**Warning!** This is a long read! Before sharing your feedback, please make sure to skim through the entire document and make sure your concern is not addressed anywhere.

# References

**Important!** We strongly advise readers to have some brief familiarity with the shared content below before proceeding further.

- git submodules
- GitHub's integration for git submodules
- repo tool (used by Android and Maven to create a global view of multiple repositories)
- git-sparse-checkout
- Maven in a Google-style mono-repo (git-sparse-checkout, `pom-template.xml`, `mr/checkout.sh`, etc.)
- How Maven wrestles with 125+ repositories

# Problems

## Excessive testing time

Tests of irrelevant modules (e.g., `log4j-osgi`) are taking too much time, hence developers grow to commit without `./mvnw verify`, which eventually is breaking the build.

### Can we decrease testing time?

- Matt put substantial effort into parallelizing the tests, though it turned out to be a futile effort since many tests mutate the JVM via `System.setProperty()` and other constants.
- Volkan thinks the lack of a Spring-like `Environment` abstraction, and hence dependence on static members and constants, force tests to use `forked=true`. This adds up to "the test lag" and makes JVM-reuse impossible.
- Maven sucks at parallelization. It is simply a hilarious mystery why we can't run `log4j-taglib` and `log4j-osgi` modules' tests in parallel given they have nothing in common and `forked=true`.

### Proposed solutions

1. Create a separate Maven project with independent release cycle
2. Improve test parallelization
3. Introduce Maven profiles to ease testing only relevant modules (and skip others)

## Coincidental survival

Many modules (Jackson-based layouts, JNDI, Liquibase, etc.) are supposed to be dead, long ago. Developers want them to be abandoned, though this conflicts with our backward-compatibility promises in a major release.

- This is a major driver for many developers to break down the core. They simply don't want to be bothered with certain modules anymore. They see the multi-repository setup as an enabler to dig graves for such modules.
- Releasing inactive/dead modules along with the rest of the active ones creates the impression that they are not dead.

### Proposed solutions

- Mark them deprecated and release 3.0 without them
- Create a separate Maven project with independent release cycle and let them rot there

## Conflicting transitive dependencies

Some modules have conflicting transitive dependencies (e.g., certain versions of Guava).

### Proposed solutions

1. Move such ITs to their individual projects where they can inject whichever dependency version they need

## Separate API and implementation

APIs should be stable and shouldn't be getting updates unless there are changes.

### Proposed solutions

1. Create a separate Maven project with independent release cycle

# Common implementation procedure

1. Break down `log4j-core` et al. in `2.x` to match the modules between `2.x` and `main`
    a. Change is binary backward compatible, though classpath needs to be adjusted
2. After moving modules to separate Maven projects
    a. Implement a tool to generate component (appender, layout, filter, etc.) docs from code
    b. Use Antora to generate the website[1] such that
        i. Prose docs (support, tutorial, configuration, architecture, etc.) are in AsciiDoc

[1] See the proposed site structure.

ii. Component docs are generated the from the tool implemented above

# Common open questions

List of open questions that are shared by both mono- and multi-repo proposals.

### Which module goes to which new Maven project? (aka. module breakdown)

### In which Maven project will the BOM module be situated?

### Will BOM contain all modules from all Maven projects?

### What will be the BOM release cadence?

### How shall we establish inter-project integration?

Assume `log4j-core` and `log4j-api` (used by `log4j-core`) are split into their own Maven projects. How can we periodically run tests in `log4j-core` against the most recent SNAPSHOT version of `log4j-api`?

# Multi-repo proposal

Break `logging-log4j2` repository down to more fine-grained repositories. Note that this implies multiple Maven projects, and hence, release cycles too.

+ Decreases testing time
+ Enables module rot
+ Enables having different shared transitive dependency versions (e.g., some modules require different versions of Guava)
+ Allows integration across versions (e.g., can test `2.x` of `log4j-osgi` against `main` of `log4j-core`)
- Requires external tooling (git submodules and repo) for a global view ala mono-repo
- Shared structure (READMEs, CI config, etc.) between repos needs to be manually sync'ed

## Procedure

1. Move modules to separate repositories
    a. At this stage, website of a moved module will still be located in the `logging-log4j2` repository

    b. The new repositories will contain both `2.x` and `main` code of the associated modules
2. Create a global repository (`log4j-global`?) to provide a global view using git submodules

## Aids

Management of a multi-repo structure is easier said than done. The following aids will be employed to ease that experience.

1. Move shared plugin configuration (`pom.xml`), profiles (e.g., `changelog-export`, `deploy`) and files (`RELEASING.md`, `SECURITY.md`, `SUPPORT.md`, etc.) to `logging-parent`
2. Use the very same build procedure everywhere and host (and reuse!) its CI configuration (`BUILD.md`, `build.yaml`, etc.) in `logging-parent`

## Open questions

### Shall we include all Log4j modules (`logging-parent`, etc.) in the `log4j-global`?

`log4j-global` will constitute a repository whose folders point to other repositories; `log4j-core`, `log4j-api`, etc. Hence, for convenience reasons, preferably, yes. – Volkan

## F.A.Q.

### Will we lose the global-view and convenience of a mono-repo?

Not exactly. Those who still want to keep that working style can use the `log4j-global` repository combining all other repositories.

# Mono-repo proposal

Break the Maven project located in `logging-log4j2` repository down to more fine-grained Maven projects using the very same repository. Note that this implies multiple Maven projects, and hence, release cycles too.

+ Decreases testing time
+ Enables module rot
+ Enables having different shared transitive dependency versions (e.g., some modules require different versions of Guava)
+ Doesn't require external tooling
+ Shared structure (READMEs, CI config, etc.) between repos can simply be sym-linked
- Requires external tooling (repo or git-sparse-checkouts) to work on only a subset of projects

- git namespace (for branches, tags, etc.) is shared and doesn't allow inter-project integration across versions (e.g., cannot test `2.x` of `log4j-osgi` against `main` of `log4j-core`)
- git commit noise

# Open Questions

## Git namespace is shared. How to do branching and releases?

Currently used global `2.x` and `main` branching scheme starts shaking when there are multiple Maven projects with their own release cycles.

1. When I start working on a release using the `release/2.22.0` branch, it doesn't indicate which project it is associated with. Same confusion happens with tagging too. Are we gonna follow a certain convention (e.g., prefixing them with the project name) for naming such git resources?
2. (Haritada "How can I, for instance, work on the `2.x` branch of `log4j-osgi` while targeting the `main` branch of `log4j-core`?

# Q&A

## Will `maven-release-plugin` work given the shared git namespace?

This is not an issue, since we won't use any plugins except `maven-deploy-plugin` while releasing. See `log4j-tools` on how this is performed. (Shared git namespace has other negative implications. See the related subject in Open Questions.)

# Procedure

1. Move modules to separate Maven projects
    a. At this stage, website of a moved module will still be located in the core Maven project
2. Find a solution for issues related with
    a. shared git namespace
    b. inter-project integration across versions
3. Implement/use a tool to work on only a subset of projects