Utilisation de Maven Lite et description de toutes les fonctionnalités

- Voir la documentation Française
- See the PDF documentation

Table des matières

List of Options

- `-f`, `--file`: Load options from a configuration file.
- `-cr`, `--create`: Create the project structure along with a default configuration file.
- `-mvc`, `--model-view-controller`: Specify to the '`--create`' option to create the structure of an MVC project.
- `-c`, `--compilation`: Compile the project.
- `-l`, `--launch`: Launch the project.
- `-cl`, `--compile-launch`: Compile and launch the project (equivalent to -c -l).
- `-lc`, `--launch-compile`: Compile and launch the project (equivalent to -c -l).

- `-q`, `--quiet`: Suppress the display of java in the terminal during project execution.
- `-v`, `--verbose`: Display the executed commands.
- `-ex`, `--exclude`: Exclude files and folders from compilation.
- `-cj`, `--compile-jar`: Create a jar file of the project.
- `-lj`, `--launch-jar`: Launch an executable jar file.
- `-it`, `--integrate-test`: Integrate unit tests into the project.
- `-s`, `--source`: Folder containing java files to compile.
- `-t`, `--target`: Output folder for compiled files.
- `-r`, `--resources`: Folder containing resource files to copy into the output folder for compiled files.
- `-cp`, `--classpath`: Specify the classpath to use during compilation and launch.
- `-lib`, `--libraries`: Folder containing jar files used by the program.
- `-args`, `--arguments`: All arguments to pass to the main class.
- `-m`, `--main`: Main class to launch.
- `-e`, `--encoding`: Change the encoding of the java files to be compiled.
- `-exp`, `--export`: Create an executable jar file to launch the project without installing MavenLite.
- `-mvn`, `--maven`: Convert the project to a Maven project.
- `-V`, `--version`: Display the version.
- `-h`, `--help`: Display help and exit.
- `-clr`, `--clear`: Clear files in the output folder for compiled files.

Basic Usage

- Navigate to your Java project folder and execute the following command:

```
mvnl [options] [arguments]
```

Command Line

The command line is the classic way to use Maven Lite, although it is less convenient than using a configuration file.

You can include as many options as you want in any order.

Just place the arguments with spaces in quotes, for example, `-args "your argument"`. On Windows, it is impossible to include the `"` character in arguments.

- It does not support comments.
- You can use options in the format `--option` or `-o`. You can use any option listed in the list of options.
- You can pass arguments with spaces to the main class of your project by using double quotes, for example, `mvnl -args "your argument"`.
- You can place multiple options on the same line, separating them with a space, for example, `mvnl -q --verbose --arguments "your arguments"` or `mvnl -quiet -v -args "your arguments"`.
- You can escape special characters with a backslash `\`, for example, `-args "examp\"le"` on Linux and MacOS only.

Example Command Line

- Example of a command line with arguments
  - In this example, we will compile and launch a Java project with the source folder `src/main/java`, the output folder for compiled files `target`, the folder containing jar files `src/main/resources/lib`, and display of executed commands.

```
mvnl --source src/main/java --target target --libraries src/main/resources/lib --verbose -cl
```

Configuration File

The use of a configuration file is done with the `--file` or `-f` option.

The configuration file is unique to each project and configures the options that Maven Lite should use.

The default name of the configuration file is `LPOM.conf` and must be at the root of the project. You can rename it, but in that case, you need to specify its name when using the option, for example, `mvnl -f myFile.extension`.

If you want to add your system CLASSPATH to the Maven Lite CLASSPATH in the configuration file, use the term `$CLASSPATH` in uppercase.

You can include as many options as you want in any order.

- It supports comments using the `#` character at the beginning of the line.
- You can use options in the format `--option` or `-o`. You can use any option listed in the list of options except for the `--file` and `-f` options.
- You can pass arguments with spaces to the main class of your project by using double quotes, for example, `-args "your argument"`.
- You can place multiple options on the same line, separating them with a space, for example, `--quiet -v --arguments "your arguments"` or `-q --verbose -args "your arguments"`.
- You can escape special characters with a backslash `\`
    - Special characters in the configuration file are: `\` and `"`. Here are examples of use:
        - `-args "examp\"le"` becomes `examp"le`
        - `-args "examp\\\\\"le` becomes `examp\"le`
        - `-args "\-example"` becomes `\-example`
        - `-args "\--example"` becomes `\--example`
        - `-args "examp\\le"` becomes `examp\le`
        - `-args "examp\le"` becomes `examp\le`
        - `-args "examp\\\le"` becomes `examp\\le`
        - `-args "examp\\\"le"` becomes `examp\"le`
        - `-args "examp#le"` becomes `examp#le`
        - `-args "examp\ le"` becomes `examp\ le`

Example Configuration File

- A typical example of a configuration file is as follows:

```
# Project source
--source src/main/java

# Output folder for compiled files
--target target

# List of libraries to add to the classpath
--libraries src/main/resources/lib

# Display executed commands
--verbose

# Add the system classpath to the Maven Lite classpath
--classpath CLASSPATH

# Add an argument to the main class
--arguments "argument 2 (in config)"
-args "argument 3 (in config)"

# Suppress java output and display executed commands
--quiet --verbose
```

Project Creation

The use of this option is done with the `--create` or `-cr` option.

It is impossible to create a project with a space in the name. It is impossible to create a project with a name that is already used by a file or folder.

You can create a project in the current folder with the following command:

```
mvnl --create ./
```

By default, the project name is `NewProject`, and the structure is as follows:

```
NewProject
├── LPOM.conf
├── src
│   ├── main
│   │   └── java
│   │       └── HelloWorld.java
│   └── resources
│       └── lib
└── target
```

Model-View-Controller

The use of this option is done with the `--model-view-controller` or `-mvc` option.

It allows specifying to the '`--create`' option to create the structure of an MVC project.

- To create an MVC project, you can use the following command

  ```
  mvnl --create ProjectName --model-view-controller
  ```

  - Project structure:

    ```
    ProjectName
    ├── LPOM.conf
    ├── src
    │   ├── main
    │   │   └── java
    │   │       ├── controller
    │   │       │   └── Controller.java
    │   │       ├── model
    │   │       └── view
    │   └── resources
    │       └── lib
    └── target
    ```

Project Compilation

The use of this option is done with the `--compilation` or `-c` or `--compile-launch` or `-cl` or `--launch-compile` or `-lc` option.

This option allows compiling the project.

- Example of using the Compilation option

  - In this example, we will compile a Java project with the default source and output folder for compiled files.

  ```
  mvnl -c
  ```

Project Launch

The use of this option is done with the `--launch` or `-l` or `--compile-launch` or `-cl` or `--launch-compile` or `-lc` option.

This option allows launching the project.

- Example of using the Launch option
    - In this example, we will launch a Java project with the default source and output folder for compiled files.

```
mvnl -l
```

Quiet

The use of this option is done with the `--quiet` or `-q` option.

This option suppresses the display of java in the terminal during project execution.

- Example of using the Quiet option
    - In this example, we will compile and launch a Java project with the display of java in the terminal disabled, as well as the default source folder and output folder for compiled files.

```
mvnl -q -cl
```

Verbose

The use of this option is done with the `--verbose` or `-v` option.

This option displays the executed commands.

- Example of using the Verbose option
    - In this example, we will compile and launch a Java project with the display of executed commands by Maven Lite, as well as the default source folder and output folder for compiled files.

```
mvnl -v -cl
```

Exclusion of Files and Folders

The use of this option is done with the `--exclude` or `-ex` option.

This option allows excluding Java files and folders from the compilation.

If you want to exclude a folder, you should use the relative path of the folder from the project's source folder and without using `./` or `../`.

If you exclude a folder, all the files and folders it contains will also be excluded.

Note: If you exclude the `test` folder, all files or folders containing the word `test` in their name will also be excluded.

- Example of using the Exclusion option
    - In this example, we will compile and launch a Java project with the exclusion of the file `Main2.java` and all files and folders contained in the `tests` folder, as well as the default source folder and output folder for compiled files.

```
mvnl -ex Main2.java tests -cl
```

Compilation into a Jar File

The use of this option is done with the `--compile-jar` or `-cj` option.

This option allows creating an executable jar file of your project.

The naming conventions for jar files are as follows: `<name>-<M>.<m>.<b>.jar`.

- `<name>`: Name of the project.
- `<M>`: Major version number. It starts at 1 and is incremented with each new version that is not compatible with the previous one and has significant changes.
- `<m>`: Minor version number. It starts at 0 and is incremented with each new version compatible with the previous one and has small changes, and resets to 0 with each major version change.
- `<b>`: Build version number. It starts at 0 and is incremented with each bug fix and resets to 0 with each minor or major version change.
- Example of using the Compilation into a Jar File option
    - In this example, we will create a jar file of the Java project with the default output folder for compiled files.

```
mvnl -cj
```

Launching a Jar File

The use of this option is done with the `--launch-jar` or `-lj` option.

This option allows launching the executable jar file of your project.

- Example of using the Launching a Jar File option
    - In this example, we will launch the executable jar file `MyProject-1.0.0.jar` located in the `target` folder.

```
mvnl -lj target/MyProject-1.0.0.jar
```

Integration of Unit Tests

The use of this option is done with the `--integrate-test` or `-it` option.

This option allows integrating unit tests into the project, i.e., creating the unit test directory structure, unit test files, and also copying the jar files used by JUnit into the libraries folder.

If you use it together with the `-cr` or `--create` option, Maven Lite will create the unit test directory structure as well as a default test file containing a unit test for the main class.

If you use it after creating the project, Maven Lite will create the unit test directory structure as well as a unit test file for each Java file in the project.

- Example of using the Integration of Unit Tests option
    - In this example, we will create a Java project with the integration of unit tests.

```
mvnl -cr -it
```

- Project structure:

```
NewProject
├── LPOM.conf
├── src
│   ├── main
│    │    └── java
│    │         └── HelloWorld.java
│    ├── resources
│    │    └── lib
│    │         ├── hamcrest-core-1.3.jar
│    │         └── junit-4.13.2.jar
│    └── test
│         └── java
│              └── HelloWorldTest.java
└── target
```

- Example of using the Integration of Unit Tests option with an MVC project
  - In this example, we will create an MVC Java project with the integration of unit tests.

```
mvnl -cr -mvc -it
```

  - Project structure:

```
NewProject
├── LPOM.conf
├── src
│    ├── main
│    │    └── java
│    │         ├── controller
│    │         │    └── Controller.java
│    │         ├── model
│    │         └── view
│    ├── resources
│    │    └── lib
│    │         ├── hamcrest-core-1.3.jar
│    │         └── junit-4.13.2.jar
│    └── test
│         └── java
│              ├── controller
│              │    └── ControllerTest.java
│              ├── model
│              └── view
└── target
```

Project Source

The use of this option is done with the `--source` or `-s` option.

This option allows specifying the folder containing the Java files to compile.

The source folder is not the root of the project but the first folder containing the Java files to compile. The Java files in this folder do not have a package.

- Example of using the Project Source option
  - In this example, we will compile and launch a Java project with `src` as the source folder and the default output folder for compiled files.

```
mvnl -s src -cl
```

Compiled Files Destination

The use of this option is done with the `--target` or `-t` option.

This option allows specifying the output folder for compiled files, the input folder for launch files, the output folder for jar files, and the input folder for .class files to include in the jar.

You don't need to create the output folder for compiled files; Maven Lite will do it for you.

You don't need to add the output folder for compiled files to the classpath; Maven Lite will do it for you.

- Example of using the Compiled Files Destination option
  - In this example, we will compile and launch a Java project with `src` as the source folder and `target` as the output folder for compiled files.

  ```
  mvnl -s src -t target -cl
  ```

- Example of using the target option to create a jar file
  - In this example, we will create a jar file of the Java project with `target` as the output folder for compiled files. You can find the jar file in the `target` folder and launch the jar file with the command `mvnl -lj target/MyProject-1.0.0.jar`.

  ```
  mvnl -t target -cj
  ```

Resources

The use of this option is done with the `--resources` or `-r` option.

This option allows specifying the folder containing resource files to be copied to the output folder for compiled files when creating a jar file.

- Example of using the Resources option
  - In this example, we will create a jar file of the Java project with `src/main/resources` as the folder containing resource files to be copied to the output folder for compiled files.

  ```
  mvnl -r src/main/resources -cj
  ```

Classpath

The use of this option is done with the `--classpath` or `-cp` option.

This option allows specifying the classpath to use during compilation and execution.

You can add the system classpath by using the term `$CLASSPATH` in uppercase in the configuration file.

In the command line, you can use the system classpath by using the term `$CLASSPATH` under Linux and MacOS. Under Windows, you can use the term `%CLASSPATH%`.

- Example of using the Classpath option
  - In this example, we will compile and launch a Java project with the system classpath, the `src/main/resources/lib` folder, and the `target` folder in the project's classpath.

  ```
  mvnl -cp $CLASSPATH target src/main/resources/lib -cl
  ```

Libraries

The use of this option is done with the `--libraries` or `-lib` option.

This option allows specifying the folder containing the jar files used by the program. All jar files will be added to the classpath during compilation and execution.

You can create subfolders in the libraries folder to better organize your jar files, and Maven Lite will take them into account.

- Example of using the Libraries option
  - In this example, we will compile and launch a Java project with the `src/main/resources/lib` folder as the folder containing the jar files used by the program.

```
mvnl -lib src/main/resources/lib -cl
```

Arguments

The use of this option is done with the `--arguments` or `-args` option.

This option allows specifying all the arguments to pass to the main class of your project.

These arguments will be passed to the main class in the order they are passed to Maven Lite.

Note: Under Windows, it is impossible to use the character `"` in arguments passed in the command line, so it should be passed in the configuration file.

Example of using the Arguments option

- Example of a command line with arguments and a configuration file
  - In this example, we will compile and launch a Java project with arguments using the command line and a configuration file.

```
mvnl -args "argument 1" -f --arguments "argument 3" "argument 4" -args -cl
```

  - Configuration file

```
# Add an argument to the main class
-args "argument 2 (in config)"
```

  - Arguments passed to the main class

```
String args = new String[]{"argument 1", "argument 2 (in config)", "argument 3", "argument 4"};
```

Arguments in the command line

Command line under Linux and MacOS

- To pass arguments to the main class of your project with the `-args` and `--arguments` options to avoid any escaping issues.
  - Special characters in the command line are: `\`, `"`
    - `-args '"examp\"le"'` becomes `examp"le`
    - `-args '"-example"'` becomes `-example`
    - `-args '"--example"'` becomes `--example`
    - `-args '"examp\\le"'` becomes `examp\le`
    - `-args '"examp\\\"le"'` becomes `examp\"le`.

Command line under Windows

// TODO: Verify that special characters work under Windows

- To pass arguments to the main class of your project with the `-args` and `--arguments` options to avoid any escaping issues.
    - Special characters in the command line are: `\`
        - `-args '"examp\\\"le"'` becomes `examp\"le`
        - `-args '"-example"'` becomes `-example`
        - `-args '"--example"'` becomes `--example`
        - `-args '"examp\\le"'` becomes `examp\le`
        - `-args '"examp\le"'` becomes `examp\le`
        - `-args '"examp\\\le"'` becomes `examp\\le`
        - `-args '"examp\\\\\"le"'` becomes `examp\"le`
        - `-args '"examp#le"'` becomes `examp#le`
        - `-args '"examp\ le"'` becomes `examp\ le`

Arguments in the configuration file

// TODO: Verify special characters

- To pass arguments to the main class of your project with the `-args` and `--arguments` options to avoid any escaping issues.
    - Special characters in the configuration file are: `\`, `"`
        - `-args "examp\"le"` becomes `examp"le`
        - `-args "\-example"` becomes `\-example`
        - `-args "\--example"` becomes `\--example`
        - `-args "examp\\le"` becomes `examp\le`
        - `-args "examp\le"` becomes `examp\le`
        - `-args "examp\\\le"` becomes `exemp\\le`
        - `-args "examp\\\"le"` becomes `exemp\"le`
        - `-args "examp#le"` becomes `exemp#le`
        - `-args "examp\ le"` becomes `exemp\ le`

Main Class

The use of this option is done with the `--main` or `-m` option.

This option allows specifying the main class to be launched with the package in the form `package.MainClass`. This option is useful only if you have multiple main classes in your project; if not, Maven Lite will automatically find and use the main class of the project.

- Example of using the Main Class option
    - In this example, we will compile and launch a Java project with the main class `Main2` from the `com.example` package and the default output folder for compiled files.

```
mvnl -m com.example.Main2 -cl
```

Encoding

The use of this option is done with the `--encoding` or `-e` option.

This option allows specifying the encoding of the Java files to compile.

- Example of using the Encoding option
    - In this example, we will compile and launch a Java project with the `ANSI` encoding and the default output folder for compiled files.

```
mvnl -e ANSI -cl
```

Export

This option is used with the `--export` or `-exp` option.

This option allows creating an executable .class file configured for your project, enabling you to compile and run your project without installing Maven Lite.

The only options that the .class file accepts are the main class options, `--arguments`, `-args`, launch, and compilation options, `--launch`, `-l`, `--compile`, `-c`, `--compile-launch`, `-cl`, `--launch-compile`, `-lc`, and the config file option `--file`, `-f`, and that's it. Other options will not be taken into account.

It is important to use the configuration file when using the `--export` or `-exp` option because it specifies the parameters that the executable .class file will use.

The purpose of this option is to be able to launch your project via a single file without arguments and without having to install Maven Lite. This allows people who do not have Maven Lite to use your project.

- Example of using the Export option
  - In this example, we will create an executable .class file configured for our project, allowing us to compile and run our project without installing Maven Lite.

```
mvnl -f -exp
```

  - Configuration file

```
# Source
--source src/main/java

# Output folder for compiled files
--target target

# List of libraries to add to the classpath
--libraries src/main/resources/lib

# Display executed commands
--verbose
```

Convert to Maven Project

This option is used with the `--maven` or `-mvn` option.

This option allows converting your project to a Maven project by creating a pom.xml file and moving files if necessary.

Version

This option is used with the `--version` or `-V` option.

This option displays the version of Maven Lite as well as the location of the main Maven Lite file, Java version, build type, Java runtime used, system language, encoding platform used by Maven Lite, operating system name, operating system kernel version, and system architecture.

- Example

```
Maven Lite 2.0.0
Maven Lite home: /usr/local/etc/maven-lite/
Java version: 17.0.9, vendor: Private Build, runtime: OpenJDK Runtime Environment
Default locale: fr_FR, platform encoding: UTF-8
OS name: "Linux", version: "6.5.0-14-generic", architecture: "amd64"
```

Help

This option is used with the `--help` or `-h` option.

This option displays the list of options along with their description, the number of arguments they take, and their default value if they have one, as well as a link to the documentation.

Clear Compiled Files

This option is used with the `--clear` or `-clr` option.

This option allows deleting all files in the output folder for compiled files. This option frees up disk space and ensures a clean jar file.

Example, Features, and Limitations

Retour