Build System Overview: Yet Another Metrics Collector (YAMeC)

C. Marcus DiMarco & Brendan Gibbons

MSCS710 Project

Spring 2025

Table of Contents

Build System Components

Maven

Maven is the main package manager and build system utilized for the YAMeC project. It is a staple build management tool used for Java projects of varying scopes. Maven offers dependency management, meaning that it will download the appropriate versions of external packages which our Java code uses and include them within the compiled source code for our project. Maven also supports plugins which can help to automate the build, testing, and delivery of YAMeC, including packaging the program into a Windows executable file and building and linking the project's native (C++) code to the Java code.

Plugins

Maven plugins can accomplish many different tasks, which are specified by using executions and goals. Executions are the definition of some task for the build system to do at a certain point (or phase) in the build process and are configured using configuration parameters. Goals are what executions seek to accomplish, which determines the actual task the build system accomplishes using those configuration parameters at a specific phase in the build process.

Spring Boot

Spring Boot is a Maven application framework plugin which facilitates the creation of web application hosted by a Java server. This will be used by the frontend code of our system to display the user interface to users. Spring Boot also plays a role in the build process of the application by repackaging all included code and dependencies into a single executable JAR file, including the web application components of the application and all dependencies. This repackaging is accomplished using the `repackage` goal within Spring Boot.

IZPack

IZPack is a Maven plugin which automatically builds an installation executable for the application upon successful builds. Conditions for the installer, including checks and installations for external dependencies, steps for the installation process, the path for the application and its components and dependencies to be installed in, and so on are specified using an XML document.

YAMeC will be distributed in a .msi executable file for Windows systems to streamline application setup for end users. Upon project completion, this executable will be made by IZPack upon successful build and will include not only the compiled code by itself, but also automatically be able to install the Java 23 and SQLite dependencies which the application depends on to run. If YAMeC was to become a cross-platform application, IZPack would also be able to create installers for Linux and macOS.

ExecMaven and CMakeMaven

ExecMaven and CMakeMaven are Maven plugins which allow native executables, DLLs, and object files to be compiled as part of the build process for the Maven project. ExecMaven allows a standard C or C++ compiler, such as g++, to be called as part of the build process, whereas CMakeMaven calls CMake build routines as part of the build process. This means that whenever the Java project is built, it should always include the most up-to-date version of the native C++ code with it.

During our development process, we found it easier to build our C++ code using CMake than manually compiling the C++ code through the compiler directly, such as by using g++. So, instead of using ExecMaven, we chose to use CMakeMaven to run the Make script we had already written into the CMakeLists.txt file. Currently, two separate executions of the CMakeMaven plugin occur. The first one, the `generate` goal execution, sets up the build tools for use in the compilation of the native code. The second one, the `compile` goal execution, builds the native code into executables. It may also be useful to include a `test` goal execution in the CMake routine so that some unit testing can be built right into the CMake build process, though that is not currently included.

Build Helper Maven

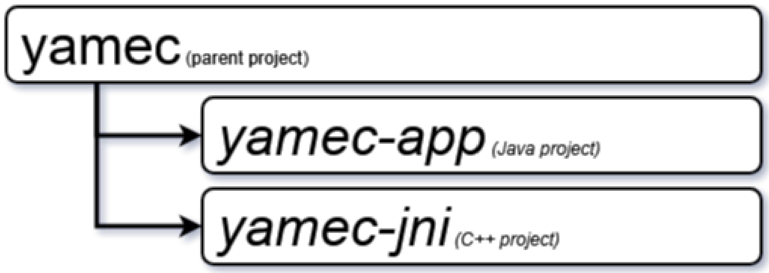
Build Helper Maven is a general-purpose plugin for Maven which provides a series of automation and build streamlining capabilities. These capabilities include the ability to automatically fill properties in the build specification with local system and build information or values based on regular expressions. It is also capable of controlling which resources to include in the build. YAMeC most notably uses the `attach-artifact` goal of the Maven plugin to include the compiled DLL binary from the C++ code in the Java JAR file created by the build system.

CMake

CMake is the build system utilized to manage compilation of all native C++ code within the YAMeC project. It utilizes a text file called `CMakeLists.txt` to specify conditions for the build, including the C++ standard, files to include in the compilation, and external dependencies and libraries to include and link to the compiled code. Types, methods, and so on which are needed for C++ code to return data to Java code are facilitated by the Java Native Interface dependency which CMake includes in the project build. If YAMeC was able to be developed for multiple platforms within the timespan of this project, CMake would also be responsible for compiling the appropriate C++ code based on the platform which the compiler is being run on.

Project File Structure

To complete a successful build, Maven expects the project files to be in a specific structure. This is because the Maven project utilizes three separate Maven build scripts, total: one for the Java-based code, one for the C++ code to be compiled through CMake Maven, and one which contains both the Java and C++ projects and executes their build processes and all general building, testing, and deployment tasks for the complete system. The project directory structure's hierarchal nature can be described by the figure below.



The `yamec-app`, `yamec-jni`, and `yamec` directories within the project files each contain a `pom.xml` file which act as each of the build scripts, with the `yamec-app` and `yamec-jni` directory being a subdirectory to the `yamec` directory. The `yamec` directory also includes the `CMakeLists.txt` file, which specifies build directions for CMake. The entire project can then be packaged into a single Java JAR file, which will then be packaged into an `.msi` installer for Windows systems via IZPack for delivery of the final product.

POM Files

Each of the `pom.xml` files within the project directory specify a Maven project which includes all files in the directory they reside in and below. These Maven projects manage dependencies and plugins needed for their own subprojects. The `yamec-parent` pom.xml (located in the `yamec` directory) specifies key information for the entire project, including the version of Spring Boot and Launch4J in use, the names of the subprojects (`yamec_jni` and `yamec_app` for each of the identically named subdirectories), the main compiler version to be used for packaging the entire project together, and plugins and dependency versions utilized in the entire project. For example, the `yamec-parent` project POM specifies the version of Spring Boot and CMakeMaven which are used throughout the entire project and contains the dependency information for IZPack. It is also the Maven project which specifies execution of the Spring Boot `repackage` goal.

Meanwhile, the `yamec_jni` and `yamec_app` project POM handle dependencies and behavior which only affects the build process of their respective projects. For example, the `yamec_jni` project POM specifies the execution of the CMakeMaven plugin to complete both its `generate` and `compile` goals, executing the CMake build script to compile its C++ code. It also specifies the execution of the BuildHelperMaven plugin to complete its `attach-artifact` goal, including the compiled native code with the rest of the project upon packaging. Additionally, the `yamec_app` project POM specifies the execution of the Spring Boot plugin so that the Java code included in the project can be used as part of a Spring Boot web application and specifies inclusion of the SQLite JDBC dependency so that the Java code can manage the database of historical system metrics through an API.

CMakeLists.txt File

The `CMakeLists.txt` file is used as a script to handle the build of C++ code into a dynamic link library (DLL) which is to be accessed by the Java Native Interface in Java. For C++ code to be built, however, the compiler needs to be able to link the C++ code written using the JNI API for C++ to the `jni.h` header file included in the files of Java's home directory. CMake supports a command called `find_package()` which looks on the development system for a copy of the JNI file on the developer's system automatically, which was especially useful in ensuring that builds were possible on both of our development systems. CMake can also read system environmental variables as a fallback for finding the JNI header files. CMake also allows only specific files to be included in builds and for builds to be aborted if they are executed on specific operating systems. This allows our application to enforce its current compatibility with Windows at the compilation step while still allowing native code to be written in the future for other operating systems with less required changes to the overall build structure. Finally, CMake allows specification of the C++ version to compile code for and other configuration details, such as a name to give output files and the directories to copy the output files to. This is done in a far more human-readable and far less error-prone way than using a command-line interface with many different arguments.

Building the Project

The YAMeC build process involves several steps that are orchestrated by Maven to ensure all components are properly compiled, tested, and packaged. It consists of a multi-module Maven structure with native C++ integration through CMake.

Prerequisites

Before building the project, ensure the following prerequisites are installed:

- Java Development Kit (JDK) 23 or later
- Maven 3.8 or later
- CMake 3.20 or later
- C++ compiler compatible with your platform
    - The Visual Studio 17 2022 C++ compiler is preferred, as it includes built-in references to certain Windows proprietary header and library files which are used by the native C++ application.
    - Additional configuration of the CMakeLists.txt file is required with another compiler.

Additionally, ensure JAVA_HOME is properly set in the system environmental variables.

Basic Maven Build Commands

To build the entire project from the command line, navigate to the project root directory (containing the parent POM) and run:

```
mvn clean install
```

This command:

1. Cleans previous build artifacts
2. Compiles the Java code in both modules
3. Triggers the CMake build process for the native C++ code in yamec-jni
4. Packages the application into an executable JAR
5. Attaches the native DLL to the build artifacts

For specific build phases or modules:

```
mvn clean                          # Remove previous build artifacts
mvn compile                        # Compile Java source code only
mvn package                        # Package the application
mvn install -pl yamec-jni          # Build only the JNI module
mvn install -pl yamec-app          # Build only the app module
mvn spring-boot:run                # Runs the built program from the
                                   # build directory using the JAR
```

Native Code Build Process

The native C++ code is built using the cmake-maven-plugin configured in the yamec-jni module. When building this module, Maven will:

1. Generate CMake files using Visual Studio 17 2022 generator
2. Compile the C++ code in Release configuration
3. Attach the resulting yamecjni.dll as a build artifact with classifier “native-win64”

Troubleshooting the Build Process

Q: Why is my Maven build failing when trying to compile the native code?

A: Check that JAVA_HOME is correctly set to a compatible JDK 23 installation. The CMake build process directly references this environment variable to locate JNI headers.

Q: I'm getting a "FATAL_ERROR" about an unsupported OS. What does this mean?

A: The YAMeC project currently only supports Windows environments as specified in the CMakeLists.txt. The build will terminate if attempted on Linux or macOS systems.

Q: CMake fails to generate build files. How do I fix this?

A: Ensure Visual Studio 2022 with C++ development tools is properly installed. The cmake-maven-plugin specifically requires the "Visual Studio 17 2022" generator.

Q: The build can't find the PDH library. What should I do?

A: Verify that the Performance Data Helper (PDH) library, which is required by the native code, is available on your system. It's typically included with Windows SDK installations.

Q: I'm getting JNI function signature errors. How do I resolve them?

A: Compare the JNI header files generated in the target directory against your C++ implementations. Function signatures must match exactly between Java native method declarations and C++ implementations.

Q: The native DLL isn't being attached to the build artifacts. What went wrong?

A: Check that the paths in the build-helper-maven-plugin configuration correctly point to ${project.build.directory}/cmake/Release/yamecjni.dll. The build directory structure must match this path.