How to stop dependencies leaking in 10 ~~days~~ steps

Problem Statement

Apache Geode is an application. It may be a specialized in-memory data{base} grid application, but an application nevertheless. Like all applications, Apache Geode uses different implementation and framework libraries to implement its features. Unlike most other applications, Apache Geode has the ability to have users deploy code into Apache Geode and have that code execute near the stored data.

Whilst this feature is great, it comes with restrictions to users wanting to use this feature. One of the restrictions is that a user has to be aware of the underlying libraries that Apache Geode uses. The users have to be aware that if they use these libraries in their deployable code, that these libraries are of the same version.

Using another version of a library might cause:

- The deployed custom code to behave in an unexpected manner (Apache Geode's version is used instead of theirs)
- Apache Geode to behave in an unexpected manner. (The custom code's library is used instead of Apache Geode's supported version).

Hitting either scenario could have detrimental outcomes to the Apache Geode product or the custom deployed.

To explain why one would see unpredictable behaviour, one has to understand how standard Java works. Java uses something called a ClassLoader to load all its class definitions, create instances of those class definitions and generally provide answers like "is ClassA an instance of ClassB" or "provide me all implementations of InterfaceA". In the case where custom code is deployed to the system, there may be conflicting classes (from different library versions). In this case, the ClassLoader will always apply the algorithm of first class loaded, regardless of version.

What is required is the ability for Apache Geode's library dependencies to not constrain the libraries that users have to use. The opposite is also true that the libraries the users use in their custom code deployments do not interfere with the workings of Apache Geode.

To state the requirement more simply, ***what's mine is mine and what's yours is yours***.

To achieve this, a mechanism to achieve class isolation is required. ***ClassLoader isolation*** is a topic that has been around for the longest of time and is used in many places. I.e WebContainers, Application Containers or Integration Servers (like Mule). ClassLoader isolation loosely states that classes loaded into one ClassLoader will not be visible by another ClassLoader. This has the benefit that classes of the same name can co-exist in the same JVM, as those classes will exist in different ClassLoaders, which has the effect that they don't conflict.

In addition to this, it is possible to load different versions of the same Class(es) into the same JVM, as ClassLoader isolation ensures that they do not conflict.

Proposed solution

The proposed solution has to satisfy the following goals:

- A solution that restricts Apache Geode's code and dependencies to not leak outside the scope of the application
- A solution that does not allow the deployed custom user code and dependencies to leak outside of itself and affect Apache Geode and other deployed code.
- Allow Apache Geode code and custom deployed code to still interact unimpeded
- A solution that controls context
- A solution that has long term support in the Java ecosystem
- A solution that supports SPI/API driven development (Java ServiceLoader and DI)
- A solution that has the ability to isolate classes / modules from one another transparently

In essence the solution has to control access to deployed Apache Geode / user code and its related dependencies. The solution has to support the constraining of what is exposed and what is "hidden".

In computer science we learned about components / modules as a compact construct that has defined boundaries and meta-data that describes:

- Code visibility
- Library dependencies
- Version

Currently there are two popular module / component frameworks for Java:

- Project Jigsaw (Java modules)
- JBoss modules

Project Jigsaw

Project Jigsaw, or now better known as Java modules is an attempt to fulfil the module/component definition for Java. Jigsaw adds the ability to modularize one's project code and describe boundaries and context for the module. Project Jigsaw is Java's answer to modular development, which is currently heavily used in Java's agile development approach to be able to release more often. Project Jigsaw has been included in the Java JDK since **Java 9**.

Jigsaw's module-info.java file describes:

- Module name
- Module class exports (what classes this module makes visible externally)
- Module dependencies
- Service definition and Service implementation

JBoss Modules

JBoss modules, is an open-source project born from JBoss Application/Web containers. It too, is a framework that supports modular development practices. JBoss modules was first open source in 2010 (according to maven repo), but has been around for a long time before that, most likely as early as JBoss AS 7. It is used in JBoss EAP and WildFly. It is complementary to the Java environment and not dictative.

JBoss modules uses a *module.xml* file to describe:

- Module name
- Module version
- Module class exports
- Module {library} dependencies
- Module {library} dependency versions
- Service definition and service implementation

Comparison

Both modular frameworks fulfil similar functionality, but they have some differences as described in the table below.

Comparing the two modular solutions:

Category	Project Jigsaw	JBoss modules
Native Java implementation	Yes (in the JDK)	Yes
Long term support	Yes	OpenSource Currently used in WildFly and JBoss EAP
Module metadata	Yes	Yes
Module version support	No	Yes
Dependency version support	No	Yes
Code / Service export or visibility	Yes	Yes
Dependency visibility	Yes	Yes
Runtime / Compile time	Both	Runtime only
Runtime loading of new modules	No	Yes

Module discoverability	Classpath Only	Classpath, disk location, jars
ClassLoader Isolation	No	Yes
Splitting packages across modules	No	Yes

Hybrid

JBoss modules will isolate modules from one another. This enables a behavior such that ModuleA cannot “see” or load any classes from ModuleB, unless configured to do so. This isolation is achieved by each module using its own ClassLoader.

But JBoss modules has the problem that it is a Runtime Only construct, which means that it breaks at runtime if there is any problem, whereas Project Jigsaw is both compile and runtime. Which allows for earlier problem resolution.

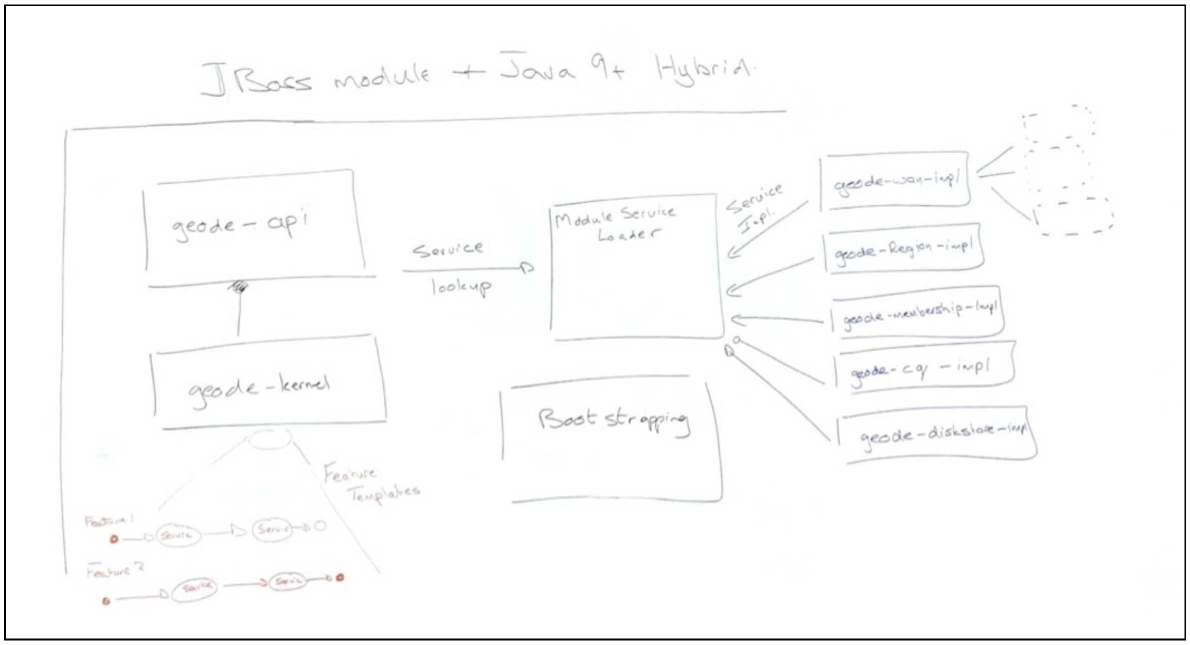
Ideally a hybrid implementation is needed. An implementation that provides both compile and runtime constraints AND allows for runtime class isolation.

But both modular frameworks use different meta-data definitions.

BUT..

On closer inspection Project Jigsaw’s *module-info.java* metadata file looks very similar to JBoss modules' *module.xml* file. The reason for that is that JBoss modules is the implementation that seeded Project Jigsaw. What this means is that Project Jigsaw’s metadata file closely maps to that of JBoss Modules. In theory it would be possible to describe the module using Project Jigsaw and have a processor that processes the metadata file to register the modules using JBoss modules’ ModuleSpec API.

The below image best describes a hybrid implementation of both modular frameworks.



It is based on the notion that every Apache Geode component is described as an implementation module. Apache Geode, at its core will become a micro-kernel. The kernel of the product will have:

- Geode API module
- Geode Operational flows
- Geode Module Framework (Based on JBoss Modules)
- Geode Bootstrapping

Apache Geode will then have a set of component implementation modules. The implementation modules will be dependent on the public Geode API module and implement the service interfaces. These are depicted as:

- Geode-wan-impl
- Geode-cq-impl
- ….

Geode API

The Geode API will define all SPI/API interfaces for Apache Geode. There will be no implementation classes in this module at all. Just API classes for all services that the Geode operational flows requires

Geode Operational Flows

All Apache Geode operations can be described as a set of operational flows. This module will be the core of the system. It will describe the "how" of Apache Geode's functional/operational flows. It will have a dependency on the Geode API module, and all flows will be defined in terms of service interface calls.

Geode Module Framework

This module will be responsible for the loading and maintaining of loaded modules. All modules will be loaded by this modular framework. Initially this module will use JBoss modules as its initial implementation.

Geode Bootstrapping

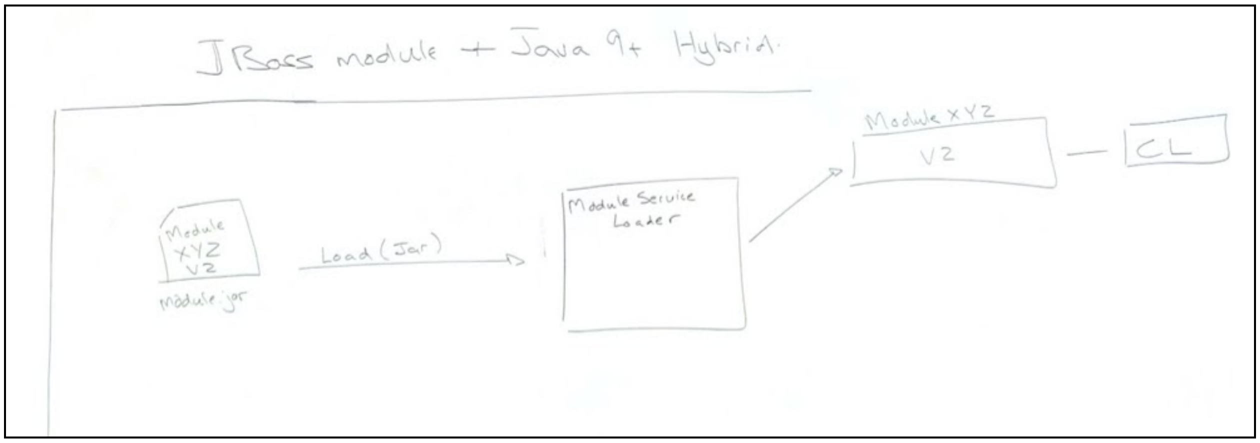
Geode Bootstrapping will be a module that is concerned only with the bootstrapping of Apache Geode. The loading service implementations and using Dependency Injection paradigms bootstrap Apache Geode with service implementations, from the Geode Module Framework.

Runtime Deployment of code

This hybrid approach will enable the deployment of code at runtime. Actually, JBoss modules enables this, BUT using the hybrid philosophy, custom code can be developed and based on the Geode API library. Using Project Jigsaw, all compile time constraints will ensure that these modules don't depend on code that is not exposed by the system.

Then using the Geode Modular framework, this code can then be loaded into Apache Geode.

Using JBoss modules' classloader isolation to its advantage, that code will be loaded into its own classloader, not affecting any other code within the Apache Geode system. The classloader isolation feature also allows users to use libraries and frameworks of their choice, without affecting any other module or Apache Geode.



First Contact

TBC…. Let’s discuss this once a clearer understanding of the subject matter is achieved.

Solution Background

JBoss Modules

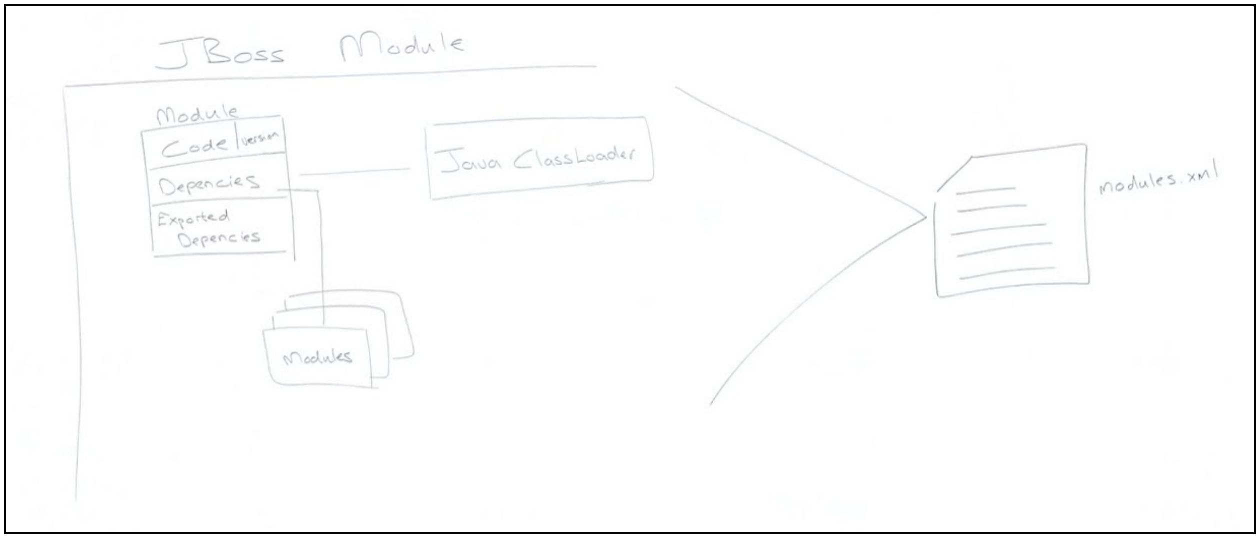
JBoss modules is an open-source library that is a standalone implementation of a modular classloading environment. It is based on the notion of ClassLoader isolation. Where each Module is backed by its own ClassLoader. This approach isolates the classes and resources that are loaded by each module. This means, that the classes loaded by ModuleA into ClassLoader A, will not be visible from another ClassLoader. One would have to specifically target the correct ClassLoader in order to load the correct class. Otherwise you will receive a "ClassNotFoundException", the relevant class will not be found in it.

A JBoss module is comprised of:

- Code
- Module.xml

The *module.xml* file contains all metadata relating to the module.

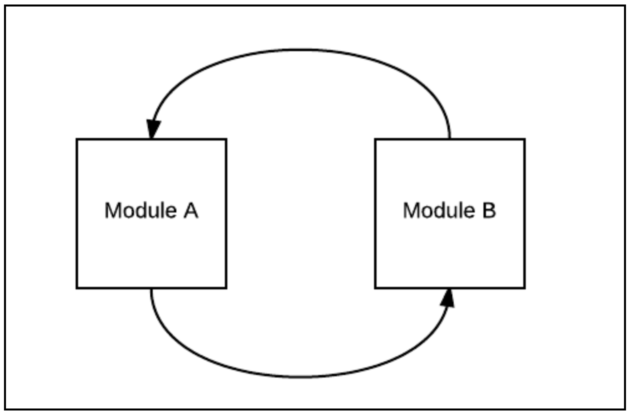
- Module name and version
- Exported Code / Services
- Dependent Modules and versions
- Resources and locations



The approach of having a *module.xml* file to describe the module allows the code in a module to definitively know what classes and resources it is sharing a classloader with. This is especially beneficial to resolve conflicts caused when more than one version of a library is used within the same application. This approach is also beneficial when modules need control over what libraries it loads services and resources from.

In JBoss modules the class loading is graph based vs the hierarchical as found in the Java JDK ClassLoader. JBoss modules class loading strategy is *child-first* and not *parent-first* as in the JDK. Given two modules (A & B) define the same class, and a dependency on each other.

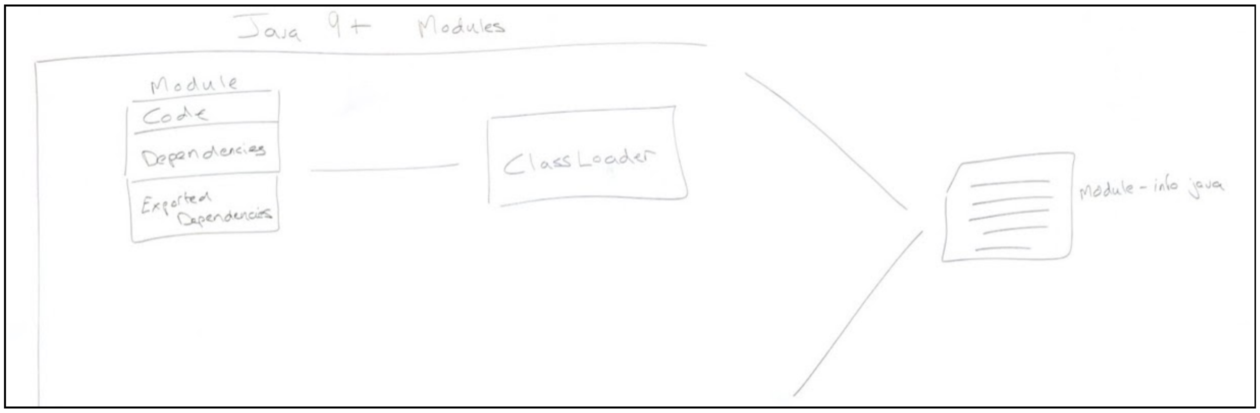
In the case of *parent-first*, if ModuleA wants to load ClassX, then it will receive the class from ModuleB, the reverse is also true. In *child-first*, if ModuleA loaded ClassX it will receive the class from ModuleA. This is far more predictable than the surprising behavior than that of a *parent-first* approach.



Project Jigsaw (Java 9 modules)

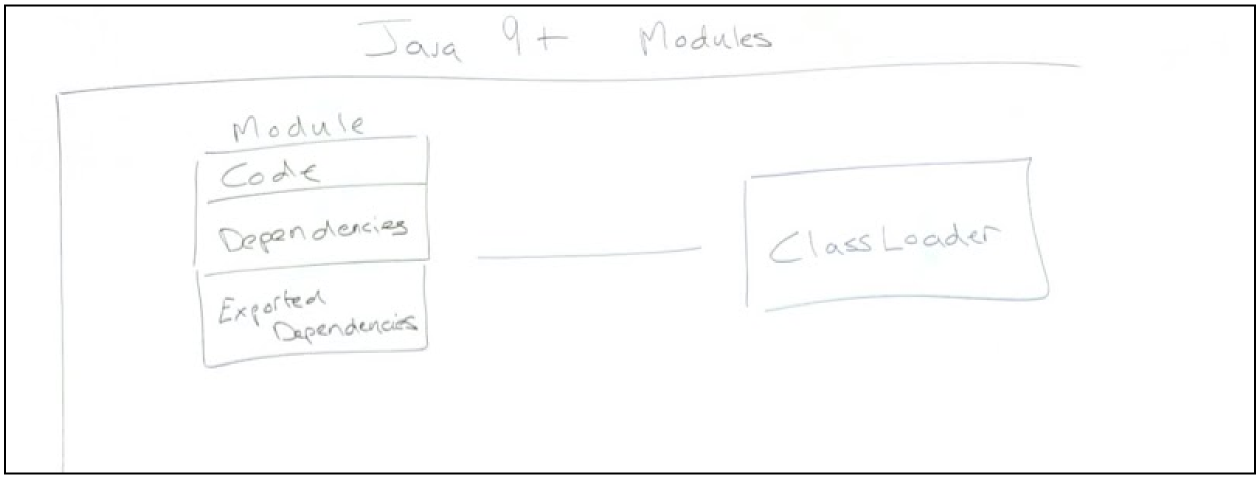
With the release of Java 9, there was a major paradigm shift that Java introduced. The JDK was completely modularized, with the help of Project Jigsaw. This paradigm shift, allowed Java to have shorter release cycles and have a more granular, modularized architecture with the ability to unit test single components, rather than having to stand up the whole Java world to test a small piece.

Project Jigsaw, or better known as Java modules, introduced Java's modularization paradigm. It is based heavily on JBoss modules, which becomes evident when comparing the meta-data files required. (which will not be directly compared here).

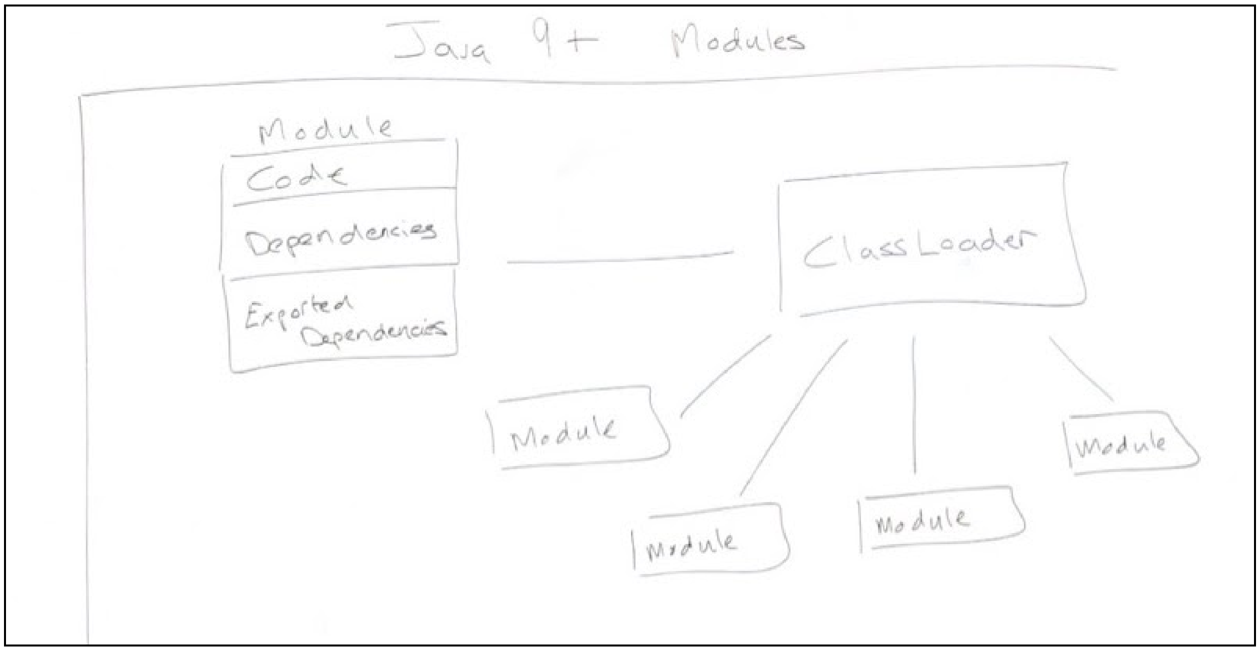


The main differences between the two modular frameworks are described in the Proposal section of this document.

The biggest difference though, is that Project Jigsaw uses only one ClassLoader.



Which has the effect that all modules that are loaded from the classpath are all registered with the same ClassLoader.



Using only one Classloader has the effect of:

- Not being able to load Classes of the same name (package + Class name)
- No isolation of classes
- No ability to load different versions of the same Class
- Everyone can access all defined classes.

What Java modules provides is:

- Ability to describe “visible” class exports
- Ability to describe dependent modules
- Ability to describe service implementation (replacement to the resource file that ServiceLoader required)
- Ability to describe “visibility” of dependent modules