CBNG – THE NEW BUILD TOOL USED TO BUILD MILLIONS OF LINES OF JAVA CODE AT CERN

L. Cseppentő[*], V. Baggiolini, E. Fejes, Zs. Kővári, N. Stapley, CERN, Geneva, Switzerland

Abstract

A large part of the CERN Accelerator Control System is written in Java by around 180 developers (software engineers, operators, physicists and hardware specialists). The codebase contains more than 10 million lines of code, which are packaged as 1000+ JARs and are deployed as 600+ different client/server applications. All this software are produced using CommonBuild Next Generation (CBNG), an enterprise build tool implemented on top of industry standards, which simplifies and standardizes the way our applications are built.

CBNG not only includes general build tool features (such as dependency management, code compilation, test execution and artifact uploading), but also provides traceability throughout the software life cycle and makes releases ready for deployment. The interface is kept as simple as possible: the users declare the dependencies and the deployment units of their projects in one file. This article describes the build process, as well as the design goals, the features, and the technology behind CBNG.

INTRODUCTION

The work on a Java-based Control System for the LHC was started in 1998 and the need for a unified build process emerged shortly. In 2002, an Ant-based [1] tool, CommonBuild was introduced for this purpose [2], which used an Apache JJAR [3]-based repository for dependency management. Over the years active development on these products have stopped as new competitors entered the market, first Maven [4] and then Gradle [5]. The latter tools provide an improved, de facto industry standard method for building Java programs and managing dependencies, which served as a motivation to look for a replacement.

Table 1: Evolution of Complexity of Java Software

	2005	2017
Products	130	1 000
Dependency levels	10	25
Developers	30	180

Nevertheless, over the last decade new requirements have emerged. The codebase and complexity of the control system has grown rapidly (see Table 1) and is now larger than 10 million lines of code, distributed among more than 1000 projects, written and maintained by 180 developers. Therefore, the CERN Controls Group decided to keep a centrally maintained, unified build process for Java instead of having all teams set up their own build procedures. Many big companies like LinkedIn [6] or Netflix [7] follow the same approach, whereby a centralised tooling team provides a modern build tool with company-specific extensions to all developers to unify the development process and culture.

We chose Gradle, an open-source build automation system as the basis to implement CBNG (CommonBuild Next Generation). CBNG helps developers to avoid duplicating the same build logic across several projects and save time by providing them a common continuous integration (CI)/continuous delivery (CD) *pipeline*. CBNG also provides a simplification layer over Gradle, *enabling physicists to focus on physics*. Our developers do not have to know anything about Gradle, by reading a few pages of documentation they are able to create their projects, do development and push to production. Nonetheless, professional software developers can still benefit from the advanced built-in Gradle features.

Finally, yet importantly, the centralised release management and the unified build pipeline of CBNG aids maintenance over time. For example, with CBNG it is considerably easier than with Gradle to explore what the relationship is across projects, detect which third-party libraries are used and which are not.

Our tooling team began to investigate modern build tools in 2011 and started the development of CBNG in 2013. In the beginning of 2017, the old build system was replaced by CBNG and by the summer all the 1000 projects were built by the new tool [8].

DEVELOPMENT WORKFLOW

The development workflow is illustrated in Fig. 1. The developer either creates a new project in their Eclipse IDE [9] or checks out an existing project from version control. To start, they call the `bob eclipse` command (`bob` is the command-line shortcut for CBNG) which reads the project descriptor (step 1), downloads the dependencies of the project from the internal Maven repository (in our case JFrog Artifactory [10]) and makes them available to the IDE (step 2).

During local development, the developer typically works inside the IDE. When they are ready to publish their library or application, they make sure all code is committed back to version control and then execute the `bob release` command (step 3). This command first asks the user for their credentials and then triggers the build process on the remote *Release Server*. This service downloads the source code and builds the project. If the build is successful, it uploads the produced artifacts to Artifactory.

If the developed project is an executable product (a server or a GUI application), then additional, fully automated steps take place (step 4). To begin with, all the files are copied

[*] lajos.cseppento@cern.ch

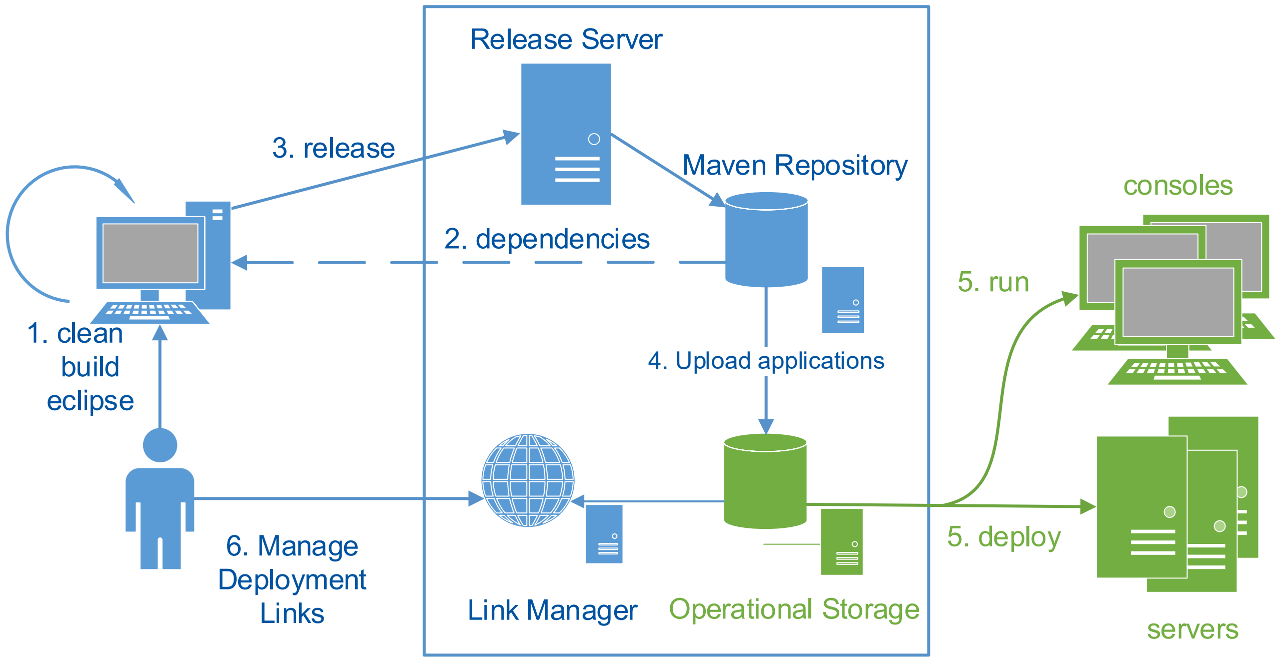

Figure 1: The Development Workflow.

from the Maven repository to the *Operational Storage*. This storage is reliable and non-stop available and is accessible through a HTTP web server, which makes it platform-independent. GUI applications can be directly started from here and server applications can be deployed to the target machine (step 5).

Developers also have the opportunity to upload *CI builds*, which are not built by the Release Server, yet the further steps for end products still take place. This feature enables our users to benefit from all the features of the pipeline during development, e.g., share executable candidate versions for manual testing.

In our environment, operation-ready artifacts can be accessed using exact version numbers or using PRO links. These links point to stable product versions that should be "generally used". These links are automatically created by the build system, but the developers can manually change them on a web interface (step 6).

PROJECT CONFIGURATION AND DEPENDENCY RESOLUTION

For our projects we use a build-tool independent XML-based descriptor format which is named `product.xml`. In this file, the developer declares the name and the version of their product and its dependencies as seen in Listing 1.

It is a general enterprise practice to avoid using exact version numbers in dependency declarations and only use a *version alias* (such as PRO for `lsa-ext-lhc` in the example), which is substituted at build time with an *exact version*. In this way, the developers do not need to update the dependency declaration in `product.xml` every time as soon as the stable version changes, e.g., when a bugfix for an internal project is re-released[1] or when the version of the Spring Framework is updated once a year. With the version aliases, we can also guarantee that everyone uses the same version of a library.

Listing 1: Basic Project Descriptor

```
<product name="lhc-injseq" version="1.0.0"
      directory="lhc/lhc-injseq">
  <dependencies>
    <dep product="lsa-core-cern" />
    <dep product="lsa-ext-lhc"
        version="PRO" preferred="true" />

    <dep product="slf4j-api" version="NEXT" />
    <dep groupId="org.jscience"
        product="jscience"
        version="4.3.1" />

    <dep product="log4j" local="true" />
    <dep product="junit" excluded="true" />
  </dependencies>
</product>
```

In our system, several aliases are available but the two most commonly used are PRO and NEXT:

PRO Refers to a *stable*, *up-to-date*, *operation-ready* version. If the version is not specified in the `product.xml`, PRO is used by default. This alias is typically the latest stable version (not necessarily the latest available version) and can be changed at any time by project owners.

[1] The maintainers of the libraries ensure that all the changes are backward-compatible and only such API points are changed or removed which are not used by any other project.

NEXT For internal packages means the latest version, for third-party packages it is also a manually set version. It is useful when developers want to test their applications to work with an upcoming version of a dependency.

While Maven repositories use POM files [11] which define JARs based on `groupId`, `artifactId`, `version`[2] (and sometimes an optional `classifier`), CBNG makes both the `groupId` and `version` fields optional in the `product.xml`: the `groupId` is automatically determined by CBNG and the default value for `version` is PRO.

CBNG also allows to distinguish different type of dependencies. For example, test dependencies such as `junit` are only needed during test execution and must not be downloaded as transitive dependencies of other projects. These dependencies are marked with `excluded="true"` in the `product.xml`.

A project can also have dependencies which are not required during compilation, should not be downloaded as transitive dependencies but must be present during application execution. Logger libraries are a good example, where developers use calls to the SLF4J API which will translate the log events to a logger implementation present on the classpath. In the example, Log4j is marked with `local="true"` to instruct the pipeline to handle this dependency properly.

Dependency resolution is a challenging topic, especially as most of our projects have dependency graphs with several layers of transitive dependencies and often resolve to over 100 JARs. For dependency resolution, CBNG uses a mechanism based on Gradle defaults. Upon dependency conflict (when two different versions of the same library are referenced in different parts of the dependency graph) the algorithm prefers the latest one.

For example, taking a product with two dependencies on PRO version in the `product.xml`:

```
<product name="my-product">
  <dependencies>
    <dep product="lib1" />
    <dep product="lib2" />
  </dependencies>
</product>
```

When the user sends command to CBNG, it first parses the project descriptor file. In order to perform compilation or other tasks, CBNG first has to determine the dependency graph using breadth-first search, which is visualized in Fig. 2:

1. CBNG determines that the PRO version of `lib1` is `1.0` and for `lib2` it is `2.0` and adds these nodes to the dependency graph.

2. CBNG detects that `lib1` depends on `slf4j-api 1.7.21` and `lib2` depends on `lib3 3.0`.

3. CBNG evaluates that `slf4j-api 1.7.21` has no dependencies and `lib3` depends on *slf4j-api 1.7.25*.

4. CBNG finds out that `slf4j-api 1.7.25` does not have any dependencies and completes the construction of the dependency graph.

5. Since at least two version of the same dependency is present on the dependency graph, *conflict resolution* is necessary.

6. During conflict resolution, CBNG prefers the higher version number of `slf4j-api`, in this case `1.7.25`.

The resolved dependencies will be `lib1 1.0`, `lib2 2.0`, `lib3 3.0` and `slf4j-api 1.7.25`.

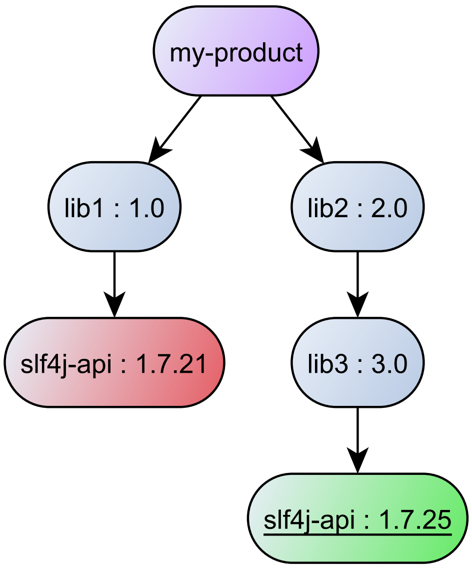

Figure 2: Dependency resolution: newer versions are preferred regardless the place in the dependency tree.

For advanced scenarios, CBNG provide several methods for developers so they can completely control the dependency resolution if they need to. For example, if the developer specifies an exact version that version will be forced and alias versions can be forced with the `preferred="true"` attribute. Dependency exclusion is also supported similarly as it can be done in Maven and Gradle.

BUILD EXECUTION AND RELEASE

When the developer calls the `bob build` command, the project is compiled, artifacts are generated and verification tasks are carried out. If compilation was successful, CBNG generates three JAR files from the code: a JAR containing the `.class` files, a JAR with the source code and another JAR with the `javadoc` documentation. If the project is an end user GUI application, the JNLP files are also created in this stage.

After that, CBNG executes the tests with JUnit. Test execution is parallel by default to provide faster build results, but the developer can disable parallelism. Optionally, it analyses test coverage using JaCoCo and runs static analysis with

[2] These coordinates are often also referenced as organization-module-revision in mainly Ivy-based tools.

SonarQube. CBNG supports these tools out-of-the-box and developers only need to call the corresponding commands, e.g., `bob coverage` or `bob sonar`. We integrated these commands in CBNG to avoid that every developer has to add the same few lines of plugin configuration to each their Gradle buildfiles.

The upload of a CI build or a release starts with POM file generation. This file declares the Maven coordinates and the dependencies of the project. CBNG also adds extra information for traceability to the POM file, such reference to the version control system, build host, build time, username, JDK version and CBNG version. Afterwards the POM file and all the JAR and JNLP files are uploaded to Artifactory. Finally, CBNG copies runnable applications from Artifactory to the Operational Storage.

Releases can be only produced by the Release Server. When the user calls the `bob release` command on their local machine, CBNG triggers the release on a dedicated Release Server which downloads the source code and performs a build in a clean environment. In this way there is no risk that the uncommitted files in the local environment of the user “pollute” the produced artifacts or affect compilation and test execution. If the build is successful, the Release Server *tags* the released code in the version control system and also adds the result of the dependency resolution to this tag. Thus, all releases are traceable and reproducible in the future even if the PRO version links of dependencies change over time.

USER EXPERIENCE

Apart from the simplified project and dependency declaration, CBNG provides other user experience improvements over Gradle. These enhancements are usually simple and were easy to put into action, yet resulted in a noticeable drop in the number of support requests and probably spared time for the developers.

By default Gradle produces a lot of log messages during a build execution. To facilitate troubleshooting upon build failure, CBNG displays a summary log after the build has finished which aggregates all the warnings and errors encountered during execution.

In addition, based on our support requests, CBNG comes with several error-detection patterns and upon a failure suggest the user “what do to next” as it can be seen in Listing 2. For example, test execution is parallel by default, but it was not the case in the old build system. Parallel test execution may fail if the test are not isolated. If parallelism is turned on, but a test fails CBNG hints to the user to turn this feature off.

If a build is successful, CBNG also prints extra information about the public changes affected by a release or a CI build as shown in Listing 3. This summary contains links to the uploaded files and the the changes to the *alias versions*.

Listing 2: Example for Troubleshooting Hints

```
------------------------------
Summary of warnings and errors
------------------------------
:test
- Try turning off the parallel execution
  with the maxParallelForks=1 property
  in case the tests are not completely
  isolated

BUILD FAILED
```

Listing 3: Example for Build Result Summary

```
------------------------------
Summary of build results
------------------------------
:release
- Artifacts:
   http://artifactory/[...]/lhc-injseq/1.1.0
- JNLP:
   http://op-store/[...]/PRO/lhc-injseq.jnlp
- PRO of lhc-injseq was updated to 1.1.0

BUILD SUCCESSFUL
```

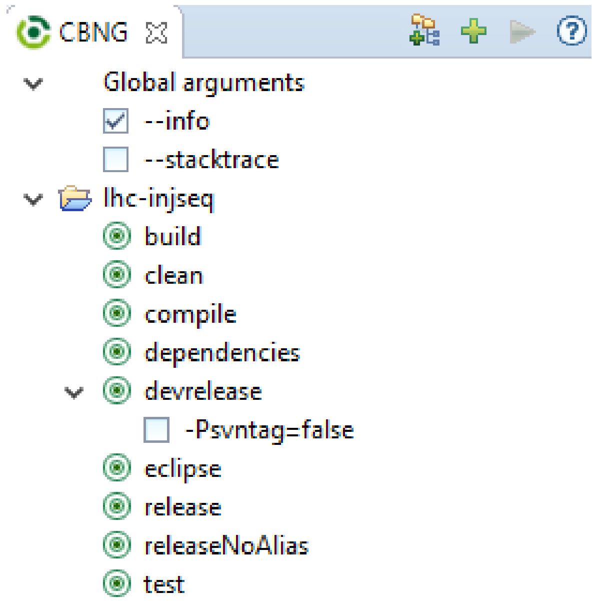

Figure 3: CBNG panel in Eclipse IDE.

CBNG also has its own Eclipse plugin (inspired by the Ant Eclipse plugin), displayed in an IDE panel as shown in Fig. 3. The developer simply drags and drops a project into the view and the plugin adds common CBNG commands and parameters. This view allows the user to run CBNG without opening a console. Over the years, we found this panel simpler and cleaner than the publicly available Gradle views for Eclipse.

FUTURE WORK

For the next year, the main goal is to enable developers to build subsystems which consist of several projects in a simpler and faster way to reduce the CI feedback time.

Multi-project Builds

It is common that the components of a server application are implemented among several projects, which declare dependencies on each other. In the current ecosystem, these projects are built and released one by one. However, by creating a multi-project (sometimes called multi-module) build among the components, all the units could be first built locally, then all the files could be uploaded in one step during the release procedure. This feature is supported by Gradle out-of-the box, however, due to the customized dependency resolution and POM generation the default behaviour produces invalid results. In order to make it work, we need to extend these customisations.

Build Promotion

The current pipeline supports to create CI builds and production releases in parallel, but does not prove a way to push a tested, well-working CI build to production. In practice, it means that after a development team has performed end-to-end tests on a development version, they would have to rebuild the project in release mode and re-run verification. We plan to provide a clean environment similar to the Release Server, which is able to create promotable CI builds.

Gradle Build Cache

Gradle has received recently a build cache feature, which hashes all the inputs of several build steps and stores their result. Developers and CI agents can share the same remote cache. When a project is built the first time, the results are stored on a remote server. The results of subsequent are simply fetched from the remote server if the input has not changed. According to reports from the Gradle team, build time can be reduced by 25% [12].

CONCLUSION

CBNG has proven itself being a stable and modern build tool able to build controls software written in Java for the upcoming years. It was a big step towards how other companies build Java software. The evolved pipeline enables developers to only use a few commands to get their application pushed to production. In the end, Gradle turned out as an excellent choice for satisfying our requirements.

REFERENCES

[1] Apache Ant,
`https://ant.apache.org`

[2] G. Kruk *et al.*, "Development Process of Accelerator Controls Software", in *Proc. ICALEPCS'05*, Geneva, Switzerland, Oct. 2005, paper FR_5-6O

[3] JJAR: Jakarta JAR Archive Repository,
`https://commons.apache.org/dormant/jjar`

[4] Apache Maven Project,
`https://maven.apache.org`

[5] Gradle Build Tool,
`https://gradle.org`

[6] S. Faber, "Building Great Tools for Developers at LinkedIn, with Gradle, 6 Years in a Row", *Gradle Summit 2017*, Palo Alto, CA, USA, Jun. 2017,
`https://summit.gradle.com/session/39273`

[7] M. McGarr, "Dependencies, Distributed Code and Engineering Velocity", *Gradle Summit 2017*, Palo Alto, CA, USA, Jun. 2017,
`https://summit.gradle.com/session/39245`

[8] E. Fejes, "Adapting Gradle for the CERN Accelerator Control System", *Gradle Summit 2017*, Palo Alto, CA, USA, Jun. 2017,
`https://summit.gradle.com/session/39268`

[9] Eclipse IDE,
`http://eclipse.org`

[10] Artifactory – Universal Artifact Repository Manager,
`https://www.jfrog.com/artifactory`

[11] Maven POM Reference,
`https://maven.apache.org/pom.html`

[12] Introducing the Gradle Build Cache,
`https://blog.gradle.org/introducing-gradle-build-cache`