or have javadoc available online. Do we rely on these established conventions or provide additional support? For users without access to sophisticated IDEs it could be convenient to have all the documentation available in "merged" form in one place.

## Build Impacts

Mostly discussed here: [Groovy 2.0 modularization](#)

Should each "submodule" be able to be built on its own? Will it have its own javadoc, own coding style rules, own coverage metrics etc.

## Module robustness

Should we provide a standard hook/mechanism to try to combat the following scenarios:

- you are loading a jar that depends on classes managed by the root classloader?
- a class with the same name is already on the classpath before yours?
- a class with the same name as one of your dependencies but from an incompatible version is on the classpath already
- an ability to register a "ping"/health check method to quickly test the component

## Runtime Introspection

In Groovy you can currently determine the Groovy version using:

```
println GroovySystem.version
```

which returns a String.

In [GROOVY-2422](#) it talks about the desire for additional version checks.

It also talks about listing capabilities. This could be "is invokeDynamic" available or could really just be about available loaded modules. In general, we would expect our dependencies to be specified as part of our pom and loaded for us automatically but it can sometimes be useful to do special things.

In general, should we be able to find out the list of loaded modules and versions? Or ask about which module/version a particular class belongs to?

## Assisting IDE support

We may define a standard place or convention, e.g. META-INF/services/groovy/dsld or META-INF/services/groovy/gdsl where IDEs can find DSL descriptors relevant for that module.

## Logging

Should modules themselves have a standard way to do logging? Is that j.u.l.Logger? Perhaps bridged with slf4j?

## JIRA issues

- [GROOVY-2422](#) API for checking version and capabilities of Groovy runtime

## Useful links

- TBD